Application-independent accurate mouse placements on surfaces of arbitrary geometry

Harlen Costa Batagelo Wu Shin Ting
State University of Campinas
School of Electrical and Computer Engineering
Campinas, São Paulo, Brazil
{harlen,ting}@dca.fee.unicamp.br

Abstract

Most direct manipulation tasks rely on precise placements of the cursor on the object of interest. Commonly, this requires the knowledge of distinct application-dependent geometry attributes of this object computed on the CPU. In this work, we present a simple yet general GPU-based framework for computing such attributes without depending on application-specific algorithms. In particular, it provides, for each pixel of the rendered model, elements of discrete differential geometry that require only the vertex data and 1-ring connectivity of each vertex stored in video memory. We validate our framework by showing how, only by providing correct mouse placements even when the geometry is modified on GPU, it can support the implementation of many direct manipulation tasks presented on the literature. Implementation results are presented.

1. Introduction

Pointing and selecting (picking) an object of interest using a pointing device are fundamental actions of any interactive graphics application based on the *direct manipulation* interaction style [10]. By combining these basic tasks, one can devise more complex interactions such as 3D painting, geometry creation, placement and editing.

Traditionally, the data required for precise 3D direct manipulation actions is computed with respect to the original geometry, mostly in analytical expressions stored in system memory. This integrated approach, widespread by the Unified Graphics Architecture (UGA) [14], allows that an interaction processing takes advantage of continuous geometric modeling techniques associated with the manipulated objects to generate appropriate visual feedbacks. For rendering, such geometry must be converted to a set of primitives that are supported by the graphics library. The most known graphics primitives are triangle meshes. Two issues may, however, arise from this paradigm:

1. Loose of generality: It is desirable to have a graphical user interface (GUI) toolkit designed to make graphical interactive application development easier. In the UGA approach, different representations of models will require different implementations of interaction algorithms for performing the same interaction task. For example, ray picking with models represented as parametric and implicit functions will require the implementation of a particular intersection algorithm for each representation.

2. Incoherence between visualization and interaction: A unambiguous mapping between what is presented to the user and what is represented in the underlying system is also an essential ingredient for assisting a user in direct manipulation tasks. On today's graphics processing unities (GPUs), the mesh geometry may be further deformed by nonrigid transformations on a vertex or geometry processor. Since the geometry attributes for interaction are still computed on the basis of the original geometry, this may introduce incoherences between what the user acts on and what the event handler can recognize from the user's action. Such issue may be worsened when considering fragment perturbation due to 3D detail mapping techniques, which potentially increases the difference between the original geometry and its final appearance on screen.

In order to solve these issues, we have investigated a set of elementary functions for direct manipulations. According to [12], only two application-dependent differential geometry data suffice for interactive placement of a cursor on the surface of arbitrary geometry while the user smoothly moves the input device cursor on it. The data are the 3D position of the point $\mathcal{P}$ under the cursor's hotspot and the normal vector of the surface at $\mathcal{P}$. Since these two data

are also used in a general rendering pipeline such as those specified by OpenGL, it is possible to deviated from the UGA paradigm and propose, in analogy to the rendering, an application-independent surface snapping algorithm. This was extended in [1] to heed the evolution of the programmable graphics hardware. In Section 2 we discuss these results in the context of known previous works on devising application-independent interaction methods.

Recently, there has been increasing interest in developing tools that assist a user to place the cursor on specific features of the geometry which she/he interacts with. Among them, we may mention snapping to valleys and ridges of the geometry [13], hatching strokes in a pen-and-ink style painting according to the principal directions [8], and producing interactively suggestive contours [4]. A careful analysis, summarized in Section 3, leads us to conclude that the fundamental data of all these manipulation tasks may be reduced to differential geometry quantities of higher order, namely curvatures and their derivatives. As far as we understood, the evaluation of the differential geometry properties of higher order is carried out on the CPU, thus all these manipulation tasks are limited to non-deformable geometry.

The geometry ultimately sent to the graphics hardware is represented according to the primitives supported by the GPU. We ask ourselves whether we may take advantages of this geometry to estimate, on the GPU, differential geometry quantities such as tangent frames, local curvature tensors and covariant derivative of curvature tensors, as well as positioning data, identification data and user-defined values. If these quantities are estimated after geometry deformation on the vertex processor, these deformations will be taken into account for interaction. Thus, we may precisely guide the movements of a cursor on arbitrary geometry, even when it is deformed on the GPU in a per-vertex level. An estimator for curvatures and their derivatives on the GPU was presented by [2].

In this work we propose an interaction framework in which differential geometry attributes of meshes are computed in accordance with geometry modifications along a rendering process, without requiring application-specific algorithms or a scene database on system memory. These data, calculated on the basis of the current mouse position, are sufficient for placing the cursor on the screen according to the user's actions and intentions. For detecting the mouse movements, we still need to listen to the mouse events delivered by a window system residing on CPU. Hence, our framework also provides an interface to a windowing system. Moreover, our framework returns the application-specific attributes of the objects under the cursor's hotspot for appropriate visual feedbacks of the model under manipulation, such as coloring or texturing one of its facets. In Section 4 the data and control flow of this framework are described thoroughly.

With our proposed framework, the development of sophisticated 3D interactive applications becomes an easier task. From a set of on-the-shelf functions, the application developer only configures which differential geometry attributes are required for each task, and the framework will estimate them for the primitives sent to the GPU. In Section 5 we present results from an implementation and summarizes the paper in Section 6.

2 Related work

In 3D, selection is usually computed by a ray picking procedure which consists of propagating a ray in world space from the viewpoint through the cursor's position, then testing the intersection between this ray and the geometry stored in a scene database. The selected geometry is the intersected geometry closest to the projection plane. Besides identification data, additional geometry data computed at the intersection point may be used for the composition of more complex tasks. For example, texture coordinates at the intersection point may be used for 3D painting, and barycentric coordinates of the intersection point in the pointed triangle can be used in a mesh cutting operation.

The OpenGL API provides an application-independent selection approach using a rendering mode called *selection mode* [6]. In this mode, if a geometry intersects a user-defined clip volume, a *selection hit* is generated. Each selection hit reports an ID of the geometry and the minimum and maximum depth values of the geometry inside the clip volume. Since the selection hits are computed after vertex processing, it consistently takes into account geometry modified in the vertex processor. However, fragment perturbation is not considered and interaction tasks are limited to tasks based on identification and positioning data.

Positioning is often used with constraints that aim at improving the precision of the cursor placement. Bier [3] introduced the idea of a *snap-dragging skitter*, a 3D cursor controlled by a 2D pointing device. When constrained by a gravity function, it automatically snaps to nearby points, curves or surfaces in the scene. The skitter, also called *triad cursor*, gives an additional visual cue to the user, as it shows the 3D position and orientation of the primitive at the snapped point. It, however, requires the local tangent frame at the point of interest. To support this, the traditional ray picking algorithm is extended to return additionally the normal vector at the intersection point and the corresponding primitive identifier. However, since the intersection tests are computed on the CPU, deformation of geometry performed on the GPU is not taken into account.

Wu *et al.* [12] presented a strategy to smoothly track the triad cursor's motion on a surface of arbitrary geometry only on the basis of the tangent frame of the manipulated primitive at the 2D cursor's current location. This tangent frame

is defined by the mouse drag direction and the normal vector of surface at the point nearest the mouse position. The position of the cursor is adjusted before re-displaying it on the screen. Performing if continuously over the frames produces the perception that the cursor snaps to the surface.

Later, Batagelo and Wu [1] introduced the idea of processing these data directly on the GPU and storing the results into off-screen *geometry buffers* (*g-buffers* [9]) as pixelwise encoded colors. By reading back and decoding the contents of the g-buffers at the pixels pointed by the 2D cursor, 3D position and orientation of the visible rendered surfaces are obtained for snapping a triad cursor to them. Barycentric coordinates and identification of primitives are handled similarly. This technique takes into account both vertex and fragment level deformations. It is also simple to integrate with existing applications and it is not tied to a particular API. As a drawback, however, orientation data can only be computed if the vertex deformation function is differentiable, as the orientation data is obtained by computing partial derivatives of the deformation function. In addition, due to the reduced number of attributes computed, the algorithm is pretty much restricted to picking, surface snapping and a simple 3D painting technique.

Our framework increases the range of attributes that may be encoded in the g-buffers by estimating elements of differential geometry on the GPU. It also allows the use of user-defined per-vertex attributes for both indexed and non-indexed triangle meshes, and per-fragment attributes. These features allow the implementation of a wider range of interaction tasks, as we analyze in the next section.

3 Case studies

In this section we present different known 3D direct manipulation tasks, emphasizing its implementation aspects. The goal is to show that the essential data that they require may be reduced to the per-pixel differential geometric ones.

3.1 Picking

Picking permits to identify a specific object among all visible and detectable objects displayed on the screen. Common visual feedback for mouse picking is to highlight the primitive pointed by a free-movement cursor. In the paradigm of storing per-pixel interaction data, each pixel of the g-buffer contains an ID value [11]. The ID in the pixel under the cursor's hotspot identifies the selected model. In order to select all models intersected by the picking ray in a front-to-back order, the following sequence of steps may be used: (1) Select the frontmost model; (2) Render the scene again, but excluding the already selected model(s); (3) Repeat from step 1 until the pixel under the cursor's hotspot does not contain any ID. The same approach applies for identifying faces. In each iteration, the frontmost face is removed until there are no more face IDs.

3.2 Interaction maps

With texture mapping one may glue a rectangular array of data, or texture map, to the geometry. When the individual values in the texture map are the event-triggering data, they are called *interaction maps*. Provided that the contents of these maps are encoded for each pixel of the g-buffer, we may define distinct responses to each pixel. In this way, when the free-movement cursor hovers pixels coincident with the mapped interaction surface, specific tasks may be automatically performed. Pierce and Paush [7] show that this is particularly useful for accurate interactions with image-based models and models where the texture map contains most of the detail.

3.3 Snapping

Snapping a cursor to a feature of interest whenever it gets nearby is desired in most interactive applications. It alleviates the time consuming precise position of the mouse on the screen.

Snapping to vertices consists of constraining the cursor location to the vertex which is closest to the current pointer's location. It may be done by rendering the geometry as points, then computing the screen-space distance between the pixels with rendered vertices and the pixel under the cursor's hotspot. The cursor is moved to the position of the nearest pixel with a rendered vertex. The same idea applies for snapping to edges. In this case, the geometry is rendered in wireframe mode. This can also be used for snapping to the surface's contours or other image features by computing the data only to pixels that coincide with the features. For surface snapping, it is usual to visually feedback through a triad cursor aligned with the tangent frame of the surface at the position pointed by the 2D cursor, as in Bier's snap-dragging technique. To do that, we should also include the differential geometry attributes of depth (for computing 3D position) and tangent frame, besides ID.

3.4 3D painting

A simple painting of the 3D objects in a rendered scene may be accomplished by a one-to-one mapping between brush samples and texture samples. The brush samples are indeed points in the neighborhood of the cursor's hotspot. Hence, we may use the texture coordinates of the pixels around the brush position for changing the corresponding texture map. Painting in screen space is handled similarly. Instead of changing the texture map, the brush samples are splatted onto the surface as new primitives that use the 3D

position of the surface fragments rendered in each pixel. Though 3D painting can be done with a free-movement cursor, surface snapping may be used for showing the 3D brush location on the surface.

In a more sophisticated painting tool, the style of the brush samples may vary according to the geometry shape. In a pen-and-ink style painting, hatching strokes may be oriented in the direction of the principal curvatures in order to enhance shape cue, or emphasized in parts coincident with suggestive contours at a given viewpoint. This is done by constraining the cursor's movement to these features, which are calculated from higher-order differential geometry elements of the pixel under the cursor's hotspot.

3.5 Geometric snapping

Geometric snapping extends the notion of image snapping [5] to 3D meshes [13]. When the user selects a vertex of the mesh with the cursor, the cursor moves to a nearby geometric feature based on the evaluation of a movement cost function when the cursor moves from a vertex to another. This requires the availability of differential geometry elements for any point of the mesh.

A Gaussian smoothing filter may be used in order to blur the approximate curvatures before applying the cost function. This softens up local minimums and emphasize global minimums in which the cursor will snap to.

In our paradigm of using per-pixel interaction data, the principal curvatures of the mesh may be stored for each pixel in a user-defined screen region around the cursor. An image-space Gaussian filter is used to smoothen such values. The result is used in the evaluation of a per-pixel move cost function, as in the image snapping technique. As the cursor moves to a neighboring pixel, the procedure is repeated until the cursor reaches a global minimum feature.

4 Framework

The case studies lead us to conjecture that, at least for most known applications, the differential geometry attributes associated to the pixel pointed by the cursor are sufficient for correctly guiding the movements of a cursor on a surface of arbitrarily geometry. This motivates us to propose a unified architecture for storing and accessing these attributes while a user interacts in order to provide correct responses.

The framework is built on top of the graphics API and is visible to the programmer as an additional set of 3D interaction-supporting commands. The underlying code benefits from the general-purpose stream computation of the GPU to estimate the differential geometry elements, and inherit all interfaces to window system that a graphics card supports.

The framework is window system independent in the sense that it only provides functions to estimate the differential geometry attributes of discrete meshes at the focused points, but is not responsible for capturing these points. How to receive the events from a window system is of charge of the application, which issues the calls to get appropriate data from the g-buffers for further processing. To accomplish this, the application must initialize the interaction context by specifying the differential geometry attributes to be computed and the semantics bindings to be assigned for correct interpretations, as explained in Section 4.2. The framework uses such data to automatically estimate the differential geometry elements of the model and to encode them into the g-buffers using the output interface, as shown in Section 4.3.

4.1 Model

The processing stages of the framework are integrated into the render loop of the application, as illustrated in Figure 1. Stages 1, 2, 3, 4 and 5, which are carried out on the GPU, are backed up by user-defined callback functions containing the rendering calls. Just before rendering the actual geometry, the application issues a command (stage 1) that updates the geometry data according to user-defined per-vertex deformation shaders. A new estimation of the local differential geometry attributes is automatically performed (stage 2) and the results are cached into render textures for later use. After updating the original geometry according to the deformed attributes cached by the previous stage, the application may modify these attributes again, now with the computed differential geometry attributes (stage 3). During rasterization, the computed per-vertex attributes, as well as the user-defined attributes, are linearly interpolated to the fragments. After that, they are updated by per-fragment deformation shaders (stage 4) and encoded as colors into the g-buffers (stage 5). Usually after the actual rendering (not mandatory), the application may request the contents of such g-buffers (stage 6). These buffers are transfered back to the system memory.

Each stage is detailed as follows:

1. **Modification of vertex attributes**: This is performed if the model undergoes a changing in its vertex attributes. For each model, it starts by the framework invoking a callback function containing the drawing call of the model. At the vertex processor, a user-defined deformation shader modifies the original vertex attributes and produces a final geometry, still in object space, that can be used for the estimation of differential geometry quantities. The attributes of the deformed model are written to render textures.

2. **Geometry attributes computation**: This stage esti-

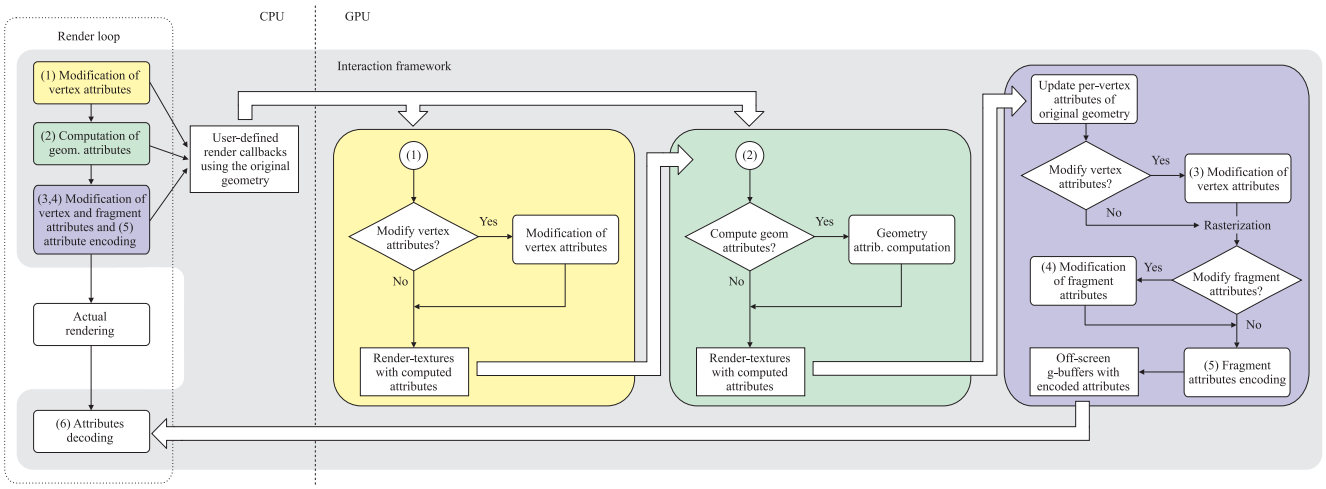

Figure 1. Framework's processing stages (gray shaded region) and the application's render loop.

mates the differential geometry quantities at each vertex of the models deformed in the stage of modification of vertex attributes. As in the previous stage, this starts by the framework running a callback function with the drawing call of the model. For each vertex, the GPU reads the deformed vertex attributes by sampling the render textures of the previous stage, and sampling pre-computed textures with connectivity data in order to compute the geometry quantities. The results are written to new render textures.

3. **Modification of vertex attributes**: Also triggered by a callback function, this stage starts in the vertex processor by sampling the render textures of the previous stages in order to update the vertex attributes before rasterization. As in the first stage, a user-defined deformation shader modifies the vertex attributes. However, these attributes now include the differential geometry properties just computed. This stage is useful for performing operations that depend on these new properties, such as applying a geometric transformation to a tangent frame that will be used by a 3D detail mapping algorithm in the fragment shader.

4. **Modification of fragment attributes**: During rasterization, the attributes computed on the GPU and defined by the user in the vertex buffer of the original geometry are linearly interpolated across the primitives. In the fragment processor, these fragment attributes are then modified by the user-defined deformation function.

5. **Fragment attributes encoding**: This is done just after the previous stage, in the same fragment shader. The modified attributes are encoded as color components of the g-buffers.

6. **Fragment attributes decoding**: In this stage the g-buffers are transfered to system memory. The attributes are decoded and made available to the application.

4.2 Input interface

The data required for the framework to compute the per-pixel attributes for interaction are the following, for each model under interaction:

- **Geometry data**: The vertex and index buffers of the original geometry used for the actual rendering, but with an additional user-defined value that contains a 0-based integer index of each vertex. These indices are used to determine the addresses of the texels of the render textures that cache the geometry data after attribute modification. In our implementation, the pointers to these buffers are set through commands `SetVertexBuffer` and `SetIndexBuffer`.

  When second and third order differential geometry attributes are computed, the framework also requires the list of 1-ring neighborhood of each vertex of the original geometry. In our implementation, this is computed internally from the vertex and index buffers. These geometry and connectivity data are transferred to textures in video memory that will be used for computing the differential geometry quantities.

- **Attributes to compute:** The set of attributes which should be created on-the-fly or processed from the deformed geometry in a per-vertex level, and then encoded for each interpolated fragment. In our implementation, the command `SetAttributes` is used to

specify any combination of the following attributes: (1) Depth in normalized device coordinates; (2) Normal vector in object space; (3) Texture coordinates; (4) Tangent and bitangent vectors aligned according to the parametrization of the texture coordinates, in object space; (5) Coefficients of the curvature tensor; (6) Principal curvatures and principal directions; (7) Coefficients of the tensor of curvature derivative; (8) User-defined value, which may be defined both for indexed and non-indexed geometry. For indexed geometry, the data is shared by all adjacent faces that use the vertex (*e.g.*, model and vertex IDs). For non-indexed geometry, each vertex may have a different value for each adjacent face (*e.g.*, face IDs and weights of barycentric equations).

- **Rendering callbacks:** User-defined callback functions containing the graphics API commands for setting up the render states used by the deformation shaders, and for issuing the drawing calls for indexed and non-indexed primitives (e.g., `glDrawArrays` in OpenGL, or `DrawPrimitive` in Direct3D) using the vertex buffers of the original geometry. In our implementation, these functions are set through the commands `SetUpdateCallback` (callback function for triggering stages 1 and 2) and `SetRenderCallback` (callback function for triggering stage 3, 4 and 5).

- **Semantic bindings:** A mapping between the usage semantic of each element of the original vertex buffer and the semantic interpreted by the framework. The semantics of the vertex buffer are those specified by the graphics API (*e.g.*, texture coordinates, normal vector, vertex color and point size). They may be mapped by the command `BindSemantics` to one of the following semantics of the framework: (1) Texture coordinates used to compute the tangent and bitangent vectors; (2) Vertex index; and (3) User-defined value for indexed geometry and non-indexed geometry.

  The binding of the vertex index semantic is the only mandatory. The vertex element containing the vertex position does not require this mapping, as it is always considered a vertex position by the framework.

- **Deformation shaders.** Definition of shader functions that accept as input a data structure containing the non-deformed vertex or fragment attributes and return the same data structure with the attributes modified. For the vertex deformation shader used in stage 1, this data structure contains the vertex position and the additional attributes defined by the semantic bindings. For stages 3 and 4, it also contains the differential geometry properties computed in stage 2.

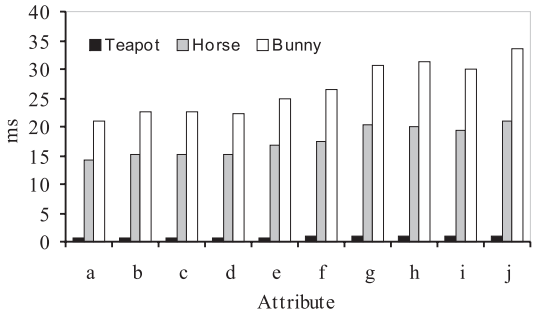

Figure 2. Processing time of different attributes.

In our implementation, these shaders are specified with commands `SetPreVertexDeform` (stage 1), `SetPostVertexDeform` (stage 3) and `SetPixelDeform` (stage 4).

4.3 Output interface

The main data returned to the application are the per-fragment attributes stored for each pixel of the g-buffer. This is done by a command `Decode` that returns a pointer to the contents of the g-buffers.

5 Implementation and results

The framework have been implemented as a C++ library in OpenGL and Direct3D. It is composed of two classes: one that initializes and manages the models under interaction, and another that is created through the manager class and contains the data specific to each model. In total, its interface consists of about 50 distinct commands that may be needed to produce accurate mouse placements in a graphics interactive application.

The performance of the framework depends mostly on the performance of the shaders used in stages 1, 3 and 4, followed by the number and type of attributes processed for each vertex. Figure 2 summarizes the average time, in milliseconds, obtained for computing different attributes on a Teapot (2,082 vertices), Horse 48,484 (vertices) and Bunny model (72,027 vertices). The attributes are: (a) No attributes; (b) Depth; (c) Texture coordinates; (d) User-defined value for indexed geometry and (e) non-indexed geometry; (f) Normal vector; (g) Tangent frame; (h) Curvature tensor; (i) Curvature tensor with principal directions and curvatures; (j) Tensor of curvature derivative. The test platform was an AMD Athlon 64 3200+ (2.2 GHz) 1 GB RAM, with a NVIDIA GeForce 7900 GTX with 512 MB

VRAM. In these results, the stage 1 was executed even when no attributes were computed. The overhead shown in (a) is mainly due to the dynamic flow control instructions and texture sampling instructions used in the vertex shaders of stages 1 and 3. Though the overhead for estimating the differential geometry elements is most evident, it is significantly more efficient than a CPU-based estimation, as shown in [2].

In order to validate our hypothesis that per-pixel attributes composed of differential geometry elements and user-defined data suffice for precisely positioning the mouse in accordance with the need of a broad range of applications, the framework was used for prototyping some of the interaction tasks presented in Section 3. Figure 3 shows snapshots of these applications: (a) Picking; (b) Determining all faces along the picking ray; (c) Interaction maps; (d) Surface snapping; (e) Snapping to borders; (f) Painting and sculpting a relief mapped quad; (g) Snapping to principal directions; (f) Geometric snapping. In the following we describe how they differ with respect to the settings of the commands for inputting data in the framework:

- **Picking**. We call `SetAttributes(USERDEFI)` to inform that the per-pixel interaction data should contain a user-defined value for indexed geometry: the model ID. `BindSemantics(TEXCOORD1 ↦ USERDEFI)` is used to inform that such value is defined in a set of texture coordinates of the geometry's vertex buffer.

- **Interaction maps**. `SetAttributes(USERDEFI)` informs that the per-pixel attribute is a single user-defined value. `BindSemantics` is not used, since the user-defined value is not a vertex attribute of the vertex buffer. Instead, we use `SetPixelDeform` to define a fragment shader that samples the interaction map and writes its value in the output register of the user-defined value.

- **Snap to surface, vertex, edge or borders**. We call `SetAttributes(DEPTH, TBN)` to inform that per-pixel interaction data should contain only a depth value and a tangent basis. `BindSemantics(TEXCOORD0 ↦ TEXCOORD)` is used to inform that the texture coordinates for computing the tangent vectors are found in the first set of texture coordinates of the vertex buffer. According to the type of cursor snapping desired, the callback functions set with `SetRenderCallback` will trigger the rendering of geometry with filled triangles (for surface snapping), wireframe (for edge snapping) or points only (for vertex snapping). For snapping to borders, filled triangles are used, but `SetPixelDeform` defines a fragment shader that filters the fragments which do not lie on the borders of the rendered model.

- **3D painting**. For performing 3D painting in texture space, we call `SetAttributes(TEXCOORD)` to inform that the per-pixel attribute should contain only the mapped texture coordinates. `BindSemantics (TEXCOORD0 ↦ TEXCOORD)` informs that such texture coordinates are found in the first set of texture coordinates of the vertex buffer.

- **Geometric snapping**. `SetAttributes(DEPTH, CURV)` is called to inform that the per-pixel attributes should be composed of a depth value, principal curvatures and principal directions. We do not use `BindSemantics` in this case, as the computation of depth and estimation of curvatures will always use the `POSITION` vertex attribute of the vertex buffer.

6 Conclusion

In spite of the increased flexibility of today's GPUs for performing geometry modeling and animation tasks without the intervention of the CPU, few efforts have been made for handling direct manipulation with 3D geometry deformed in a programmable rendering pipeline. We conjecture that the main hindrance roots in the design philosophy of the existing graphics cards, which are window system independent, whereas the success of a direct manipulation processing relies on both the event handler of a window system and the rendering pipeline of a graphics hardware. The more flexible and powerful are the GPUs, the wider is the gap between what the event handler of a window system is able to process on CPUs and what is rendered.

We have proposed an interaction framework that circumvents this ever growing distance problem between an action triggering and its visual feedback. It supports the implementation of direct manipulation tools which consistently take into account deformation of geometry on the GPU. The main idea is to use the actual visualization pipeline to process all the attributes required for each interaction task and then to store such attributes in the image-space domain as encoded pixel colors. Since the direct manipulation is performed on the basis of pixel data, it may work with any primitive handled by the rendering pipeline.

Our framework does not require a scene database in system memory for computing the attributes, since geometric information is obtained directly from the primitives stored in video memory. In special, differential geometry quantities can be computed entirely on the GPU after the deformations performed in a per-vertex level.

We have implemented the proposed framework as a C++ library in OpenGL and Direct3D, along with sample tools that demonstrate that it can be used to implement a number of direct manipulation actions presented in the literature.

It is worth mentioning that, because the estimation of the geometrical quantities is done on the basis of vertices delivered by the application and the missing values are estimated from linear interpolations, our framework cannot automatically handle non-linear deformations performed in the per-fragment level, such as per-fragment normal and depth perturbation due to normal mapping or relief mapping. In such

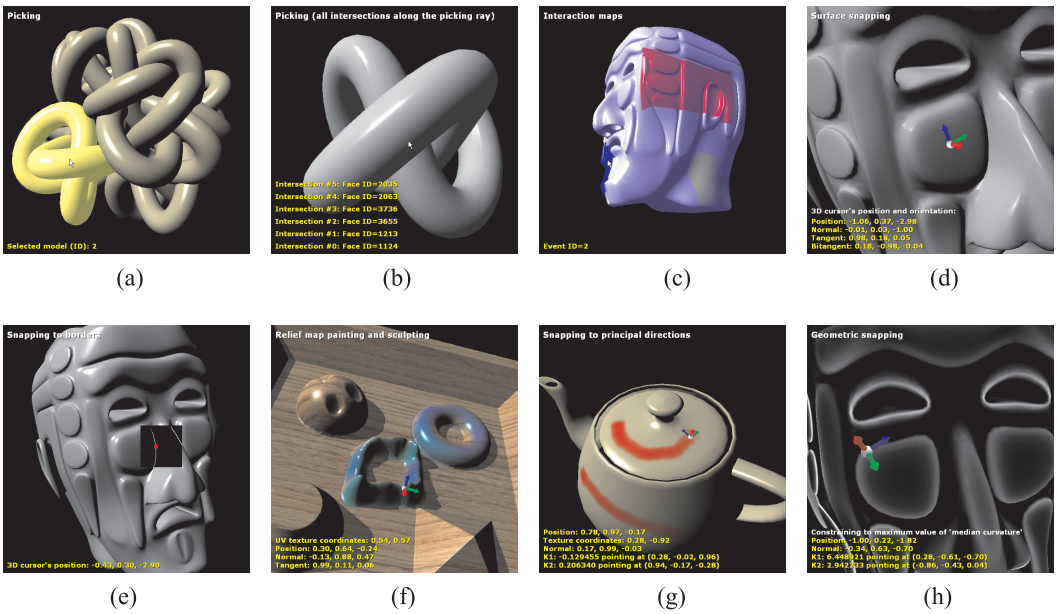

Figure 3. Snapshots of the direct manipulation applications implemented with the framework.

cases, the user should implement a per-fragment deformation shader that updates the attributes according to the detail mapping used in the actual rendering. As further work, we intend to focus on this issue by using estimators of differential geometry properties on the basis of data available in image space only.

References

[1] H. C. Batagelo and S.-T. Wu. What you see is what you snap: snapping to geometry deformed on the gpu. In *Proc. of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 81–86, Washington, DC, USA, April 2005. ACM Press.

[2] H. C. Batagelo and S.-T. Wu. Estimating curvatures and their derivatives on meshes of arbitrary topology from sampling directions. *The Visual Computer*, 2007. http://dx.doi.org/10.1007/s00371-007-0133-8.

[3] E. A. Bier. Skitters and jacks: interactive 3d positioning tools. In *Proc. of the 1986 Workshop on Interactive 3D Graphics*, pages 183–196, Chapel Hill, NC, USA, 1987. ACM Press.

[4] D. DeCarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *Proc. of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pages 15–24, New York, NY, USA, 2004. ACM Press.

[5] M. Gleicher. Image snapping. In *Proc. of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 183–190. ACM Press, 1995.

[6] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, release 1.* Addison-Wesley Publishing Co., Reading, MA, USA, 1st edition, 1993.

[7] J. S. Pierce and R. Pausch. Specifying interaction surfaces using interaction maps. In *Proc. of the 2003 Symposium on Interactive 3D Graphics*, pages 189–192, Monterey, CA, USA, 2003. ACM Press.

[8] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *Proc. of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, page 581, New York, NY, USA, 2001. ACM Press.

[9] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *Proc. of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 197–206, Dallas, TX, USA, 1990. ACM Press.

[10] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[11] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[12] S.-T. Wu, M. Abrantes, D. Tost, and H. C. Batagelo. Picking and snapping for 3d input devices. In *Proc. of the XVI Brazilian Symposium on Computer Graphics and Image Processing*, pages 140–147, So Carlos, SP, Brazil, October 2003.

[13] K.-H. Yoo and J.-S. Ha. Geometric snapping for 3d meshes. In *International Conference on Computational Science*, pages 90–97, Krakow, Poland, June 2004.

[14] R. C. Zeleznik, D. B. Conner, M. M. Wloka, D. G. Aliaga, N. T. Huang, P. M. Hubbard, B. Knep, H. Kaufman, J. F. Hughes, and A. van Dam. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics*, 25(4):105–112, 1991.