Picking and Snapping for 3D Input Devices

WU, SHIN - TING, MARCEL ABRANTES, DANIEL TOST, AND HARLEN COSTA BATAGELO

Image Computing Group (GCI)
Department of Industrial Automation and Computer Engineering (DCA)
School of Electrical and Computer Engineering (FEEC)
State University of Campinas (Unicamp)
P.O.Box 6101, 13083-970 - Campinas, SP, Brazil
{ting,tost,ra003168,harlen}@dca.fee.unicamp.br

Abstract. A picking mechanism (pointing and indicating) with the cursor is essential for any direct-manipulation application. The windowing systems, under which control a direct-manipulation application runs, provide facilities that together with special utility routines allow identifying which object within the region the user is pointing at. Such picking algorithms have been widely used for selecting objects under a 2D mouse cursor. In this paper, we present a simple yet effective application-independent 3D picking algorithm for 3D input devices. We also discuss a differential geometry based surface constraint that can be applied to the 3D cursor position for improving points matching. In order to demonstrate the techniques, two sample applications using a 3D input device are shown.

1 Introduction

In almost all direct-manipulation graphics systems an interaction task, such as scaling and rotating an object, is built on top of two basic interactions: selecting and dragging with a pointing device. Moreover, almost universally acceptable interaction syntax in a windowing system is selecting an object by clicking on it with a mouse and dragging the object by moving the mouse in a 2D space while the button is down. Besides mouses, there is a variety of complementary interaction devices such as joystick, trackballs and spaceballs for improving the control on the movement of an object. For example, the joystick is often used to move an object selected by a mouse.

The decreasing cost of input devices and the increasing computer power lead to continuing improvement in interaction devices both in precision and in dimension in which the device works. There are a variety of 3D interaction devices that provide 3D position and orientation. Among them we may mention the *spaceball*[1] that allows accurate fine positioning in 3D-space. Even though, the usually recommended procedure for manipulating an object with a spaceball is to pick it with a mouse and then drag or rotate it with the spaceball. This motivates us to look for application-independent 3D positioning and picking mechanisms with which we may perform the two basic interactions with the same input device also in 3D space so as already it is done in 2D space.

Differently from the concept of 3D cursor given by Foley *et al.* [4] which is controlled by a 2D cursor, Mesquita [6] defined a 3D cursor as an extension of a 2D cursor. The 2D cursor is the visible representation on the screen of the 2D pointing device's position. Because the cursor must frequently have a resolution of a single pixel, it is of common practice designating one single pixel of a cursor as the *hotspot*. In an analogous way, a 3D cursor is the visible representation on the screen of the 3D pointing device's position and its unprojected shape in 3D always has a single hotspot voxel (*volume element*).

Furthermore, Mesquita [6] developed an algorithm for emulating 3D positioning functionality for 2D input devices. Two movement modes are defined on the basis of the functions developed by Navarro *et al.* [3]: one is on the xy-plane and the other is on the xz-plane in the viewing volume (Figure 1). The user can switch from one mode to another by simply pressing or releasing the middle mouse button. The method may appear restrictive once the user is working in two dimensions at a time. However, experiments with different users let us conclude that most users can quickly adapt to the syntax of the context switching and feel that the emulated pointing device moves freely in 3D.

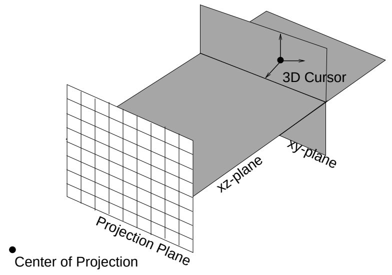

Figure 1: Emulation of 3D pointing devices.

[1]http://3dconnexion.com/products/5000/

In this paper our focus is on the 3D picking mechanism for input devices capable of providing an absolute or relative position in 3D space. In other words, given a position in 3D we would like to determine which object is at that position without resorting to application-dependent functionalities. A 3D snapping algorithm on the basis of differential geometry of the manipulated surface is also devised for a user to position a pointing device more accurately in 3D. An implementation with OpenGL selection and pick commands is presented and the experiments were carried out with use of the emulated 3D pointing device and the spaceball 3003.

In the next section our application-independent 3D picking algorithms are presented. Next, we discuss how we can control its movement constrained to a surface of interest in 3D. Then in Section 4, we show how they can be efficiently implemented with use of OpenGL. In Section 5, results of application of these algorithms to a 3D pointing device, more precisely the spaceball 3003, are given. We could successfully determine which object it points at in 3D and program an imaginary gravity field around each existing surface for constraining its movement. Finally, some concluding remarks are drawn.

2 3D picking

A 3D picking problem can be reduced to a problem of determining the object that intersects at a given point the eye-ray fired from the center of projection through the pixel's center into the unprojected scene. In principle, this problem can be solved by determining all the objects that intersect the ray and performing point-in-solid inclusion tests to find out which object contains the specified 3D point (Figure 2). This approach is application-dependent, once it requires the application-dependent point inclusion test algorithms.

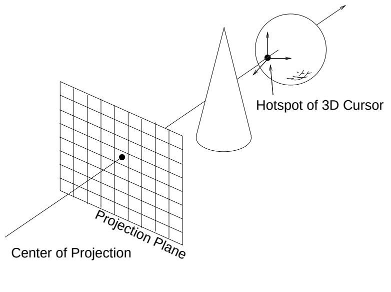

Figure 2: Point-in-solid test approach.

As an alternative, one may reduce the problem to a one-dimensional problem by determining the intersection intervals along the eye-ray for each object. Then, the object that is pointed at can be obtained with a point-in-segment inclusion test (Figure 3). This approach is computationally attractive when z-buffer algorithm is applied for visible-surface determination. In this case, the depth value of each point in the viewing volume is always passed to the graphics hardware. Therefore, gathering and sorting z-coordinate values along an eye-ray for defining the intersection intervals can be carried out without resorting to the application.

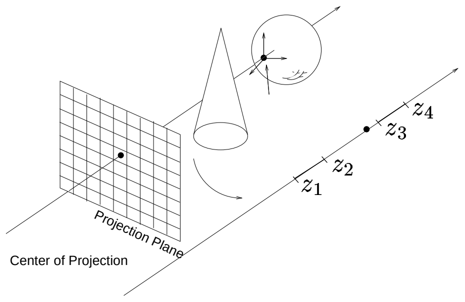

Figure 3: Point-in-segment test approach.

If a scene only contains convex objects, the point-in-segment test can be performed more efficiently by graphics hardware that is capable of returning the minimum and maximum z values of all vertices that intersected the eye-ray. In this case, we can determine whether an object is picked by drawing it and comparing the returned z values with the depth value of the 3D cursor. Figure 4 pictures the front and back faces of the cone and the sphere that intersect the eye-ray on which the 3D cursor lies. The depth value of the 3D cursor neither lies in the z-interval of the cone nor in the z-interval of the sphere. Therefore, the 3D cursor is outside of both objects.

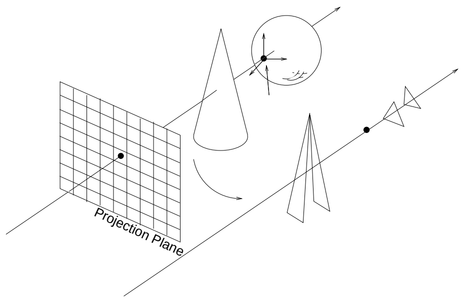

Figure 4: Improved point-in-segment test approach.

For concave polyhedra, the point-in-segment test can also be performed by graphics hardware. In special, a hardware stencil buffer may be used to store the parity counting of objects' faces and thus determining whether a given point is inside the object on the basis of the fact that if an eye is outside of an object of interest and an eye-ray intersects the object, this eye-ray always intersects an even number of points. In this case, we can issue drawing commands to a graphics hardware for drawing only the faces of the object of interest that appear at the pixel under the cursor before (or after) the position of the 3D cursor and inverting the stencil buffer at this pixel. The cursor is outside the object if and only if the number of inversions is even, i.e., the stencil buffer at the pixel under the cursor should remain unchanged. Once the algorithm is applied on each object at time, for the sake of clarity, only the state of stencil refering a torus is shown in Figure 5. It induces five discrete intervals for stencil buffer: 1 (s_1), 2 (s_2), 3 (s_3), 4 (s_4), and 5 (s_5) along the picking ray. As only the faces before the cursor are drawn in the stencil buffer, the stencil buffer value of the pixel in consideration is even, the 3D cursor is outside the object. Again, the procedure explores the graphics hardware capabilities.

Actually, stenciling is not a essential part of this algorithm. Since this method performs only parity checking using operations of inversion and does not use stencil comparison functions, this feature may be emulated with any buffer that can be used to count pixel overdraw, such as an accumulation buffer (by accumulating the overlapping fragments) or color buffer via alpha-blending (by using additive blending of fragments).

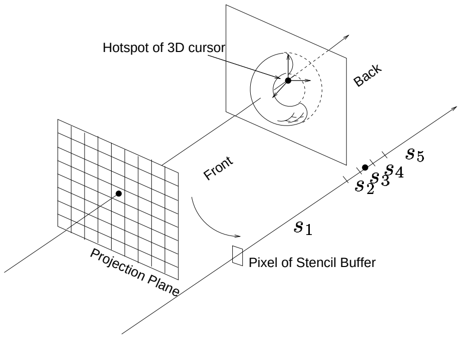

Figure 5: Point-in-segment test for concave objects.

3 3D snapping

In most cases, to place a 3D cursor exactly on the object of interest or to constraint its movement precisely on the surface of an object are essential for accurate interactions with shapes. For example, if we want to trim a surface, then we want the 3D cursor to be on the surface as we are moving it for defining a trimming curve. A technique that can help these tasks is snapping, which can round the actual position of the pointing device to some more appropriate position.

The simplest snapping technique is grid-snapping, also known as vertical or horizontal alignments. In this case, the snapping problem is reduced to a simple rounding of the device position to a point of a specified grid. There is, however, a number of snapping to an arbitrary shape problems that cannot be reduced to vertical or horizontal alignments. To our knowledge, for handling an arbitrary shape, the problem is reduced to the computation of the nearest point on the shape, most of which require potentially time-consuming point-and-shape or the ray-and-shape intersection algorithms.

In this section we present a novel procedure for snapping to a smooth convex closed surface $\mathcal{S}$ on the basis of two local geometry properties at each point $\mathcal{P}$: the normal vector $\vec{n}$ and a vector $\vec{u}$ on the tangent plane of the surface at $\mathcal{P}$. The existence of a "tangent plane" at all points of $\mathcal{S}$ is guaranteed because of the smoothness condition. With $\vec{n}$ and $\vec{u}$, we define a snapping reference coordinate (SRC) system at each point $\mathcal{P}$, $SRC(\mathcal{P})$, using $\vec{n}$ and $\vec{u}$ and consider that the 3D cursor locally "snaps" to this plane called *snap plane*. That is, its motion is restricted to this plane in the neighborhood of $\mathcal{P}$ (Figure 6)

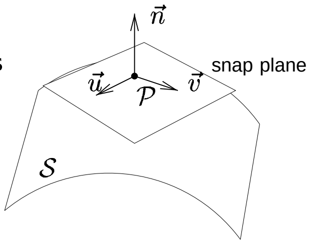

Figure 6: Snapping reference coordinate system.

The next point on $\mathcal{S}$ to be reached by the 3D cursor from $\mathcal{P}_i$ in the world coordinate (WC) system is computed as follows (Figure 7):

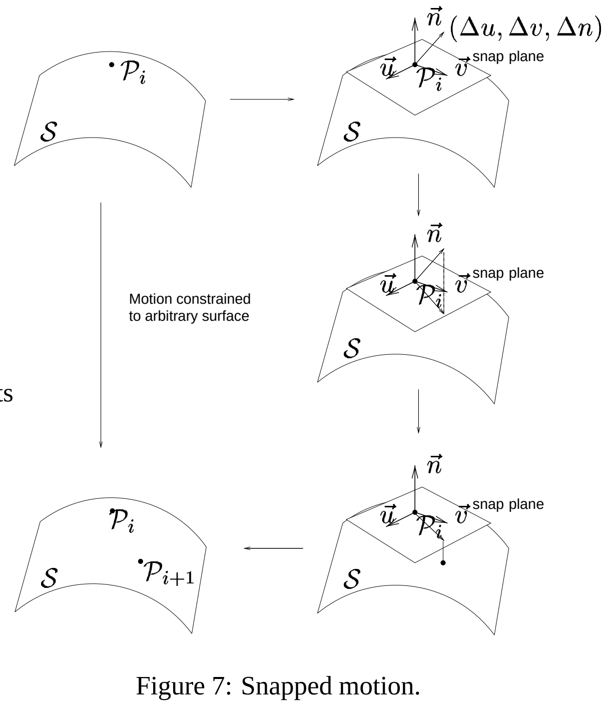

Figure 7: Snapped motion.

1. change the reference system of $\mathcal{P}_i$ to the $SRC(\mathcal{P}_i)$

system;

2. the displacement $(\Delta u, \Delta v, \Delta n)$ of the 3D cursor is rounded to $(\Delta u, \Delta v, 0.0)$ and added to the coordinates of $\mathcal{P}_i = (0, 0, 0)$ in $SRC(\mathcal{P}_i)$;

3. project this point parallel onto $\mathcal{S}$ in the direction of $\vec{n}$ to obtain $\mathcal{P}_{i+1}$; and

4. change the reference system of $\mathcal{P}_{i+1}$ back to WC.

We may perform the z-buffer visible-surface determination for getting the "n-values" of all the objects that are mapped to the pixel under the snapped 3D cursor position (u, v) in SRC(P), including the depth value n of the surface of interest.

The procedure is repeated after computing the $SRC(\mathcal{P}_{i+1})$ reference system with the two differential geometry properties of $\mathcal{S}$ at $\mathcal{P}_{i+1}$: the normal vector and one direction vector on the tangent plane.

4 OpenGL Implementation

To be self-contained, the commands related with the selection and picking mechanism provided by OpenGL are briefly described before we show how they can be used to implement the third paradigm given in Section 2 and partially the snapping algorithm described in Section 3. A detailed 2D picking and selection algorithm using selection mode is presented elsewhere [5].

4.1 OpenGL

OpenGL is designed to support 2D and 3D interactive applications without knowing the windowing specific functions and the input events [5]. It provides a selection mechanism that automatically reports which objects are drawn inside a specified viewing volume; and a picking mechanism that allows to restrict drawing to a small viewing volume, typically near the cursor in two modes: `GL_RENDER` and `GL_SELECT`. In the `GL_RENDER` mode, the drawing commands are used to alter the contents of the framebuffer. In the `GL_SELECT` mode, the drawing commands are used to construct a stack of names of primitives of interest. The contents of the framebuffer do not change until exiting this mode.

A viewing volume is defined by the matrices stacked in both the *modelview* stack and the *projection* stack. Those matrices can be built by issuing the transformation commands provided by OpenGL. Particularly, a projection matrix that restricts drawing to a small region of the viewport is created by calling `gluPickMatrix()`.

To use the selection mechanism of OpenGL, a selection context must be prepared as follows:

1. Specify the array with `glSelectBuffer()` to be used for storing the objects that are in the name stack and that appear at the same pixel under the cursor;

2. Enter into selection mode by specifying `GL_SELECT` with `glRenderMode()`;

3. Initialize the name stack of objects of interest using `glInitNames()`;

4. Define the viewing volume to be used for selection with `gluPickMatrix()` and/or other projection commands. It is important that these projection commands are issued in the projection mode by specifying `GL_PROJECTION` with `glMatrixMode()`; and

5. Issue drawing commands and commands to manipulate the name stack, such as `glLoadName()` or `glPushName()`, so that each primitive of interest has an appropriate name assigned.

When the selection mode is exited, such as switching to render mode by specifying `GL_RENDER` with `glRenderMode()`, OpenGL returns a list of primitives that would have intersected the given viewing volume. The elements of the list are also known as *hit records*. Each hit record consists of the following items:

1. the number of names on the name stack when the hit occurred;

2. both the minimum and maximum of z values of all vertices of the primitives that intersected the specified viewing volume since the last recorded hit. These two values are each multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer. This means that the corresponding floating-point values can be restored by dividing the unsigned value by `0xffffffff`; and

3. the contents of the name stack at the time of the hit, with the bottommost element first.

It is important to note that picking is usually triggered by an input pointing device and the viewing volume should be dynamically updated according to the mouse position returned by the mouse event handler for ensuring the correctness of the list of picked objects.

OpenGL Utility Library (GLU) includes several routines that encapsulate OpenGL commands. Besides `gluPickMatrix()`, there are two routines that help simplify the conversion between the basis of reference systems: `gluProject()` and `gluUnproject()`. Respectively, they transform world coordinates into window coordinates and vice-versa.

4.2 Picking

The picking algorithm we implemented is the third alternative presented in Section 2. It exploits OpenGL's selection mode both for reducing the number of objects of the scene which we use to test against the cursor hotspot and for obtaining the minimum and the maximum z values of each of these objects that intersected the viewing volume.

Our algorithm was implemented as follows:

1. Set as the parallel viewing volume only the region near the cursor location.
2. Compute the hit list under the projected hotspot using the standard OpenGL 2D picking algorithm as described in Section 4.1.
3. Project the 3D hotspot onto the screen with `gluProject()` to obtain its depth value in the parallel viewing volume.
4. For each object in the hit list: Compare the hotspot's z-position with the object's minimum and maximum depth returned by the hit record. The 3D cursor is outside the object if the hotspot's z coordinate is outside the object's depth range. Otherwise, the 3D cursor is inside the object.

4.3 Snapping

Because of limited graphics power available in our laboratory, our implementation of the 3D snapping algorithm is mostly application-dependent in the sense that only the steps 1 and 2 of the snapping algorithm given in Section 3 are performed by OpenGL (a potential solution using programmable graphics hardware is outlined in the concluding remarks). This, however, does not compromise our main objective which is to validate the feasibility of the idea presented: concise and simple interface between the graphics hardware and the application-dependent geometric modeling functionalities.

The algorithm consists of mapping the displacement provided by the input device to the surface tangent plane under the cursor point. The cursor point and two orthogonal direction vectors (*tangent* and *normal*) are provided by the application. A new point on the surface tangent plane is calculated using the displacement as a linear combination of the direction vectors. This point is returned to the application, which shall project it on the surface. The process is repeated at each cursor movement.

The choice for two directional vectors over the normal plus one directional vector was made because with the directional vectors we can calculate the new point in the tangent plane using a linear combination instead of applying affine transformations. This choice requires less calculation, thus raising the efficiency of the snapping.

The snapping implementation uses callback functions, *i.e.*, user-registered functions which are triggered by system events. We define three callback functions: two of then are responsible for providing the directional vectors and the other one for the projection of tangent plane point onto the surface.

4.4 Integration with MTK

The Manipulation Toolkit (MTK) is a 3D graphics library developed on top of OpenGL, aiming at providing a simple and direct interface to the fundamental operations of 3D graphics rendering and interaction [3, 7]. Our 3D picking and snapping algorithms are integrated into MTK.

MTK is comprised of the mtkCore class and eight "independent" abstract classes (Figure 8):

- Graphics Models (mtkDisplayList).
- Cameras (mtkCamera).
- Lights (mtkLights).
- Selection (mtkSelection).
- Guides (mtkGuide).
- Constraints (mtkConstraint).
- Draggers (mtkDragger).
- 3D Cursors (mtkCursor).

The mtkCore coordinates the interaction between these classes by delegating requests in order to realize a semantic action. For example, the Selection passes to the mtkCore the identifier of selected elements and the method cameraPointer() in mtkCore is responsible for deciding whether a dragger (a pictorial representation to convey visual feedback to user's actions) or a 3D model should be notified to handle the subsequent events. Instead of being self-triggered in response to the user actions, the visual feedback of 3D models and draggers is triggered in response to the attribute change determined by applications.

Guides provides additional depth cues when necessary and Constraints comprise of a set of constraining functions for moving more precisely the cursors and draggers in the scene.

In comparison with the existing graphics libraries, one new class, the mtkCursor, was included in MTK to enhance 3D interaction support. An instance of Cursors is controllable through a 2D or 3D input device via the class mtkxWindow. The 3D picking and snapping algorithms are yet additional features that improve the 3D interaction facilities provided by MTK.

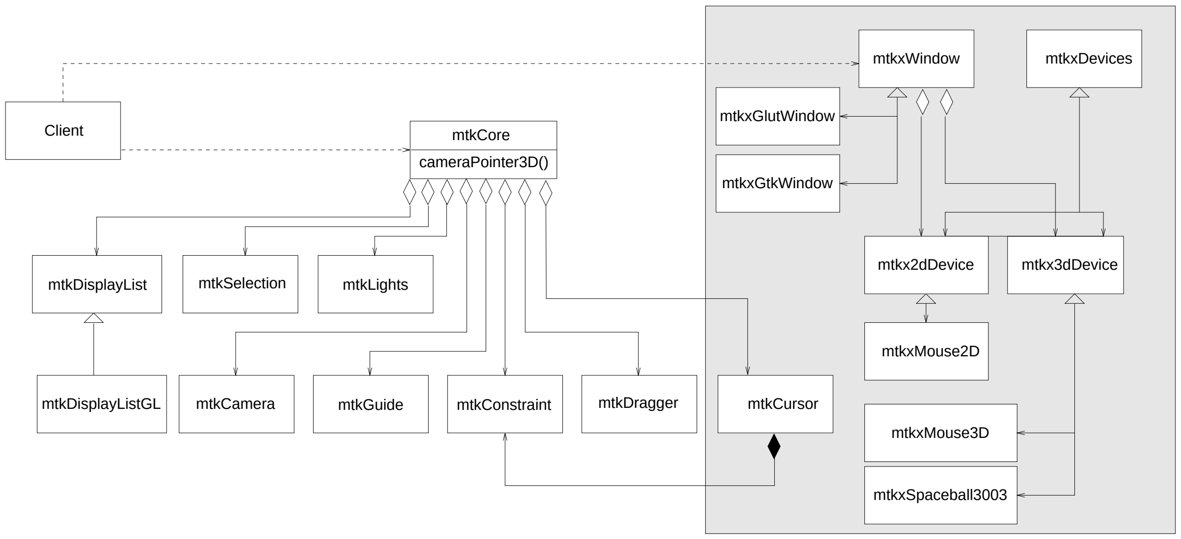

Figure 8: MTK framework

5 Sample applications

The implemented algorithms use the spaceball as the 3D input device. The spaceball permits rotation and translation movements in 3D space according to the pressure the user applies on the controller's ball. The ball senses pressure one applies to it - pushes, pulls, twists - and uses that information to correspondingly move the object of interest on the screen. Pulling up or pushing down the ball displaces the object in y-direction; pushing to the left or right will move it in x-direction; and pushing the ball away or towards the user will move it in z-direction. By simply twisting the ball in any direction we can rotate the object about the x, y, or z-axis. Combining all these movements, one have 6 degrees of freedom over any object on the screen.

To illustrate the application of the proposed algorithms for controlling a spaceball, we developed two sample examples on top of MTK and GLUT. GLUT is the OpenGL Utility Toolkit which implements a simple windowing application programming interface (API) for OpenGL functions. For communication with spaceball we developed a set of functions on the basis of the LibSBall library[2] as a idle-event callback functions. Then, whenever the windowing system is in the idle state, this function is activated, the communication is initiated and the input events are processed conveniently.

5.1 3D Picking

We developed a simple cube matching game that allowed us to observe and evaluate the picking process. A set of 3 pairs consisting of wireframed and solid cubes are randomly placed in the scene (Figure 9.a). The user can move the 3D cursor in the space by pulling and pushing the spaceball and select a 3D model of interest by pressing the right button (Figure 9.b). Whenever an object is selected, the object and its corresponding wireframe are highlighted in the same color (red, blue or green) and the subsequent actions on the spaceball are only applied on it (Figure 9.c). The goal of the game is moving spatially the selected cube through the spaceball in order to put it inside its corresponding wireframe (Figure 9.d).

[2]http://jedi.ks.uiuc.edu/ johns/projects/libsball

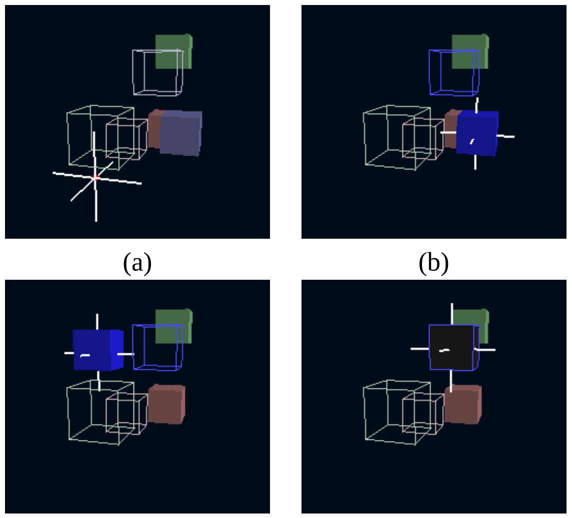

Figure 9: Matching cubes game.

5.2 3D Snapping

Two applications were designed to evaluate the 3D snapping algorithm. One that snaps the 3D cursor on a sphere and the other on a cube. In each one the application program must correctly deliver the x- and y-directions on the tangent plane at each point $\mathcal{P}$ reached by the cursor 3D.

For a sphere, we used the following procedure for obtaining the $x-$ and y-directions [8] at $\mathcal{P}$: we determine the two vectors from the center $\mathcal{O}$ of the sphere, $\mathcal{OP}$ and $\mathcal{O}N_p$, where N_p is the north pole of the sphere. The x direction is the cross product between $\mathcal{O}N_p$ and $\mathcal{OP}$. The y direction is the cross product of the normal vector at $\mathcal{P}$ and x.

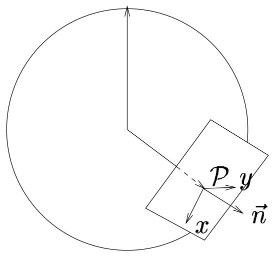

Figure 10: x- and y-directions on a tangent plane.

Figure 11 pictures two snapshots of the application program with the movement of a 3D cursor constrained on the sphere.

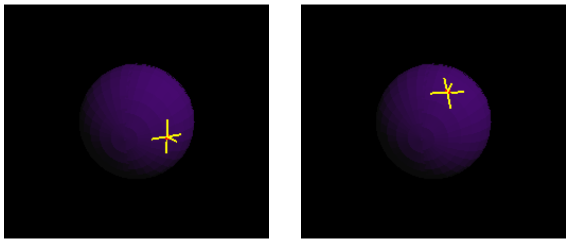

Figure 11: Snapping on a sphere.

Since the objective of our sample programs is to demonstrate the feasibility of our purpose, the heuristic we adopted for handling the edges of the cube is very simple: we predefined the x- and y- directions for each of six faces. Hence, the orientation of displacements only change when the cursor crosses the edge. Additionally, we chose its adjacent face for constraining the further movement of the cursor as depicted in Figure 12.

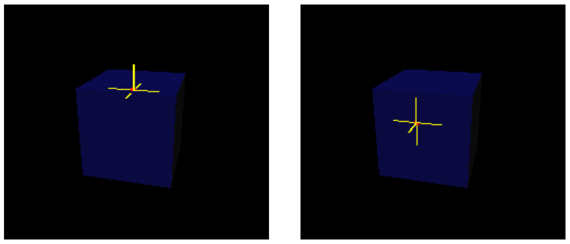

Figure 12: Snapping on a cube.

6 Concluding Remarks

We presented two direct manipulation tools for three-dimensional geometry using 3D input devices: a picking algorithm for pointing and indicating polyhedral objects and a snapping technique for smooth convex surfaces. Typical applications of such algorithms include virtual reality, geometry modeling applications and games.

Our 3D picking is totally application-independent; it is based solely on simple functionalities available on graphics hardware. The algorithm is also robust, since it does not depend on reading z-buffer values to determine which object contains the 3D cursor. Thus, it may be widely used in applications that represents complex polyhedral structures, making the work all easier because it will not be necessary to evaluate such structures. Despite our implementation is for convex objects, we plan to exploit a user-defined clipping plane and a stencil buffer – though any buffer capable of storing information about pixel overdraw, including the accumulation buffer or even a color buffer with additive alpha-blending, may be used – to efficiently check the parity counting of intersections between the picking ray and the scene objects in order to correctly handle the concave objects.

The 3D snapping technique is intended for interactively moving a 3D mouse cursor on smooth convex surfaces. A differential geometry based surface constraint is used to decrease the cost of performing intersection tests between the surface and the eye-cursor ray, yet improving points matching. Although the presented implementation still relies mostly on the application for providing points on the surface while the 3D cursor moves on it, we were successful in defining a simple interface between the graphics hardware and the application. We showed that for each point only two types of data are required: the normal vector and its orthogonal projection on the surface.

We discussed an implementation of the proposed algorithms. Both the picking and snapping algorithms were integrated in the Manipulation Toolkit (MTK), a high-level direct manipulation library using OpenGL and C++. The 3D picking algorithm uses the OpenGL's selection mode to decrease the number of objects for testing the depth range, while staying application-independent. However, our snapping algorithm is application-dependent, since the application is responsible for calculating the direction vectors and intersection points.

As future work, we intend to exploit programmable real-time graphics processing units as an alternative implementation of the picking algorithm and to extend the snapping algorithm for using graphics hardware, thus achieving application independence. The 3D picking may be implemented as a single vertex and fragment program [2] in which the user-defined clipping plane is emulated using the `texkill` shader instruction [1], *e.g.*, on cards that do not provide support for `glClipPlanes()`. In addition, the snapping algorithm may use graphics hardware to help determining the surface's normals of the points projected from the tangent planes onto the surface. Instead of calculating an exact intersection of the projected point with the surface and then computing the normal vector, the graphics hardware may only "snap" the projected point to discrete surface points with pre-computed normals. Such technique is intended to work with surface meshes that are capable of providing predefined tangent basis at the meshes' vertices to the vertex program. For each displacement event, the snapped object may be rendered in an off-screen buffer with the tangent basis components stored as color components, then reading back the (linearly) interpolated direction vectors for the pixel that corresponds to the displaced point in tangent plane converted to window coordinates. This is somewhat different from the current snapping algorithm, since the 3D cursor will snap only to visible parts of the geometry. However, as well as the 3D picking algorithm, this new algorithm will be totally application-independent.

7 Acknowledgments

We would like to acknowledge CNPq for financial support under the grant number 141685/2002-6 and FAPESP under the grant numbers 02/01161-3 and 02/01162-0.

References

[1] NVIDIA Corp. *Clip-planes with texkill*, 2001. (`http://developer.nvidia.com/view.asp?IO=clipplanes_texkill`)

[2] Fosner, R. *Real-Time Shader Programming*, Morgan Kaufmann, 2002.

[3] Fernandes, F.N. *An architecture for constructing interfaces with 3D direct manipulations*, (in Portuguese). State University of Campinas, M.Sc. Thesis, 1998. (`http://www.dca.fee.unicamp.br/projects/prosim/publications/thesis/navarro-1998-mtk.pdf`)

[4] Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[5] Neider, J., Davis, T., and Woo, M. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, release 1.* Addison-Wesley, 1993.

[6] Mesquita, L.A.G. *Final Technical Report about the Design of 3D Cursors for 2D Input Devices*, (in Portuguese). State University of Campinas, Technical Report, March 2001. (`http://www.dca.fee.unicamp.br/projects/prosim/publications/reports/lmesq-2001-cursor3d.pdf`)

[7] Malheiros, M.G., Fernandes, F.N., and Wu, S.-T. MTK: A direct 3D manipulation toolkit. *Proc. of SCCG '98*, 81–88, April 1998.

[8] Shaffer, C. A. Getting around on a Sphere *Graphics Gems II*, ed. by James Arvo, 172–173, AP Profissional, 1991.