A GPU-based Architecture for Supporting 3D Interactions

Harlen C. Batagelo, Wu Shin-Ting (advisor)

[1]Department of Computer Engineering and Industrial Automation
School of Electrical and Computer Engineering
University of Campinas - UNICAMP
13083-970, Campinas, SP, 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 application-dependent geometry attributes of this object computed on the CPU. We present a simple yet general GPU-based framework for computing such attributes without depending on application-specific algorithms. It provides, for each pixel of the rendered model, application-defined values and elements of discrete differential geometry computed on the GPU. We show how this framework can support the implementation of many direct manipulation tasks presented in the literature, even when the geometry is modified on the GPU.*

1. Introduction

Direct manipulation of 3D geometry using 2D pointing devices requires algorithms that compute local geometry attributes for interaction tasks of selection and constrained positioning (snapping). Traditionally, CPU-based techniques are used to provide such attributes by performing a series of intersection tests between a selection ray and the geometric models. How these tests are carried out depend on the geometry's representation used in the application. In addition, such approach does not take into account the modifications of geometry that may happen on the GPU, thus producing a gap between what is visualized and what the user may expect to interact with.

In this work, we employ a new paradigm of obtaining geometric attributes which does not require application-dependent computations. This is based on the hypothesis that differential geometry properties and application-defined attributes made available in a per-pixel basis suffice for many direct manipulation tasks presented in the literature. Generality may be achieved by using only the data sent to the programmable graphics hardware in order to estimate the differential geometry quantities. Therefore, direct manipulation may be accomplished even if a scene database on system memory is not available.

Based on this new paradigm, we propose an interaction framework in which differential geometry attributes of triangle meshes are efficiently computed in accordance with all geometry modifications along the rendering pipeline. 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. With this architecture, 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. The doctorate thesis, videos, related publications and source code of the proposed architecture are available at http://www.dca.fee.unicamp.br/projects/mtk/batagelo/.

2. Related work

In 3D, selection is usually computed by a CPU-based ray casting 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 (e.g. normal vector and curvatures) may be used for the composition of more complex tasks.

Positioning is often used with constraints that aim at improving the precision of the cursor placement. Bier introduced the idea of a *snap-dragging skitter*, a 3D cursor controlled by a 2D pointing device [Bier 1987]. 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 casting algorithm should be 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.

Since these selection and positioning algorithms are based on techniques proposed before the advent of programmable graphics hardware, the efficient interaction with geometry deformed on the GPU has been an open problem, and current workarounds consist of simply duplicating the geometry data and deformation code in system memory in order to perform the processing on the CPU when events for interaction are triggered. This problem tends to be worsened, as GPUs can create primitives on geometry processors and manage scenes that exist only in video memory.

3. Case Studies

In this section we present known 3D direct manipulation tasks, emphasizing its implementation aspects. We consider as representative tasks: picking, snapping, 3D painting and geometric snapping. The goal is to show that the essential data that they require may be reduced to per-pixel differential geometry properties and application-defined data. Our proposal of a GPU-based application-independent interaction framework is based on such hypothesis.

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 rendered object contains an identifier (ID) value. The ID stored in the pixel under the cursor's hotspot identifies the selected model.

When the individual values of a texture map of a texture mapped object are event-triggering data, this texture is called an *interaction map* [Pierce and Pausch 2003]. Provided that the contents of these maps are encoded for each pixel of the rendered object, we may define distinct responses for such pixels. In this way, when the free-movement cursor hovers pixels coincident with the mapped interaction surface, different tasks may be automatically performed according to the application-defined values of the interaction map. This is particularly useful for accurate interactions with image-based models and models where the texture map contains most of the detail.

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, each pixel also includes the differential geometry attributes of depth and tangent frame, besides the identifier.

Painting of 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. The brush samples are splatted onto the surface as new primitives that use the 3D position of the rendered surface fragments.

In *geometric snapping*, 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 [Yoo and Ha 2004]. This requires the availability of higher-order differential geometry elements (e.g. principal curvatures and principal directions) for each pixel of the rendered object.

4. Framework

The case studies support the conjecture that per-pixel geometric attributes and application-defined attributes are sufficient for interaction tasks composed of selection and constrained positioning. It motivated us to validate our idea by implementing a simple snapping architecture which only requires the position and normal vector of the surface at each selected point [Wu et al. 2003, Batagelo and Wu 2005]. Based on the promising results, we propose in this work an architecture built on top of the graphics API, that provides to the programmer a set of commands for supporting the processing of these attributes stored in off-screen render textures called *geometry buffers* (*g-buffers*).

To concretize our proposal, we further investigated an efficient way to estimate differential geometry elements for each pixel of the rendered objects. Differently from the previous snapping architecture, we took advantage of the general purpose stream computation of the GPU in order to implement novel algorithms for estimating differential geometry properties up to the third order, solely on the basis of the vertex data available on the GPU [Batagelo and Wu 2007b].

The framework is window system independent in the sense that it only provides functions to handle the attributes of the focused points, but is not responsible for using these attributes for interaction feedback. 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 attributes to be computed and the semantics bindings to be assigned for correct interpretations. Our code has been designed to inherit all interfaces

to the window system that a graphics card supports. Therefore, the framework can get user interaction data to automatically estimate the differential geometry elements of the model and to encode them into the g-buffers using an output interface [Batagelo and Wu 2007a].

4.1. Processing Flow

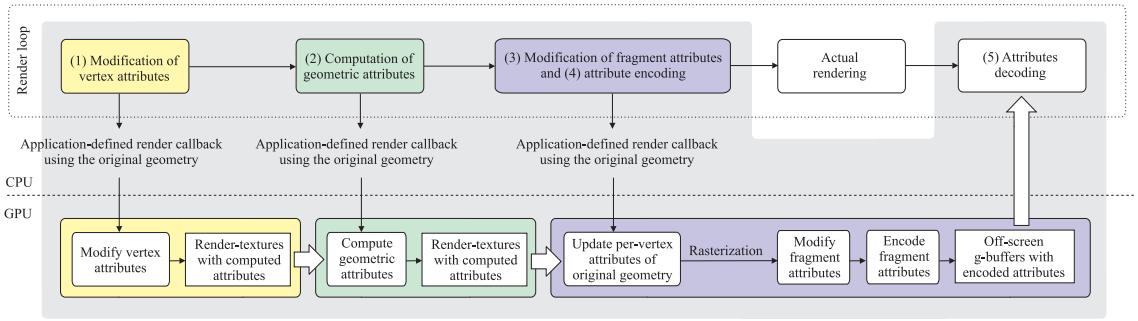


Figure 1. Framework's processing stages (gray shaded region) and the application's render loop. Thick arrows indicate texture data flow.

The processing stages of the framework are integrated into the render loop of the application, as shown in Figure 1. Each stage is detailed as follows:

1. **Modification of vertex attributes**: This is performed if the geometry undergoes a change in its vertex attributes. It starts by the framework invoking a callback function containing the drawing call of each model. At the vertex processor, an application-defined deformation shader modifies the original vertex attributes and produces a final geometry, still in object space. The modified attributes are written to render textures.
2. **Computation of geometric attributes**: As in the previous stage, this starts by the framework running a callback function with the drawing call of each model. The differential geometry quantities are computed by using the data obtained from the render textures of the previous stage, and from pre-computed textures with connectivity data. The results are written to new render textures.
3. **Modification of fragment 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. During rasterization, these per-vertex attributes are linearly interpolated along the primitives. In the fragment processor, these fragment attributes are then modified by an application-defined function.
4. **Encoding of fragment attributes**: 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.
5. **Decoding of fragment attributes**: On demand, the g-buffers are transferred 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 attributes for interaction are the following, for each model:

- **Geometry data**: Vertex and index buffers of the original geometry used for the actual rendering, but with an additional application-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 contain the geometry data after attribute modification. In our implementation, the pointers to these buffers are set through commands `SetVertexBuffer` and `SetIndexBuffer`.
- **Attributes to compute:** Set of attributes that will be processed and encoded for each pixel. 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) Curvature tensor; (6) Principal curvatures and principal directions; (7) Tensor of curvature derivative; (8) Application-defined value. 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:** Callback functions containing the graphics API commands for setting up the render states used by the deformation shaders, and API commands for issuing drawing calls using the vertex buffers of the original geometry (e.g. `glDrawArrays` in OpenGL, or `DrawPrimitive` in Direct3D). In our implementation, these are set through the commands `SetUpdateCallback` (for triggering stages 1 and 2) and `SetRenderCallback` (for triggering stage 3).
- **Semantic bindings:** 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 elements are specified by the graphics API. They are 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) Application-defined value; (3) Vertex index (mandatory binding).
- **Deformation shaders.** 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. In our implementation, these shaders are specified with `SetVertexDeform` and `SetPixelDeform`.

Besides the data required for each model under interaction, the framework also requires that the application indicates the screen space coordinates of a rectangular *region of interest* (ROI) in which the per-fragment attributes will be computed for all models. This is set through a command `SetROI` with arguments composed of the top-left and bottom-right screen coordinates of the rectangle. This region corresponds to the dimensions of the g-buffers. Primitives outside the ROI are discarded and fragment processing is reduced to fragments within this region.

4.3. Output interface

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

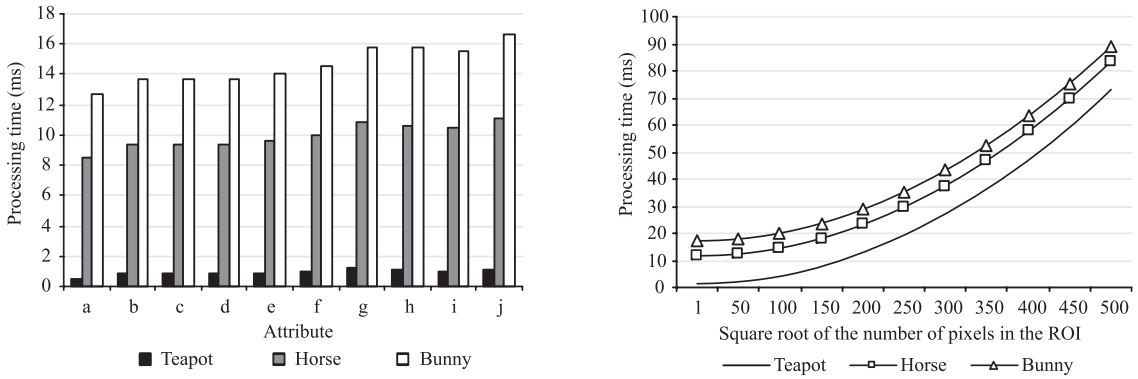


Figure 2. Left: Processing time of different attributes. Right: Processing time as a function of the size of the ROI.

5. Implementation and results

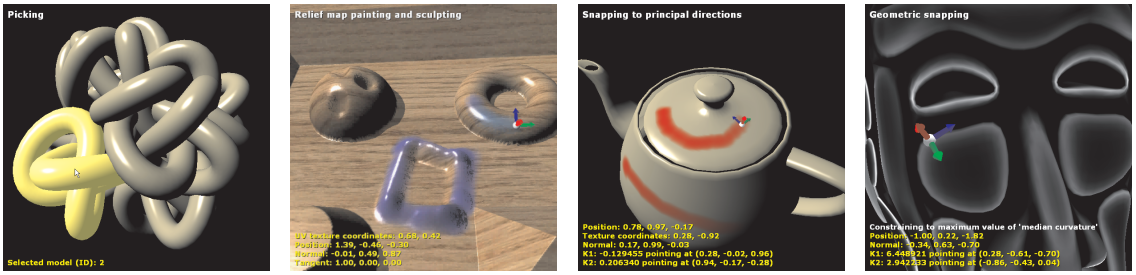
The framework has been implemented as a C++ library in OpenGL and Direct3D. It is composed of a class that initializes and manages the models under interaction, and another that contains the data specific to each model. It is portable between Windows and Linux.

The performance of the framework depends mostly on the performance of the vertex shaders used in stages 1 and 3, followed by the number and type of attributes processed for each vertex. We present processing times for three models of distinct sizes: teapot (2,082 vertices), horse (48,484 vertices) and bunny model (72,027 vertices). Figure 2 (left) summarizes the average time, in milliseconds, for calculating different attributes on them. The attributes are: (a) No attributes; (b) Depth; (c) Texture coordinates; (d) Application-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 3500+ with 2 GB RAM, and a NVIDIA GeForce 8800 GTX with 768 MB VRAM. In these tests, 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. Although the overhead for estimating the differential geometry elements is most evident, the efficiency is much better than a CPU-based estimation, as we show in [Batagelo and Wu 2007b].

Figure 2 (right) shows timing results of a surface snapping task as a function of the square root of the number of pixels in the ROI. The attributes computed are depth, application-defined value for indexed geometry, tangent frame and curvature tensor. The results include the time for decoding and transferring the attributes to the CPU. The performance degrades linearly, which confirms that the bottleneck of the processing flow is indeed in the vertex processing stages.

The framework was used for prototyping surface-oriented interaction tasks presented in Section 3. Figure 3 shows snapshots of some of these applications: (a) Picking; (b) Painting and sculpting a relief mapped quad; (c) Snapping to principal directions; (d) Geometric snapping. In the following we describe how they differ with respect to the settings of the commands of the framework's input interface:

- **Picking**. We call `SetAttributes(APPDEFI)` to inform that the per-pixel



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

Figure 3. Direct manipulation applications implemented with the framework.

data should contain an application-defined value `APPDEFI` for indexed geometry, which is the model ID. `BindSemantics(TEXCOORD0 ↦ APPDEFI)` informs that such value is defined in the first set of texture coordinates (`TEXCOORD0`) of the vertex buffer. Finally, we call `SetROI` to set the ROI to the pixel under the cursor's hotspot.

- **Interaction maps**. `SetAttributes(APPDEFI)` is called to inform that the per-pixel attribute is an application-defined value. `BindSemantics` is not used, since the application-defined value is not provided by the vertex buffer. Instead, we use `SetPixelDeform` to set a fragment shader that samples the interaction map for obtaining the application-defined value. The ROI is the same as in picking.
- **Snap to surface, vertex, edge or borders**. For snapping, we call `SetAttributes(DEPTH, TBN)` to inform that per-pixel attributes should contain only a `DEPTH` value and a tangent frame `TBN`. `BindSemantics(TEXCOORD0 ↦ TEXCOORD)` is used to inform that the texture coordinates (`TEXCOORD`) for computing tangent and bitangent vectors are found in the first set of texture coordinates of the vertex buffer. `SetROI` is called to define the area of influence of gravity around the cursor's hotspot. According to the type of snapping desired, the callback functions set with the command `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 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)` is used to inform that such coordinates are found in the first set of texture coordinates of the vertex buffer. `SetROI` sets the paint brush region.
- **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 (`CURV`). `BindSemantics` is not used, and `SetROI` is called as in snapping to surfaces, vertices, edges or borders.

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. 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.

The major contribution of this work is the design of an interaction framework that circumvents this ever growing distance 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 rendering pipeline to process 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 (triangle meshes, point-based and image-based models).

Our framework does not require a scene database in system memory for computing the attributes. Instead, geometric information is obtained directly from the primitives stored in video memory. Algorithms have been devised for computing first, second and third order differential geometry quantities on the GPU after the deformations performed in the per-vertex level. This supplementary contribution is indeed useful for many other applications. 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 surface-oriented direct manipulation tasks presented in the literature. Due to the special interest that this work has raised in the community of visualization of medical data and data of fluid dynamics simulation, we now plan to extend the architecture to handle volumetric data rendered with isosurfaces extraction techniques. This may require the availability of geometric attributes in a volume region around a 3D cursor.

References

Batagelo, H. C. and Wu, S.-T. (2005). What you see is what you snap: snapping to geometry deformed on the GPU. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 81–86. ACM Press.

Batagelo, H. C. and Wu, S.-T. (2007a). Application-independent 3D interaction using geometry attributes computed on the GPU. In *Proceedings of the 20th Brazilian Symposium on Computer Graphics and Image Processing*, pages 19–26. IEEE CS Press.

Batagelo, H. C. and Wu, S.-T. (2007b). Estimating curvatures and their derivatives on meshes of arbitrary topology from sampling directions. *The Visual Computer*, 23(9,11):803–812.

Bier, E. A. (1987). Skitters and jacks: interactive 3D positioning tools. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pages 183–196. ACM Press.

Pierce, J. S. and Pausch, R. (2003). Specifying interaction surfaces using interaction maps. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 189–192. ACM Press.

Wu, S.-T., Abrantes, M., Tost, D., and Batagelo, H. C. (2003). Picking and snapping for 3D input devices. In *Proceedings of the 16th Brazilian Symposium on Computer Graphics and Image Processing*, pages 140–147. IEEE CS Press.

Yoo, K.-H. and Ha, J.-S. (2004). *Computational Science - ICCS 2004*, volume 3039/2004 of *Lecture Notes in Computer Science*, chapter Geometric Snapping for 3D Meshes, pages 90–97. Springer Berlin / Heidelberg.