Non-linear Perspective Widgets for Exploration of Complex Data-sets

Nisha Sudarsanam, Cindy Grimm, and Karan Singh

Abstract—Viewing data sampled on complicated geometry such as a human pelvis or helix is hard. A single camera view is not sufficient to view different parts of such a complicated dataset. Multiple views are needed in order to completely examine the structure. In this paper, we present a general tool-kit consisting of a set of versatile widgets, namely the unwrap, the clipping, the fish-eye, and the panorama widgets, each of which encapsulates a variety of complicated camera placement issues and then combines these several camera views into a single view in real-time. These non-linear views give a more complete visualization of the structure without modifying the underlying geometry of the dataset. Multiple widgets can be combined to facilitate better understanding of the underlying structure.

Index Terms—Non-linear perspective, User interfaces, Visualization, Camera control, rendering

1 Introduction

Faster computers, new data acquisition techniques, and increased storage capacity are all contributing to the generation of highly complex data-sets. Data-sets can either be composed of inherently complex geometry (Figure 1) or a large collection of relatively simple geometry (Figure 2).

Finding the "best" view in which **all** of the interesting features can be seen distinctly is challenging. As certain features come into a view, others move out. Also, it can be quite difficult to manipulate a virtual camera to view a particular feature. Knowing which of the many camera parameters to change in order to get a desired view of the scene can be quite an art. In this paper, we propose combining sub-views, each of which allows particular features to be seen clearly, into a single composite view. Data-sets can then be viewed through this composite view instead of multiple single views. The idea of combining multiple views into a single one is not new. This concept has been illustrated in the works of artists such as M.C. Escher, David Hockney and Picasso. These artists deliberately introduced distortions of perspective in their work for several reasons such as a desire to create artistic effects or as mood changes or for controlling the composition of a scene. The perspective distortion introduced in these examples still relied on traditional linear perspective, but only locally. Singh[2002] referred to such distorted perspective as a "non-linear perspective" view.

In this paper, we use this non-linear perspective idea to enable the viewing of data-sets through a single composite view. This composite view is built by continously combining a set of individual views (sub-views). In each sub-view, certain user-specified regions are clearly displayed. To simplify building these composite views, we present a tool-kit of simple widgets, each of which encapsulates a set of complicated camera transformations. Some of the views generated by our framework are shown in Figure 4, Figure 19 and Figure 14. We believe that the compositions encompassed by our widgets are useful for exploring a large variety of data sets. Thus, even though our widgets have been applied to specific examples in this paper, we believe that they are general enough to be applied to other situations.

2 Approach

Each of our non-linear perspective widgets represents a 3D volume and a 2D area in screen-space. We refer to the 3D volume as the "source volume" and the screen-space area as the "destination area". The source volume specifies the region of the data-set which the user is interested in viewing differently. The destination area controls the "after-projection" aspects of the region such as placement on the screen and projected size of the volume. For example, in Figure 3(a), the maroon region encapsulates the source volume while in Figure 3(e) the red box indicates the position and scale of the corresponding the destination area.

The user first specifies a default view, which controls the overall view (global perspective) of the data-set. Next, the user selects a source volume using our 3D widget. The widget controls the position, orientation and scale of the selected volume. In previous approaches [7] users were required to manually specify the destination area. This caused problems because, as the user changes the default view, they also had to change the corresponding destination area. To simplify our interface, we introduce automatic methods for calculating the destination area for each widget. Thus in general, the user only needs to specify a volume in order to generate a simple non-linear projection (Figure 4).

The system automatically creates and assigns different cameras to different parts of the data set, based on a given data point's location relative to the source volume. Exactly how these cameras are constructed from the default one depends on the particular widget being used. The net effect is that data lying within the source volume is viewed differently than data that lies outside. We refer to these new cameras as "local cameras" (these correspond to the sub-views that make up a composite view).

More specifically, each widget modifies a subset of the default camera's parameters to create each local camera. The remaining set of parameters are the same as the corresponding parameters in the default camera. For example, a widget could change just the zoom parameter of all of the local cameras, creating a fish-eye effect in the source volume. In order to ensure a continous transition between the default view and the local views, the cameras smoothly interpolate to the default one at the boundaries of the source volume. In order to combine cameras, we build a fall-off function around the selected volume (Section 9).

Finally, our framework makes no assumption regarding the representation of the data. The only constraint we require is that the data be sufficiently sampled. If, for example, meshes are used to represent the data, then the triangulation should be sufficiently dense. The exact description of a source volume along with the blending techniques used varies from one widget to another. The specifics of each widget will be expanded upon in the following sections.

3 Contributions

This paper presents a novel, real-time, view-based deformation toolkit for visualizing complex data-sets. Multiple widgets can be combined to visualize different features within a single view. Our real-time interface allows the user to continue to view these features even as the

- *Nisha Sudarsanam is with Washington University in St Louis, E-mail: nsudarsa@cse.wustl.edu*
- *Cindy Grimm is with Washington University In St Louis, E-mail: cmg@cse.wustl.edu*
- *Karan Singh E-mail: karan@dgp.toronto.edu*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 2 November 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

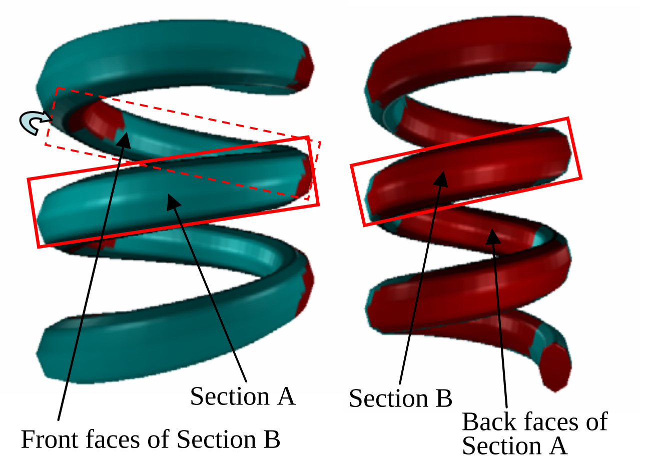


Fig. 1. This figure shows the front (left-side) and back (right-side) faces of a helix. Trying to examine the front-faces of section A and section B simultaneously in a single camera view is impossible. The unwrap widget rotates the back faces of section B (shown in dashed lines) to show its front faces while leaving the rest of the helix to be seen as is in the front-view (see Figure 7).

global view of the data-set changes. All of the widgets presented here are built on top a very general non-linear perspective rendering framework which allows special kinds of projections such as panoramas to be defined easily.

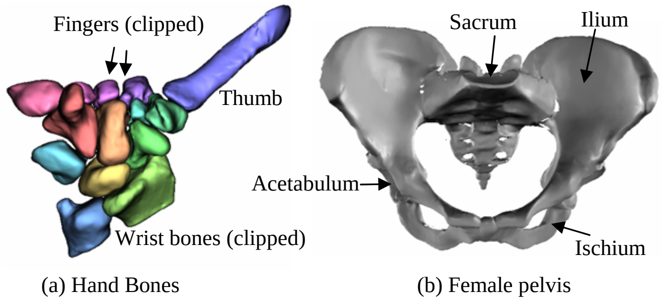
4 Related Work

A large body of research has been devoted to interactive viewing of volume data-sets. However, these methods actually deform the underlying geometry of the data-set in order to expose interesting structures ([5, 10, 17]). The problem with such methods is that the deformation depends on the current view of the model. Changing the view of the model results in an inconsistent deformation.

Rademacher et al. [19] and Martin et al. [16] presented approaches for deforming geometry based on the observer's position. Martin et al. [16] used observer-dependent control functions to indirectly control the transformations applied to the model. In contrast, Rademacher et al. [19] blended a set of pre-defined object-deformations corresponding to a set of view-points closest to the current view-point. Both these methods provide a convenient way of deforming the model based on the view. All of the deformations presented in this paper similar in that they are view-based but are different in that they leave the underlying geometry intact.

Fish-eye lenses have been found to be extremely useful for visualization of information graphs and other applications. Magnification lenses [24, 14] are one method of visualizing expanded views of volume-data. Similar to these papers, we introduce a transformation of the view-space for exploring data in this manner. However, in our work lenses are only a subset of the entire set of possible view-transformations available in our framework.

In this paper, we present a set of widgets that make controlling of several cameras easy. A perspective camera can also be controlled indirectly by introducing image-based constraints into the scene ([4, 8]). A general-purpose solver is used to solve for the change in the camera parameters that would satisfy the given image-space constraints. Our tool-kit is structured in a similar way in that given a set of 3D points (source volume) and the corresponding 2D points (destination area), we compute a set of cameras that project the 3D points to the 2D points. In contrast to previous approaches, for each widget we allow only a certain set of camera parameters to change and compute the cameras analytically, without resorting to a solver. Also, the goals of the two systems are different. Constraining methods are more general-purpose and are well suited to camera placement or animation where the primary goal is to obtain a smooth camera motion that results in the given change in screen-space. We believe that such a general approach to data visualization is unnecessary and in some cases not useful. In this paper, we describe a set of widgets, each of which produces a specific type of transformation useful for viewing data in a particular way. The kind of transformation a widget can perform is clearly defined to the user before-hand and modifying a widget is directly correlated to the way in which the final view changes.



(a) Hand Bones (b) Female pelvis

Fig. 2. (a) This model consists of several bones which lie at different depths and that overlap each other. A single slicing plane would not expose any single bone. The clipping widget can be attached to individual (or several) bones to reveal the underlying details. (b) This model of the human female pelvis has several features at different locations (for example, the left and right acetabulum lying on opposite sides of the pelvis) which cannot be seen in a single view. We apply our tool-kit on this model for examining these different features.

Multi-projection techniques ([2, 9]) allow each object to be rendered from a different view-point. Neither of these approaches allows for continuously varying projections over a single object. The idea of creating a non-linear perspective projection from multiple linear perspectives was first introduced by Singh [21]. While Singh's work did not have methods to specify global scene coherence, Coleman et al. [6] employed constraints to control the overall composition of the scene. Both approaches required users to specify individual camera parameters of different cameras in the scene which makes the overall interface extremely cumbersome. Instead, each widget in our interface encapsulates a pre-defined type of perspective transformation (for example the fish-eye widget creates a single fish-eye zoom over a user-selected region). Also, the general-purpose interfaces of Singh and Coleman et al. tends to be unnecessary for exploring scientific data-sets since only a handful of camera transformations are usually performed (Section 5.5).

A different approach to the manual camera specification used in the above systems is the system introduced by Coleman et al. [7]. Their system defines a simple set of primitives, such as points, lines, and bounding boxes, that are used to express image-space constraints. A simplex solver finds the camera that satisfies these constraints. Several such primitives are combined to generate a set of cameras that form a non-linear projection. Our framework follows a similar approach for generating a non-linear projection by combining several linear perspective cameras. Also, both approaches use fall-off functions in order to combine multiple cameras. However, our system differs from the system in [7] in terms of the final goal. Their system was primarily intended for simulating the non-linear perspective effects seen in art. Thus, their interface was based on simple art-based primitives. The primitives were very general and changing the primitive even slightly could produce a significantly different projection. Also, it was sometimes hard to know which primitives to use in order to get a specific non-linear projection. This makes their system challenging to use for exploring data-sets. In our approach, each widget produces a pre-defined, simple non-linear projection. In contrast, the approach in [7] requires the composition of several primitives in order to produce even a simple non-linear projection. Also, in addition to the 3D primitive the user has to specify also specify a 2D component. We simplify our interface by making certain assumptions about the

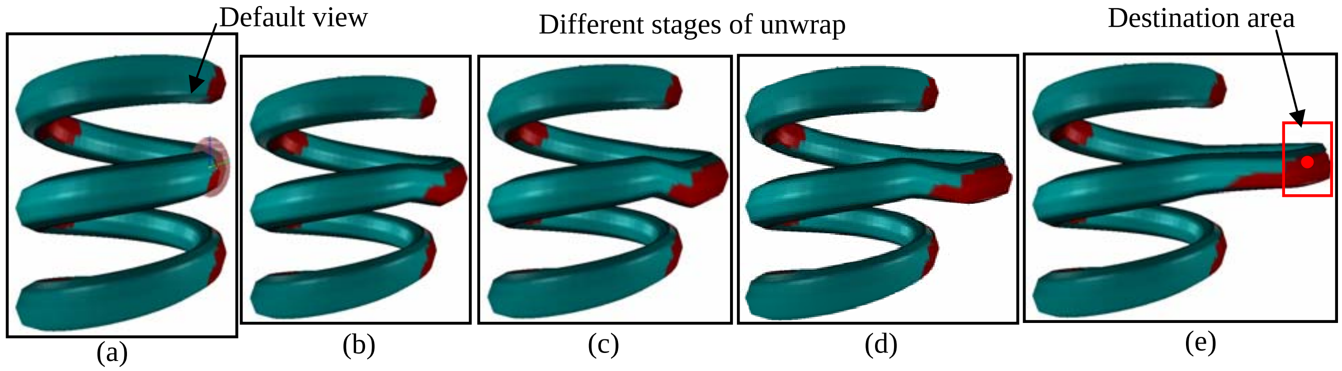


Fig. 3. (a) The default view with the unwrap widget. The source volume is located on the right-side of the helix and is same as Figure 7. Note that this volume is only slightly visible in the default view. (b)-(d) show the different intermediate stages of unwrap until the source volume is completely seen in the default view. The red bounding box shows the destination area, which is automatically calculated by the system.

position of the 2D component. Manually specifying a 2D component is important when the user is concerned with composition of the final non-linear perspective scene. This is not as important when the user is only concerned with exposing key features. Finally, the simplex solver used in [7] occasionly gets stuck in local minima. While it is useful to express constraints for a single camera through these primitives, the non-linear rendering interface is not interactive. Our framework does not make use of a solver and our GPU implementation of the rendering pass makes the interface real-time. Thus, the only common feature between two methods is the global approach for generating non-linear projections.

The work presented by Yagel et al. [13] is an alternative approach to deforming a model. Their approach consisted of deformation proxies or rendering agents which were inspired by traditional object-deformation paradigms. These rendering agents were view-independent. That is, irrespective of the view, the object appeared to be deformed in the same way as it was initially defined. Thus, using a rendering agent to deform part of an object does not guarantee that that part will be seen in all views. It only guarantees that it appears to be deformed (i.e the model itself remains intact) in a particular way. Our goal is to present an alternative approach to viewing and exploring an object which allows users to view the model as they like while simultaneously guaranteeing that interesting features are always seen.

Work has been done in unfolding specific structures such as a human colon [3] and human blood vessels [12]. Among the various differences between the unwrap widget and these approaches, one of them is applicability. The unwrap widget can be used on any kind of structure, and the interface (as well as the technique) is not restricted by the topology of the object. Thus, unwrapping or flattening a tubular structure is one of the transformations that can be done by our system by adding a set of chained unwrap widgets along a user-defined path or a path that is determined based on the specific structure at hand. Furthermore, this can be done in real-time.

Work has been done in building multiperspective renderings using ray tracing ([26, 27, 15]). Hou et al. [11] present a real-time multi-perspective framework that involves discretizing a multi-perspective surface followed by a two-step projection process on the GPU. Yu et al. [27] presents a novel camera model that includes the pinhole camera along with other classes of cameras. The authors then use this formulation to build multiperspective renderings as seen in art ([26]). The underlying framework used in our work is a subset of General Linear Cameras since all the cameras used to build a non-linear perspective rendering are pin-hole cameras. However, the goals of Yu et al. [26], [15] and [11] are different from ours. Our goal is not to generate a single non-linear perspective image, but instead to generate a continous set of composite images each of which satisfy a set of constraints (in particular, certain specific features are always seen) while a user explores the scene. Also, keeping in mind the domain we are interested in, using only a particular class of cameras (pinhole cameras) is natural since pinhole cameras are commonly used for exploring 3D environments. Pinhole cameras also have the added advantage of being fast to compute.

Takahashi et al. [23] presents a system that renders different features in a geographical map such that features can be seen within a single global viewpoint. The goals of this system and ours are extremely close. However, unlike our view-based approach, their approach actually deforms the geometry based on constraints that ensure that interesting features are seen within that view-point. Also, the constraints must be redefined every time the view-point changes. Our approach allows users to change the view without having to change the constraints.

Popescu et al. [18] present a camera model that looks around occluding objects to capture details of the occluded objects. This was used in the context of image-based rendering where, given a reference image, a depth image and color information, a scene can be rendered from novel views. Their camera model augmented the existing depth image with additional depth samples at places where depth discontinuities occur when the view is changed. Since their domain was image-based rendering, their non-pinhole camera model was used primarily for storing samples of nearby occluded objects and held no information regarding objects further away from the current view.

Work has been done in producing multi-perspective panoramas given either a 3D model and a camera path [25] or a set of images [22]. A panorama is a special kind of non-linear projection. Thus we adapt our framework to generate panoramas making the process completely interactive. Extensive work has been done in creating panoramic views from a set of input images without recovering the scene geometry ([28, 20, 1]). In contrast, our approach tackles a different problem of generating panoramas when the 3D geometry is known.

5 The non-linear projection tool-kit

Our tool-kit is composed of three widgets, namely, the unwrap widget, the clipping widget, the fish-eye widget, and the panorama widget. We believe that these widgets are general enough to be used to build useful non-linear projections.

5.1 Unwrap Widget

Exploring a data-set is commonly done through a series of camera rotations and translations. However, in certain data-sets a small camera rotation can result in a significant portion of the data going out of the field-of-view. The unwrap widget applies a view transformation to bring such data back into the current view. The data remains in the field-of-view irrespective of how the current view is rotated. An example of such data-sets are helices where two surfaces are interleaved. In Figure 1, it would be impossible to see both section A and section B of the helix in the same view since looking at either part would involve a rotation of the camera that would result in the other moving out of the view. The unwrap widget allows user-selected regions to

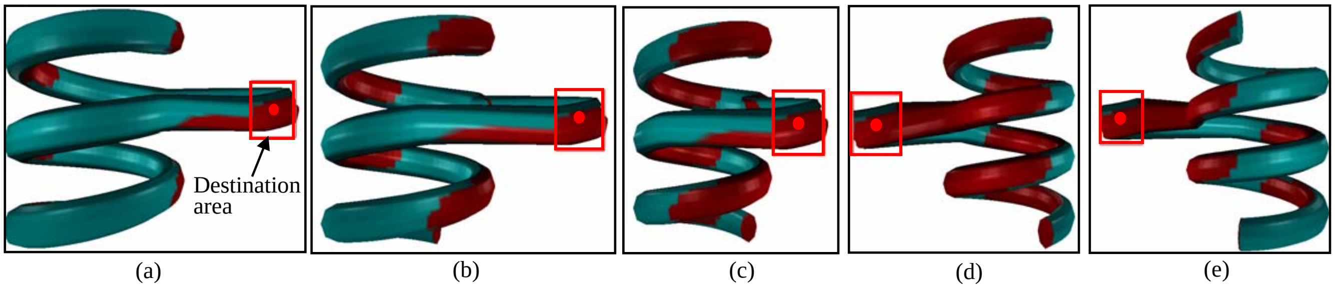


Fig. 4. Figure shows a sequence of unwrappings obtained by rotating the default camera around the helix. The source volume is same as Figure 3 and is always seen. In Figures (a)-(c) the camera approaches the source volume as a result of which the length of the unwrap region reduces and the size of the destination area increases. In Figures (d),(e) the camera has gone past the source volume. Note in all these examples, the destination area is automatically calculated as the default camera changes.

be rotated into the current field-of-view while ensuring that remaining parts of the model are seen as before.

5.2 Clipping Widget

The goal of the clipping widget is to expose structures that are occluded in a particular view. Usually, parts of the model that are farther from the current view-point are occluded by those that are closer (Figure 2(a)). The clipping widget allows regions of the model located at different depths along a particular view-direction to be exposed. Changing the global view results in exposing structures lying at those depths but along a different view-direction.

5.3 Fish-eye Widget

The fish-eye widget simulates a fish-eye lens. When a scene is viewed through a fish-eye lens, regions within or close to the center of fish-eye lens appear more magnified than regions that lie outside the lens region (Figure 12).

5.4 Panoramas

Panoramas are a concatenation of several views to create a composite view. We adapt our framework to combining different views of a 3D model to generate a single panorama (Figure 17).

5.5 Design Rationale

Our widgets are inspired by commonly-performed camera operations and object-space transformations. Frequently used camera operations include camera panning, camera-dollying, camera rotation and camera zoom. Camera rotation is encapsulated within the unwrap widget and dollying is encaspulated within the fish-eye widget. The clipping widget is derived from the common object-space transformation of using slicing planes to view the internal structure of a model. Each of the widgets affect different camera parameters independently, thus combining the widgets is quite natural. Combining these widgets aggregates the functionality of these operations and simultaneously applies these deformations to the model.

6 The Unwrap Widget

6.1 Overview

As described in the previous section, the unwrap widget is used to rotate selected regions which may not be seen into the current field-of-view of the camera. Only the vertices located within the region are affected, the rest of the model is seen as before.

6.2 User's View

The user specifies a source volume which corresponds to the part of the model that they are interested in viewing from a different direction. The source volume is selected using a 3D widget which contains handles for controlling the 3D position and 3D scale of the selected volume. We use the surface normal at the center of widget to fix the orientation of the widget (Figure 7). If we directly use the local camera, the region will be projected to the center of the screen and would therefore be occluded by the rest of the model which is rendered using the default view. Thus we need to determine a good screen-space position for projecting the volume. The position of this area corresponds to the position on the screen where the volume will be projected while the scale corresponds to the final projected size of the volume. We could have the user specify the 2D position and scale of the destination area, but for most data-sets we can automatically determine a good area.

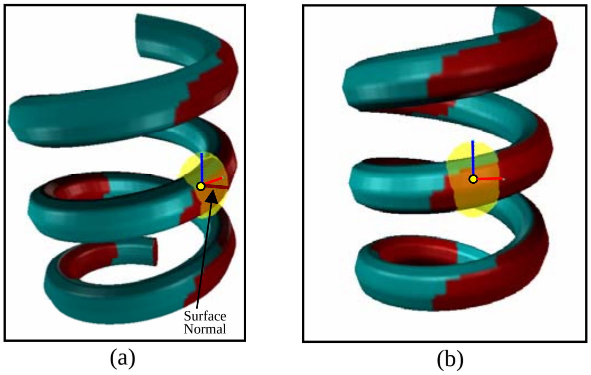
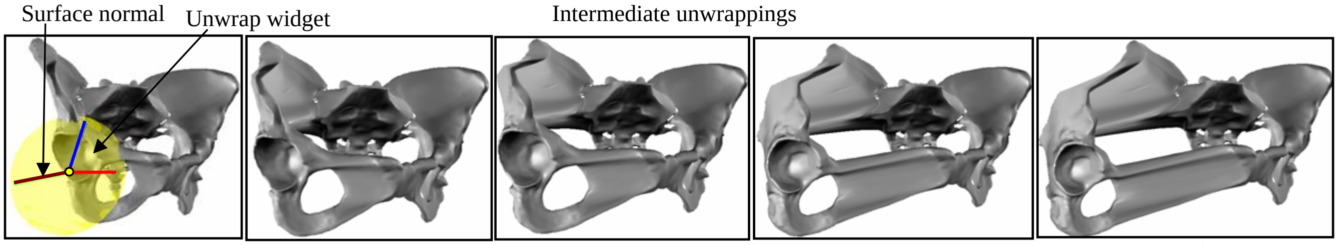


Fig. 7. (a) The helix with the unwrap widget. The unwrap widget controls the position, orientation and scale of the source volume. (b) The local camera encapsulated by the unwrap widget. The camera looks at the center of the widget along the normal at that point (which is denoted by the brown axis in this case).

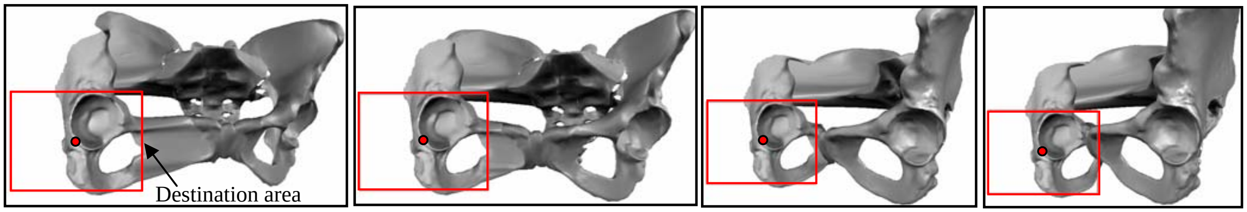
Automatic calculation of the destination area: To simplify our interface, we make a few assumptions about the destination area. First, we project a given source volume using the corresponding local camera to compute the 2D bounding box, p_v. We also compute the 2D bounding box of the entire model by projecting its 3D bounding box using the default view, p_m. Ideally, we would like p_v to lie completely outside of p_m i.e no part of the model seen in the default view occludes the source volume seen in the local camera. Also, we would like p_v to be entirely visible on the screen i.e no part of p_v lies outside the screen. Thus, we alternately pan and zoom the default camera until p_v is completely seen (unoccluded) on the screen.

Representing a source volume: Internally, the source volume is represented by an implicit function which represents the position, orientation and scale of the volume. Additionally, a fall-off function is associated with the source volume. The fall-off function is used to blend between the points projected using the local camera and the default view. Section 9 explains the method used for blending views in more detail.



(a) Default View (b) (c) (d) (e) Final View

Fig. 5. The unwrap widget is added to surround the right acetabular cavity. In the default view, only part of the cavity can be seen. The intermediate steps between the default view to the final view is show. In the final view, the whole cavity can be seen.



(a) (b) (c) (d)

Fig. 6. The unwrap widget is added to the right acetabular cavity as before. The camera is then rotated around the pelvis. In each frame, the right cavity is visible until both the cavities are seen side-by-side. The red box indicates the destination area of the right cavity.

6.3 Methodology

Each vertex [1] in the model is assigned a camera. The parameters of the camera are computed based on the location of the vertex. Vertices lying within the source volume are assigned a rotated camera C_{rot} looking at the center of the widget whose view-direction is given by the surface normal at that point. In other words, for the unwrap widget the local camera is a camera that looks at the surface volume from the intended direction. The screen-space position of the rotated region is calculated as explained in the previous section.

$$COP^{'} = (-c_x, -c_y) \tag{1}$$

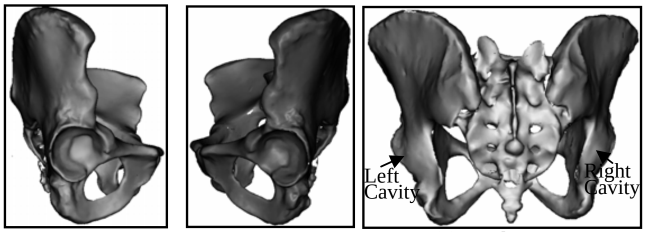
$COP^{'}$ is the center-of-projection of local camera and $(-c_x, -c_y)$ is the calculated center of the destination area. Finally, all vertices lying outside of the source volume are assigned the default camera. Thus, the rest of the model is guaranteed to be projected as before while only the source volume is assigned a different local perspective.

6.4 Examples

Unwrapping a helix: A real-world example where helical structures are manipulated is in the case of protein data which is commonly found to lie on helical structures. Figure 3 shows the final unwrapping starting from the initial default view of the helix (number of faces = 27,689, number of vertices = 13,248) The intermediate stages of unwrap are used to give a sense of how the geometry appears to be deformed from the default view to the final unwrapped view.

One of the key advantages of applying a view-based transformation to the model is that the unwrapping is invariant to changes in the global view. This is illustrated in Figure 4 where the camera is rotated around the helix. As the default camera changes the source volume is always seen and the corresponding destination area is recalculated based on the projected size of the source volume and the available screen-space.

Exploring a human pelvis: Figure 8 shows different views of the pelvis (Number of faces = 1,289,814, number of vertices = 49,989). In this example, our goal is to compare the left and right acetabular cavities which lie on opposite sides of the pelvis. The acetabular cavities play an important role in surgeries relating to hip and joint replacements. Finding a single view where both these cavities can be seen simultaneously is challenging. We start by adding an unwrap widget to the right cavity adjusting its scale to match the scale of the cavity. Thus, a local camera looking at the cavity is created. The default view, along with intermediate unwrappings obtained, are shown in Figure 5.



(a) Right-side view (b) Left-side view (c) Back view

Fig. 8. We apply the unwrap widget for comparing between the left and right acetabular cavities that lie on opposite sides of the human pelvis.

By simply rotating the default camera, both cavities are presented side-by-side for comparison. In each intermediate step of rotation, the right cavity is always seen. The entire sequence is illustrated in Figure 6. The destination area shown in red is automatically calculated.

7 Clipping Widget

The clipping widget is used to view occluded structures in a model.

7.1 User's View

As before the user specifies a default view-point which determines the overall projection of the model. In addition, the user specifies one or more sections of the model that need to be exposed. These sections can be thought of as a set of slices taken along a particular viewing direction. Each section corresponds to a single source volume and is represented by a bounding box (Figure 10(b)).

Automatic calculation of destination area: The selected sections are projected to lie outside the projected bounding box of the model in the default view. The sections are projected either to the left or right

[1] For volume rendering we use the data grid vertices and the gradient.

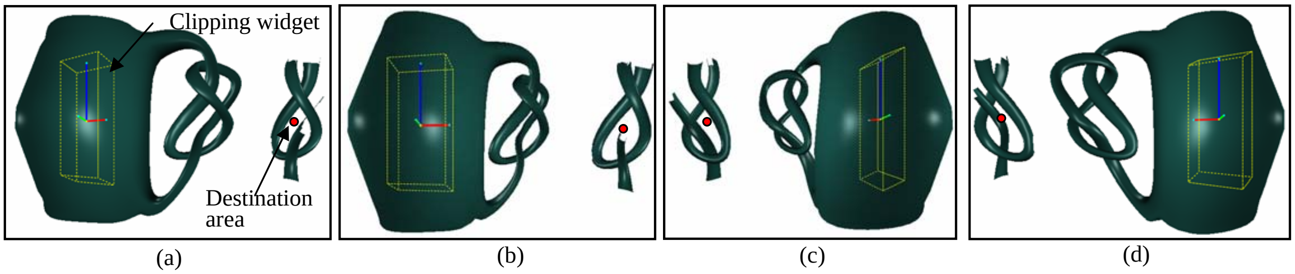


Fig. 9. This mug has a knot on the inside as well as a knot in the handle. As the default camera rotates different views of internal knot are seen and can be compared with the external knot. The destination position is recalculated for each frame.

of the default view depending on the side that has the most space. The default camera is panned and zoomed out if necessary to accomodate the sections as before. Each section is placed vertically one above the other. Finally, the camera corresponding to each section is zoomed out if necessary to fit the slices (Figure 19).

7.2 Methodology

Vertices that lie within a source volume are assigned a local camera that is the same as the default view but differs in its center-of-projection and zoom angle. The center-of-projection corresponds to the center of the region where the volume is to be projected. Projecting a source volume with this camera results in it being seen on an unobstructed portion of the screen. Vertices that lie outside a given source volume are assigned the default camera as before.

One important difference between the clipping widget and the remaining widgets presented here is that in the case of the clipping widget no blending is performed. This is because a given vertex should be associated with only one camera : the default camera or the local camera. Thus, only a cut-out of the relevant section is rendered.

7.3 Examples

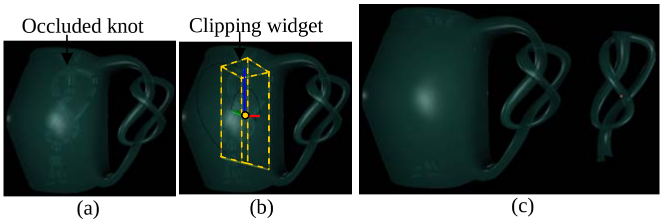


Fig. 10. (a) The mug encloses an internal knot that is completely occluded. The front-view of the mug acts as our default view. (b) We attach the clipping widget to the knot. (c) Both the front-view and the knot are seen simultaneously.

Knotted Mug: The first example illustrates the use of the clipping on the knotted mug (Number of faces = 10,768 , Number of vertices = 5382). Within this mug, lies an internal knot that is completely occluded in all views. Our goal is to compare the outer and internal knot. If a clipping plane was passed through this mug, the internal knot would be revealed at the expense of the outer knot. When the clipping widget is attached to the inner knot, both the outer and inner knot can be seen side-by-side (Figure 10). The destination position for the clipped volume is automatically computed as described in Section 7.1. As explained in Section 6.2 alternating between panning and zooming out the camera is important to ensure that the source volume is both unoccluded and entirely visible in the final view. We illustrate one such scenario in Figure 11. Initially, the destination position is found to lie outside the screen and thus the clipped volume is only partially visible. Using our method of alternately panning and zooming out the camera both the clipped volume and the model can be seen completely. Even though, the model appears smaller, the clipped volume appears at the original scale since the zoom parameter of the local camera is unaffected.

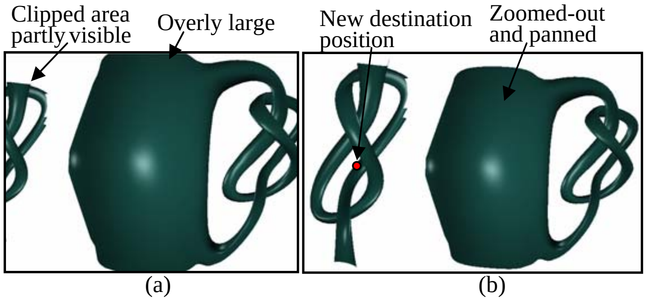


Fig. 11. (a) Initially, the destination position is found to lie outside the screen. Note that the model occupies most of the screen-space in the default view. (b) We alternate between panning and zooming the default camera until the source volume is entirely visible. Even though the model itself appears zoomed out the source volume is seen at the original scale.

When the user rotates the default camera, they can simultaneously view and compare the outer and inner knot from different angles (Figure 9). Thus, using the clipping widget any feature lying within a model can be compared with exterior features.

8 Fish-Eye Widget

8.1 Overview

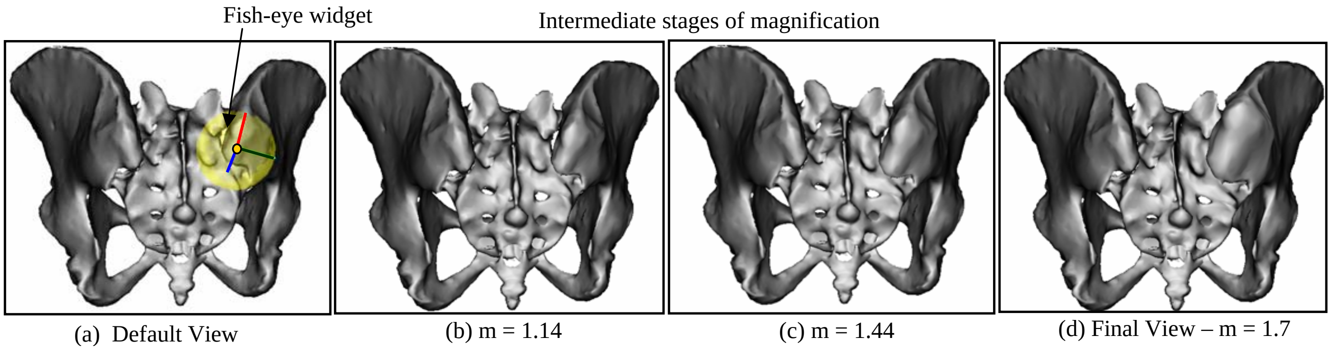
Fish-eye lenses are a well-known data exploration tool. In this section, we show that our framework is general enough to produce such visualizations.

8.2 User's view

A fish-eye widget is represented by two concentric spheres; the internal sphere corresponds to the region where the magnification is maximum. The region between the inner and outer sphere corresponds to the region over which the magnification gradually reduces. Moving the widget around the model changes the region over which the fish-eye is applied. Each fish-eye is also associated with a magnification factor, m. Finally, all vertices that lie outside the outer sphere are projected using the default view.

8.3 Methodology

Similar to the methods outlined in the previous section each vertex on the model is assigned a camera. Vertices that lie within the inner sphere of the fish-eye are assigned a local camera that is panned-in and whose center-of-projection is adjusted appropriately (Equation 2a-2c). Essentially, these equations build a camera that has been translated down the view-direction of the default view by reducing the focus distance of the camera. In addition, the center-of-projection of the camera is adjusted so that the 3D point corresponding to the center of



(a) Default View (b) m = 1.14 (c) m = 1.44 (d) Final View – m = 1.7

Fig. 12. (a) The fish-eye widget is represented as a sphere. Note the increasing size of the joint in successive frames of the magnification.

the fish-eye remains at the center even in the new camera. Vertices that lie in the region between the inner and outer regions are assigned a camera that is a weighted combination of the fish-eye camera and the default camera. Section 9 explains the method used for blending views in more detail.

$$panAmount = f_d \times (1.0 - 1.0/m) \tag{2a}$$

$$Eye^{'} = Eye + \vec{Look} \times panAmount \tag{2b}$$

$$COP^{'} = COP + (p - c) \tag{2c}$$

$panAmount$	:	Amount the camera is panned in
f_d	:	Focus distance of default camera
$Eye^{'}$	:	Eye point of fish-eye
m	:	Magnification factor
$COP^{'}$	:	Center-of-projection of fish-eye
$Eye, Look, COP$	:	Camera parameters of the default camera
c	:	Center of fish-eye widget
p	:	Projection of 3D point P corresponding to c

8.4 Examples

We illustrate the use of the fish-eye widget on the human pelvis. We place the fish-eye widget on the sacroiliac joint to magnify it. The intermediate steps of the magnification are shown (Figure 12).

9 Blending views

As described in Section 2, each widget is associated with a fall-off function. We use fall-off functions to blend between the local camera and the default camera, in order to ensure that a continous non-linear perspective projection is generated. The fall-off function is:

$$w(Q) = \begin{cases} 1 & ||Q - C|| < r_{in} \\ g(\frac{||Q-C|| - r_{in}}{r_{out} - r_{in}}) & r_{in} \leq ||Q - C|| \leq r_{out} \\ 0 & ||Q - C|| > r_{out} \end{cases} \tag{3}$$

Q	:	3D vertex position
w	:	Weight of Q
C	:	Center of the source volume
$g(x)$	=	$(x^2 - 1)^2, x \in [0, 1]$
r_{in}	:	Inside radius of the source volume
r_{out}	:	Outer radius of the source volume

Equation 3 defines a fall-off function based on 2 parameters, namely r_{in} and r_{out}. r_{in} is calculated based on the scale of the source volume. r_{out} controls size of the transition region over which blending will takes place and is user-controlled. Increasing r_{out} increases the size of the fall-off function, which in turn improves quality of transition between the different views of data-set.

9.1 Blending projections of points

Each vertex is associated with a weight which is calculated based on Equation 3. Then, the final projection of the vertex is computed according Equation 4. Each vertex is simply a weighted combination of the projections using the local camera and the default camera. The weights of the different cameras are normalized in case their sum exceeds 1.

$$q = (1 - \sum_{k=1}^{n} w_k) D(Q) + \sum_{k=1}^{n} w_k C_k(Q) \tag{4}$$

n	:	Total number of local cameras
Q	:	3D vertex position
D	:	Default camera view
C_k	:	k^{th} local camera
w_k	:	Degree of influence of k^{th} local camera on vertex Q

Another possible method of blending is to blend the individual camera parameters of the two cameras and project the vertex using the final blended camera. While this method produces the “correct” result, it is time-consuming. We found that Equation 4 is a good enough approximation which works well in practice.

10 Multiple widgets

One of the primary goals of our work is to create a tool-kit which is flexible enough to encapsulate a large number of useful non-linear projections. We accomplish this by creating compositions of the different widgets. There are three types of relationship that can exist between multiple widgets placed in a data-set. The type of relationship that exists becomes important when we are computing the destination area for each widget since the approach followed depends on the relationship.

10.1 Independent widgets

In this case, multiple widgets are place in a scene and are treated independently of each other. Each widget has its own region of influence which may or may not overlap with another. We compute the destination area independently for each widget as per the approach outlined in the previous sections.

10.2 Parent-child widgets

In this case, widgets are usually embedded within one another. More specifically, both the inner and outer radii which define the region of influence for the child widget must be less than the inside radius of the parent widget. For example, a fish-eye widget could be embedded within an unwrap widget for magnifying part of an unwrapped region. In this example, the parent widget is the unwrap widget while the child widget is the fish-eye widget. When the user selects such a relationship, the system automatically assigns the default camera of

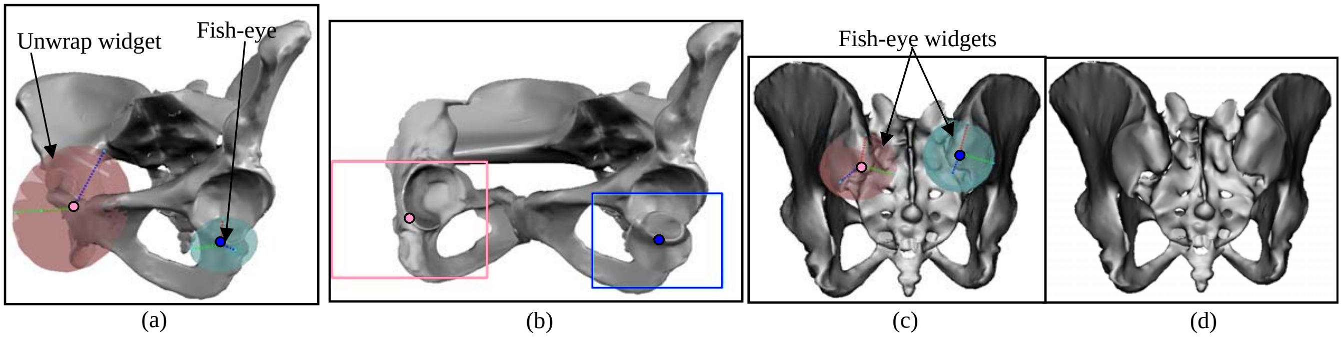


Fig. 13. (a),(b) - In addition to the unwrap widget, we add a fish-eye widget to the left cavity to aid the comparison of the two cavities of the pelvis. (c),(d) - We add two fish-eye widgets to each joint in the pelvis.

the child widget to the local camera encapsulated by the parent widget. Thus, the fish-eye widget's default camera is the rotated camera of the unwrap widget. The calculation of the destination area for the child widget is then done with respect this new camera.

10.3 Chained widgets

In this case, multiple widgets (usually of the same type) are strung together. Chaining widgets is useful in order to specify a complicated piece of geometry or to prevent a fold-over of geometry. When multiple widgets are chained together, then the calculation of the destination area follows the standard approach only for the first widget in the sequence. The center-of-projection (or zoom) is then propagated down the chain. Thus, the after-projection parameters of successive widgets are computed with respect to earlier widgets in the chain. This is done so that widgets appear sequentially on the screen.

Foldover of geometry: Foldover or self-intersection of geometry occurs when the difference between the default camera and the local camera is large. In such a case, different faces in the transition region overlap one another resulting in self-intersections in the transition region. No amount of blending can prevent this if the two views are really disparate. The solution is for the user to place additional smaller widgets that individually result in fold-over free transformations. These smaller widgets exhibit a chain dependence on each other.

10.4 Examples

Bones data-set: The bones data-set (Number of faces = 158,080, number of vertices = 316,100) is an especially challenging data-set to view since there are several overlapping bones. Finding a single view that reveals multiple bones is hard. We would like to use our tool-kit for solving the following challenging visualization problem : isolating and viewing the contact points between two bones. We apply the clipping widget for isolating the two bones. We then introduce a fish-eye widget as a child widget to visualize the actual contact point. We can expose multiple bones by adding multiple independent clipping widgets. These exposed bones can be simultaneously compared with the remaining bones as seen in the default view (Figure 19).

Unwrapping a helix: Chained widgets are useful for selecting volumes that are large and vary in shape. We illustrate the use of chained widgets on the helix. Each widget is an instance of the unwrap widget, which when chained together unwrap a single coil of the helix (Figure 20).

Human Pelvis: We illustrate the application of independent widgets on the human pelvis. In addition to the unwrap widget we added in Figure 6, we add a fish-eye widget to the left acetabular cavity to aid comparison of the two cavities (Figure 13(a),(b)).

We demonstrate another instance of multiple independent widgets on the same data-set in Figure 13(c),(d). In this case, we add multiple independent fish-eye widgets to compare the two joints lying on either side of the sacral promontory.

11 Panoramic Views

In this section, we illustrate the flexibility of our framework by applying it to the problem of generating panoramic views. Similar to the widgets presented in this paper, panoramic views are used to view a single data-set through multiple sub-views. Panoramas form a subset of non-linear projections since they can be formulated through a concatenation of several views. Thus, we can easily generate panoramic views using the existing framework.

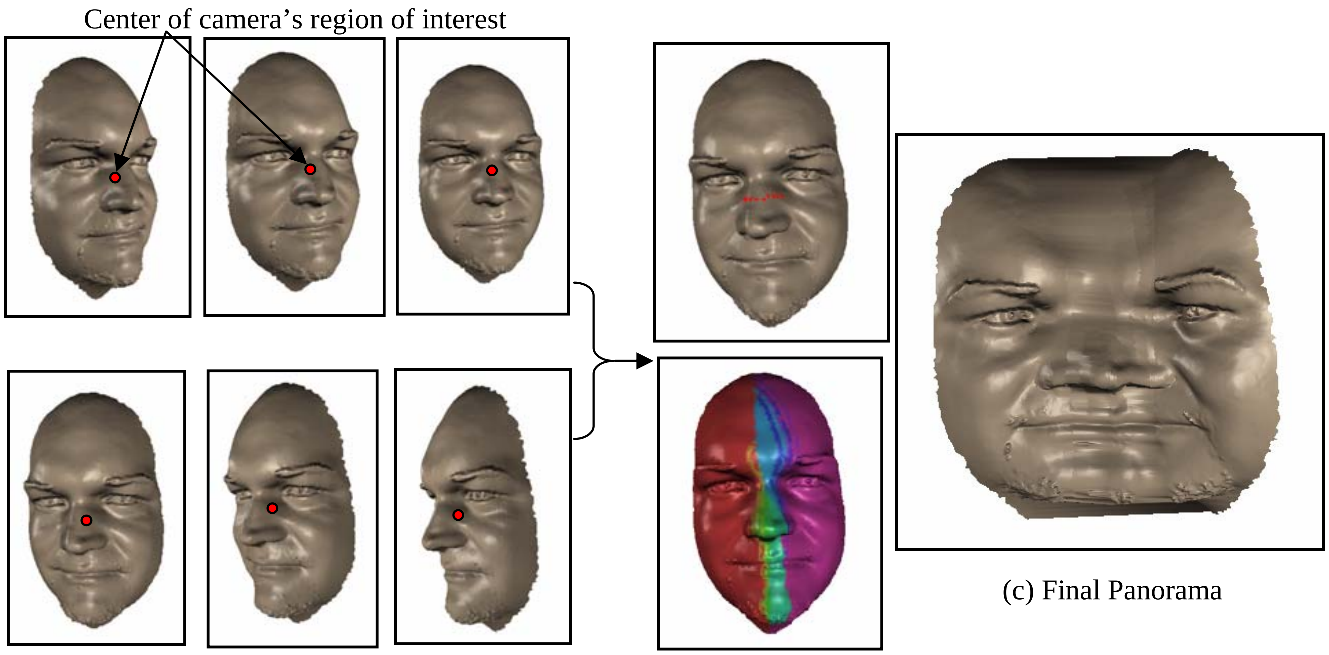
11.1 Approach

The user provides either a set of keyframes or a camera path to the system (Figure 14(a)). Given a set of keyframes, our goal is to combine these views to create a single panoramic image. When creating panoramic views, the user does not explicity specify a source volume for each camera. The system automatically calculates a source volume for each of the input keyframes (Section 11.2). Adjacent source volumes are then blended together to ensure a continous panoramic view (Section 11.3). Finally, the system automatically calculates initial screen-space positions for different parts of the panoramic view (Section 11.4). The screen-space positions can be later modified by the user.

11.2 Computing a source volume

Each camera is usually centered on a specific region of the model. We refer to this region as the region of interest (ROI) for a camera. Even though this is same as the source volume for a camera as defined in the previous sections, we will refer to this region as a camera's ROI in this section. We need to determine the center of a camera's ROI and the vertices that lie within it. To determine the center, a ray is cast from the camera's eye along the its view direction. The first point of intersection corresponds to the center of its ROI (Figure 14(a)). Applying this approach to all the input cameras yields a set of centers on the model (Figure 14(b)). For every vertex on the model the system find the closest center. This results in a set of clusters corresponding to the different ROIs on the model. Finally, each vertex is projected with the camera corresponding to the closest center, $i_{closestcam}$ to give the panoramic view as seen in Figure 14(c).

Using the above method, it is possible for vertices that are seen in one view to be assigned to another view where they are occluded (Figure 15). This occurs since our metric for clustering vertices does not take into account the visibility of a vertex but only takes into consideration its proximity to the center of a ROI. One possible method to account for the visibility of a vertex is to compute the dot product between the normal at that vertex and the look direction of a given camera. A vertex is assiged to the camera that yields the minimum dot product. However, we found that solely relying on the dot product to cluster vertices produced several isolated clusters since such methods are extremely sensitive to small changes in the geometry of the model. To offset this problem, we incorporate an occlusion procedure. When occlusion checking is turned on, in addition to finding $i_{closestcam}$, the system also finds the camera that is looking most directly at a given



(a) Input Keyframes

(b) Camera Clusters

(c) Final Panorama

Fig. 14. (a) Illustrates a few of keyframes (total number of keyframes = 9) obtained used to generate a panorama of the face model. The center of each camera's ROI is found using the method outlined in Section 11.2. (b) Each vertex is associated with the center closest to it. This results in a set of clusters as shown in (b). Each color corresponds to the region of interest of a particular camera. (c) The final panorama. Notice that both sides of the face can be seen completely along with the features lying in between.

vertex, $i_{bestdotcam}$. This is done by finding the dot product between a given camera's view direction and the vertex normal and picking the camera which yields the smallest dot product. Then, $i_{bestdotcam}$ is used only if the vertex when projected with $i_{closestcam}$ lies at a depth greater than a threshold, $depth_{iclosestcam}$.

Determining the depth threshold: The depth threshold is the maximum depth upto which the i^{th} camera can see. It depends on a given camera's position and look direction. To find $depth_i$, we project the center of the model using $camera_i$. The z component of the projected center is then assigned to be $depth_i$. After determining the depth thresholds for all cameras, we can apply the occlusion check procedure to each vertex as explained previously.

11.3 Blending

Since a panorama is a special kind of a non-linear projection, the blending methodology used to combine different source volumes as specified in the previous sections can be specialized to be applied in the case of panoramas. As explained before, a panorama is made of multiple camera views. Thus any given view must be combined with the adjacent views to yield a single continous composite view. This is the underlying assumption of the technique used to create the panorama.

Given a vertex V lying within the ROI of $camera_i$ (R_i), the system first identifies the two closest ROIs R_{i-1} and R_{i+1}. V is projected with $camera_{i-1}$ and blended if it lies between the center of R_{i-1} and the center of R_i. Such a vertex would yield a positive value for the expression given in Equation 5a. This procedure is also used for finding those vertices that are to be blended with R_{i+1}. Equation 5b is then used to find a blend weight for each vertex. Equation 5b is same as the falloff function used for blending between source volumes in the previous sections (Equation 3). However, we can make assumptions about inside and outside radius of the fall-off function. The outside radius can be at most the distance between the centers of the ROIs that are being blended while the inside radius can be adjusted by the user. Experimentally, we have found that assigning the outside radius to be half of the distance between the centers and the inside radius to be half of the outside radius works well. Finally, the blend weight is used to combine the projections of V to give the projected vertex, v (Equation 5c). Note that at the boundary, the first and last cameras are combined with only the second and second to last cameras respectively.

$$LV = dot(V - O_i, O_{i-1} - O_i) \tag{5a}$$

$$w(V) = \begin{cases} 1 & LV < r_{in} \\ g\left(\frac{LV - r_{in}}{r_{out} - r_{in}}\right) & r_{in} \leq LV \leq r_{out} \\ 0 & LV > r_{out} \end{cases} \tag{5b}$$

$$v = wC_i(V) + (1.0 - w)C_{i-1}(V) \tag{5c}$$

V	:	Vertex lying within R_i
O_i, O_{i-1}	:	Centers of R_i and R_{i-1}
LV	:	Length of projected V on the vector between O_i and O_{i-1}
$g(x)$	=	$(x^2 - 1)^2, x \in [0, 1]$
r_{out}	:	0.5×Distance between O_i and O_{i-1}
r_{in}	:	$0.5 \times r_{out}$
C_i, C_{i-1}	:	i^{th} and $(i-1)^{st}$ cameras

11.4 Calculation of the destination area

The initial destination position of each source volume can be computed by first calculating the average of all the keyframes. We refer to the camera corresponding to the averaged keyframe as the average camera. The average camera has each of its camera parameters

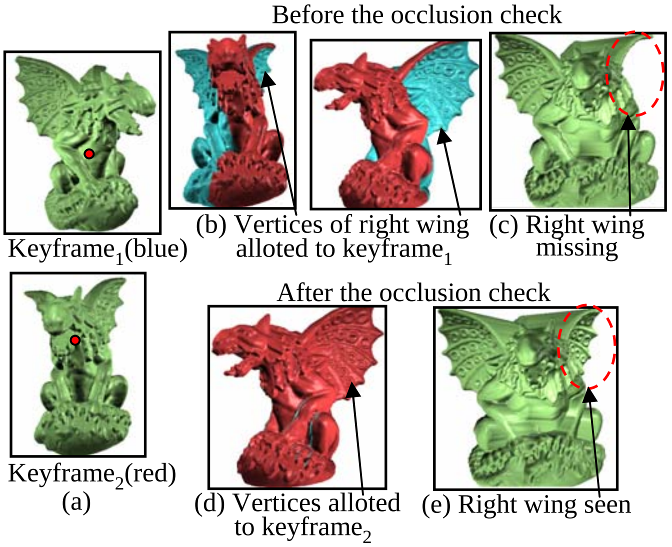

Before the occlusion check
Keyframe$_1$(blue)
(b) Vertices of right wing alloted to keyframe$_1$
(c) Right wing missing
After the occlusion check
Keyframe$_2$(red)
(a)
(d) Vertices alloted to keyframe$_2$
(e) Right wing seen

Fig. 15. (a) Two keyframes are given as input into the system (the center of each keyframes's ROI is also shown). Note that while the left wing of the gargoyle is clearly seen in $keyframe_1$ the right wing is completely occluded. The right wing is clearly seen in $keyframe_2$. (b) The right wing is closer to the center of $camera_1$'s ROI than $camera_2$'s ROI. Thus it is assigned to $camera_1$. (c) The resultant panorama. Although the right wing is seen in $keyframe_2$ the false assignment to $camera_1$ results in it being occluded in the final panorama. (d),(e) After applying the occlusion check, the right wing is assigned to $camera_2$ and seen in the final panorama.

equal to the average of the corresponding camera parameters of the input keyframes. The centers of the different ROIs are then projected with the average camera to yield the initial destination positions for the source volumes. Changing these positions changes the span of the panorama (Figure 16).

11.5 Examples

Cow model: The first example applies our method on the Cow model (number of vertices = 11606, number of faces = 23208) which uses three keyframes to yield the final panorama (Figure 16).

Gargoyle model: We applied our method to the a low resolution version of the gargoyle model (number of vertices = 129722, number of faces = 259440) using 11 keyframes to create the final panorama (Figure 17). The destination positions for this panorama are same as the initial positions calculated according to the method outlined in Section 11.4.

Face model: In general, our framework can combine significantly different views smoothly as long as the blending function is smooth and the underlying geometry is well-sampled. The face model (number of vertices = 7256, number of faces = 14271) illustrates this by using two significantly different keyframes to create a single panorama (Figure 18).

12 Implementation

All of the widgets presented in this paper share the same underlying idea. Every point in the data-set is projected with a unique camera. To ensure that the original lighting effects are retained even in the final rendering, lighting calculations have to be done on a per-vertex basis with respect to the original camera.

Implementing such a framework on the CPU results in slowing down both the rendering as well as the interaction phase. For example, we ran our framework on a laptop which had a 1.73Ghz Pentium M processor with a 1GB RAM on the human pelvis model (number of vertices = 1289814, number of faces = 49989). We introduced a single unwrap widget to this model at which point the system computed and stored the new projection of the vertices. This projection step took 115.95372054 seconds while the rendering time was 13.42311523 seconds. These times made interactive manipulation impossible.

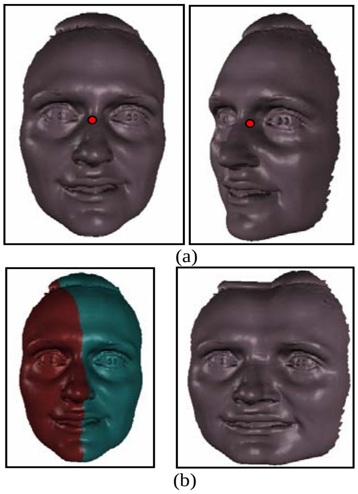


Fig. 18. (a) Input keyframes along with the center of each keyframes's ROI. (b) The camera clusters and the final panorama.

Thus, we implemented the framework on the GPU (a GeForce Go 6400 graphics card). A single vertex program performs the main task of blending and projecting the vertices. Uniform inputs to the vertex program are the falloff functions, the local cameras and the default camera matrices. The vertex program projects a given vertex with either a local camera, a default camera, or blends between the two depending on whether the vertex lies within any of the fall-off functions. In addition to the screen-space position, the output of the vertex program includes the surface normal and world-space position of the vertex transformed by the default camera. The entire data-set was stored in a display list in order to speed up the rendering phase. The GPU-accelerated program takes 0.775422016 seconds to render a non-linear projection of the human pelvis model with a single local camera. Finally, the rendering time is dependent on the number of widgets but independent of the type of widgets introduced in the scene.

13 Limitations

As the number of widgets added into the scene increase, it becomes more difficult to specify how they interact. With a large number of widgets, it also becomes hard to perceive the underlying structure of the data-set and the usefulness of the final non-linear projection is reduced. Thus we have found that up to four widgets on a large model (such as the human pelvis) works well.

14 Conclusion

We have presented a flexible, general-purpose tool-kit for building non-linear projections of data-sets. Multiple widgets can be combined easily in real-time to allow viewing of several features within a single view. We illustrate the flexibility of our framework by adapting it to create panoramic views.

15 Acknowledgments

The work presented here was funded in part by NSF grant 0238062. The authors would like to thank J.J.Crisco of Brown Medical School, Rhode Island Hospital for the bone data-set (captured under NIH AR44005), the AIM Shape repository for the human pelvis model and the gargoyle model and Cyberware for the face scans.

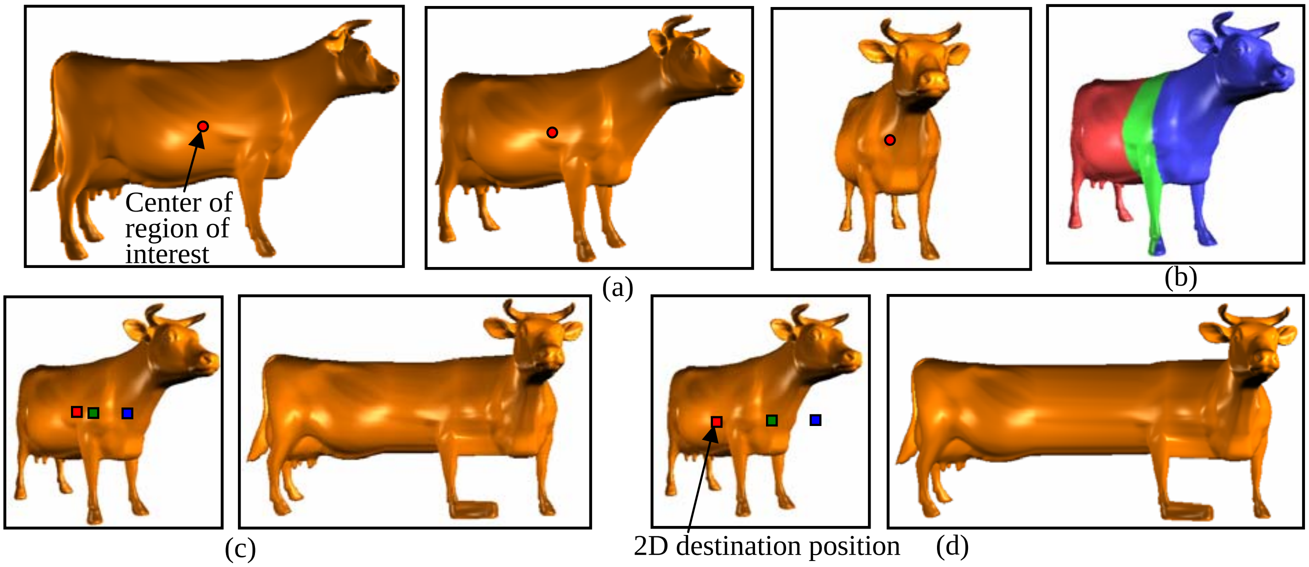


Fig. 16. (a) Shows the keyframes along with the centers of the corresponding ROIs used to generate the final panorama. (b) The camera clusters created as seen in the average camera. (c) The initial destination positions calculated automatically using the method outlined in Section 11.4 along with the corresponding panorama. (d) The modified destination positions and the corresponding panorama. Note that the span of the panorama has increased compared to (c).

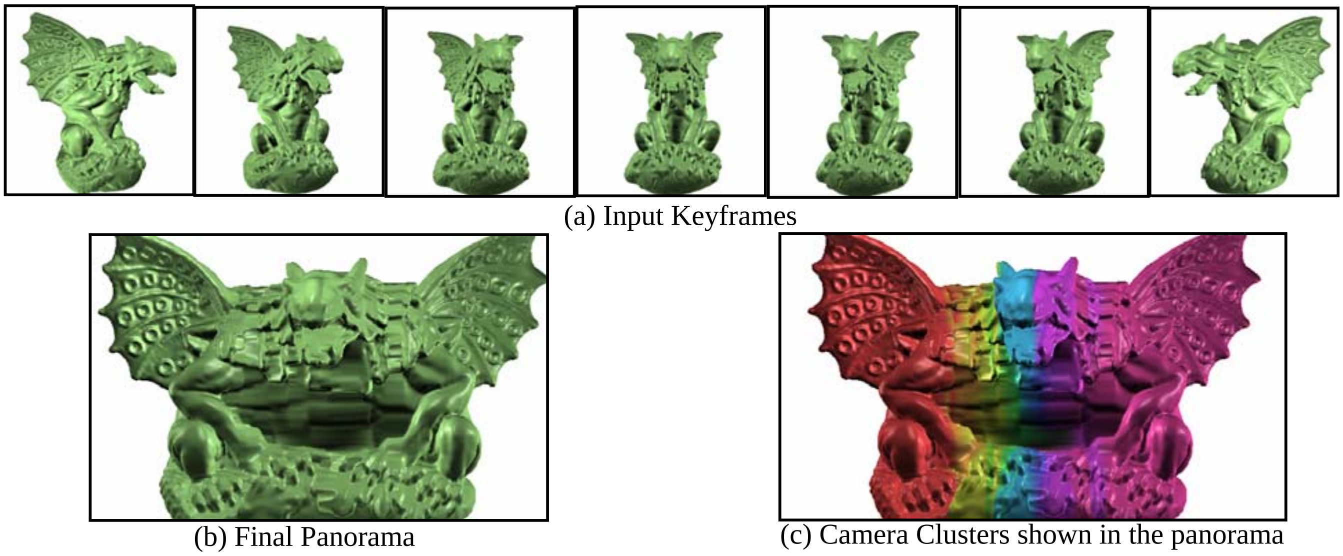


Fig. 17. (a) Shows few of the keyframes used to generate the final panorama (total number of keyframes = 11). (b) The final panorama. (c) Clusters corresponding to the keyframes.

References

[1] A. Agarwala, M. Agrawala, M. Cohen, D. Salesin, and R. Szeliski. Photographing long scenes with multi-viewpoint panoramas. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 853–861, New York, NY, USA, 2006. ACM Press.

[2] M. Agrawala, D. Zorin, and T. Munzner. Artistic multiprojection rendering. In *Proceedings of Eurographics Rendering Workshop 2000*, pages 125–136. Eurographics, 2000.

[3] A. V. Bartrolí, R. Wegenkittl, A. König, and E. Gröller. Nonlinear virtual colon unfolding. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[4] J. Blinn. Where am i? what am i looking at? In *IEEE Computer Graphics and Applications*, volume 22, pages 179–188, 1988.

[5] S. Bruckner and M. E. Gröller. Volumeshop: An interactive system for direct volume illustration. In H. R. C. T. Silva, E. Gröller, editor, *Proceedings of IEEE Visualization 2005*, pages 671–678, Oct. 2005.

[6] P. Coleman and K. Singh. Ryan: rendering your animation nonlinearly projected. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 129–156, New York, NY, USA, 2004. ACM Press.

[7] P. Coleman, K. Singh, L. Barrett, N. Sudarsanam, and C. Grimm. 3d screen-space widgets for non-linear projection. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 221–228, New York, NY, USA, 2005. ACM Press.

[8] M. Gleicher and A. Witkin. Through-the-lens camera control. *Siggraph*, 26(2):331–340, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[9] C. Grimm. Post-rendering composition for 3d scenes. *Eurographics short papers*, 20(3), 2001.

[10] S. Grimm, S. Bruckner, A. Kanitsar, and M. E. Gröller. Flexible direct multi-volume rendering in interactive scenes. In *Vision, Modeling, and Visualization (VMV)*, pages 386–379, Oct. 2004.

[11] X. Hou, L.-Y. Wei, H.-Y. Shum, and B. Guo. Real-time multi-perspective rendering on graphics hardware. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 79, New York, NY, USA, 2006. ACM Press.

[12] A. Kanitsar, R. Wegenkittl, D. Fleischmann, and M. E. Groller. Advanced curved planar reformation: Flattening of vascular structures. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Y. Kurzion and R. Yagel. Interactive space deformation with hardware-assisted rendering. *IEEE Comput. Graph. Appl.*, 17(5):66–77, 1997.

[14] E. C. LaMar, B. Hamann, and K. I. Joy. A magnification lens for interactive volume visualization. In H. Suzuki, L. Kobbelt, and A. Rockwood, editors, *Proceedings of Ninth Pacific Conference on Computer Graphics and Applications- Pacific Graphics 2001*, pages 223–232, Los Alamitos, California, 2001. IEEE, IEEE Computer Society Press.

[15] H. Löffelmann and E. Gröller. Ray tracing with extended cameras. *Journal of Visualization and Computer Animation*, 7(4):211–227, 1996.

[16] D. Marttin, S. Garcia, and J. C. Torres. Observer dependent deformations in illustration. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 75–82, New York, NY, USA, 2000. ACM Press.

[17] M. J. McGuffin, L. Tancau, and R. Balakrishnan. Using deformations for browsing volumetric data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 53, Washington, DC, USA, 2003. IEEE Computer Society.

[18] V. Popescu and D. G. Aliaga. The depth discontinuity occlusion camera. In *SI3D*, pages 139–143, 2006.

[19] P. Rademacher. View-dependent geometry. In *Siggraph*, pages 439–446. ACM, ACM Press/Addison-Wesley Publishing Co., 1999.

[20] A. Roman, G. Garg, and M. Levoy. Interactive design of multi-perspective images for visualizing urban landscapes. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 537–544, Washington, DC, USA, 2004. IEEE Computer Society.

[21] K. Singh. A fresh perspective. In *Proceedings of Graphics Interface 2002*, pages 17–24, 2002.

[22] R. Szeliski. Video mosaics for virtual environments. *IEEE Computer Graphics and Applications*, 16(2):22–30, 1996.

[23] S. Takahashi, N. Ohta, H. Nakamura, Y. Takeshima, and I. Fujishiro. Modeling surperspective projection of landscapes for geographical guide-map generation. *Comput. Graph. Forum*, 21(3), 2002.

[24] L. Wang, Y. Zhao, K. Mueller, and A. Kaufman. The magic volume lens: An interactive focus+context technique for volume rendering. In *Proceedings of IEEE Visualization (VIS) 2005*, pages 367–374, 2005.

[25] D. N. Wood, A. Finkelstein, J. F. Hughes, C. E. Thayer, and D. H. Salesin. Multiperspective panoramas for cel animation. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 243–250. ACM, ACM Press/Addison-Wesley Publishing Co., 1997.

[26] J. Yu and L. McMillan. A framework for multiperspective rendering. In *Rendering Techniques*, pages 61–68, 2004.

[27] J. Yu and L. McMillan. General linear cameras. In *ECCV (2)*, pages 14–27, 2004.

[28] A. Zomet, D. Feldman, S. Peleg, and D. Weinshall. Mosaicing new views: The crossed-slits projection, 2003.

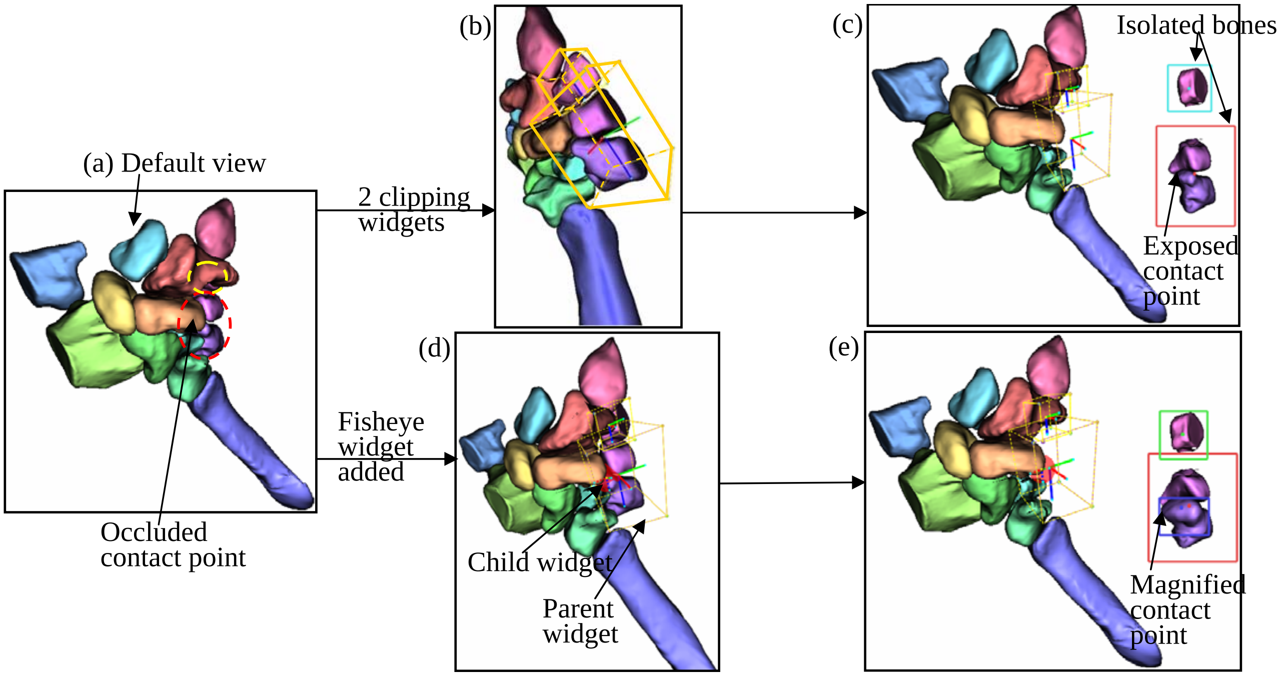


Fig. 19. Our goal is to expose the contact points between the bones within the red oval and also expose the bone lying within the yellow oval as seen in (a). Note that both these features are completely occluded in the default view. We add two clipping widgets, each for the relevant bones as seen in (b). Thus we see all three bones completely in the final view in (c). To magnify the contact point between the two bones we add a fish-eye widget as a child widget to the clipping widget seen in (d). In the final view, (e), the contact point appears magnified.

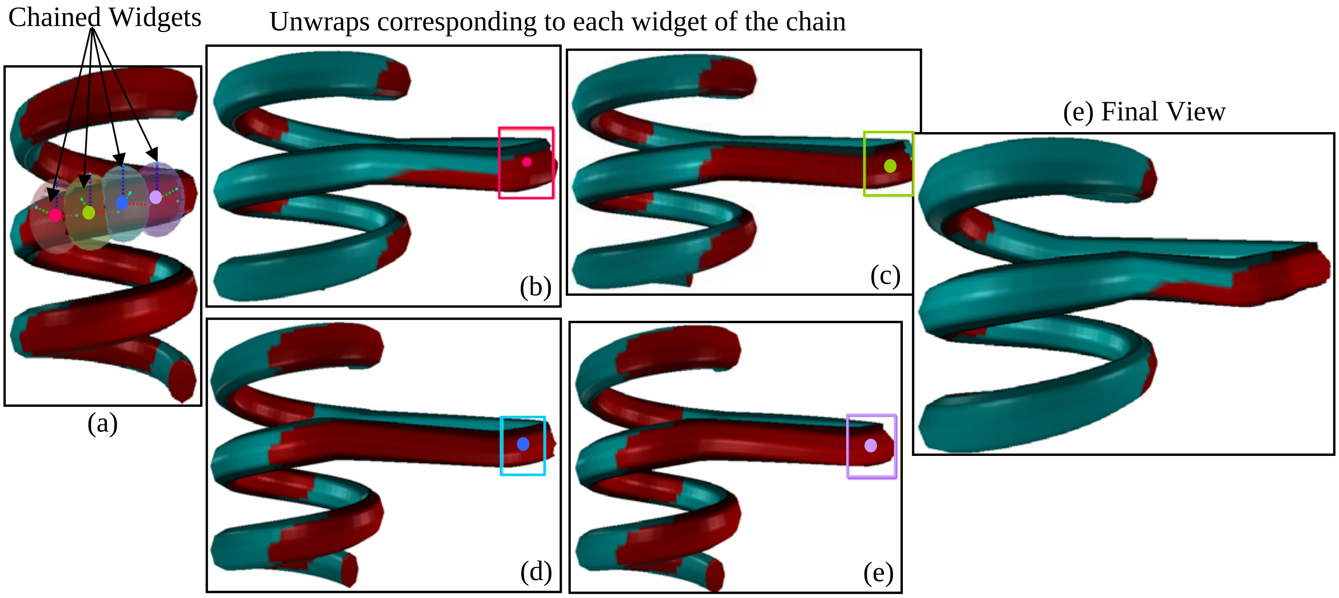


Fig. 20. Our goal is to unwrap a large portion of one of the coils of the helix. We add several instances of a chained unwrap widget along the back of the helix as shown in (a). Each unwrap widget produces a unique unwrapping shown in figures (b)-(e). These unwrap are combined together to produce the final view as shown in (e). In addition to the local cameras being blended together, the destination positions of consecutive widgets are shifted as described in Section 10.3.