What You See Is What You Snap: Snapping to Geometry Deformed on the GPU

Harlen C. Batagelo Wu, Shin-Ting*
State University of Campinas

Abstract

We present a simple yet effective snapping technique for constraining the motion of the cursor of an input device to the surface of 3D models whose geometry is arbitrarily deformed by a programmable hardware fragment and vertex processor. The technique works in image space and thus snaps the cursor to the geometry actually rendered instead of the geometry originally submitted to the rendering pipeline. We also present a method to establish a correspondence between snapped geometry in image space and object space, and an efficiency improvement based on the control of frequency of frame buffer accesses. Performance tests are conducted and compared against the standard picking and snapping algorithm used by the D3DX library of the Microsoft Direct3D API. We conclude by emphasizing the feasibility of our algorithm when facing the new advances of the graphics hardware for deforming geometry on the GPU.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques;

Keywords: constraints, direct manipulation, programmable graphics hardware

1 Introduction

Cursor snapping is a widely used direct manipulation technique that provides precise cursor positioning by constraining the motion of the associated input device towards a geometric element. Usually, it consists of restricting the motion of the cursor to the shape of geometries such as grids of points or lines by means of a rounding or gravity function that gives to the user the impression that the cursor is pulled towards these objects.

In desktop applications such as CAD/CAM (*Computer Aided Design/Manufacturing*) and games, it is common to the user want to pick 3D models (surfaces) rendered on screen and snap the cursor to the visible portions of them. In such snapping mode, the depth coordinate of the unprojected cursor is set to the depth of the sampled 3D surface pointed by the cursor's hotspot. This can be seen as a special picking mechanism that determines the current 3D positions of the cursor on the surface and the corresponding surface's normal vector.

Picking is a fundamental interaction task that consists of selecting a geometric element pointed by a 2D or 3D cursor [Foley et al. 1990]. For a 2D cursor, a traditional implementation of picking consists of calculating a *picking ray* defined from the current cursor's screen coordinates unprojected back to the 3D object space according to the inverse transformation of the world, view and projection matrices. The nearest object to the viewer that intersects the picking ray is selected. To extend this to surface snapping, the 3D cursor along the picking ray is simply moved to the object's surface at the intersection point and aligned to the surface's normal. Performing this continuously over the frames will produce the perception that the cursor is snapped to the surface.

*e-mail: {harlen,ting}@dca.fee.unicamp.br

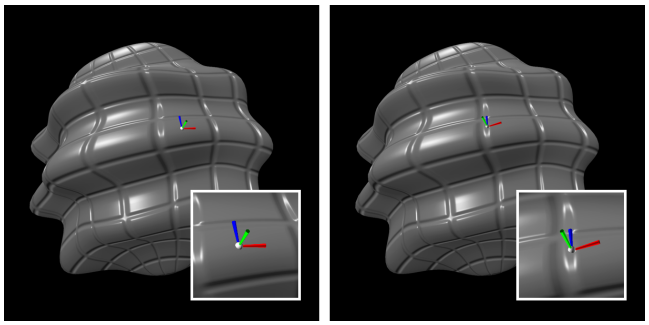

Figure 1: Snapping to a sphere deformed on the GPU. Left: on a portion with smooth variation in the normal; Right: on a portion with large variation in the normal due to bump mapping.

Nowadays, with the advent of programmable graphics hardware, the geometry submitted to the rendering pipeline can be arbitrarily modified by geometric transformations in a vertex processor running a user-defined *vertex shader* and further changed with respect to depth and normal in a per-fragment level by a *fragment shader*. The vertex shader can instruct the vertex processor to change the attributes of the stream of vertices that compose the original geometry, namely the position and the normal of the surface, thus producing a geometry that differs in shape with respect to the submitted model. The fragment shader, in its turn, can modify the attributes of each fragment, such as the depth and the normal vector, after the rasterization stage [Shreiner et al. 2003; Mic 2004].

This approach starts to be widely used in techniques such as terrain rendering, geometry blending and skinning, since it minimizes the processing bandwidth required to render the models. Unfortunately, the standard snapping algorithm based on the original geometry does not work satisfactorily with geometry deformed on graphics hardware, as the primitives snapped in object space may not correspond to the expected fragments rendered at the 2D screen pixel containing the hotspot of the input device cursor.

In this work, we propose a snapping algorithm in image space that can handle geometry arbitrarily modified in the rendering pipeline both by changes of the vertex position and direction of vertex normals in object space, and changes of depth and normal in a per-fragment level. Figure 1 shows two snapshots of the snapping result on a sphere whose points and normal vectors have been changed in the vertex shader with use of sine functions and whose rasterized data have been disturbed by the fragment shader to produce a bump mapping effect. Observe in the small squares (zoom in on the cursor

with a triad shape) that the hotspot (white ball) and the z-axis (green axis) of the cursor accompanies the geometry of the deformed surface. The point to which the cursor snaps is consistent to what the user is actually seeing. This is because our technique uses data read from the frame buffer, as explained further in Section 3.

For having correct visual feedback of the actions on the snapped point, it is sometimes necessary to know its corresponding point in the original geometry. We show in Section 3.1 how to determine, on scenes consisting of triangular faces, a correspondence between the snapped primitives in image space and the original primitives in object space. Besides the snapped point position and the surface's normal vector at this point in the object space, the proposed snapping algorithm is capable of returning which triangle the cursor snaps to along with the barycentric coordinates of the point relative to that triangle.

As our snapping algorithm read data from the frame buffer after flushing the rendering pipeline, CPU stalls happen, which may therefore slow down the overall rendering performance. In Section 3.2 we present an efficiency improvement by using a granularity time control so that the frame buffer does not need to be read for every frame. This decreases the number of stalls along the frames and maintains the sensation of a smooth interaction.

Performance test results of an implementation of the optimized and non-optimized algorithm are given in Section 4. In addition, they are compared against the results from the picking and snapping approach in software, as implemented in the D3DX library of the Microsoft Direct3D API [Mic 2004].

Finally, in Section 5 we discuss the reasons that make us think that the standard object space picking and snapping approach is not feasible for efficiently dealing with tasks related to deformation of geometry in graphics hardware, from the common per-fragment perturbation of the normals for bump mapping to cutting-edge features such as real-time displacement mapping.

2 Related Work

The snapping technique was traditionally used in CAD/CAM applications as a tool for positioning the input device cursor relatively to the position of markers, axis-aligned grids and geometry already created, thus helping the user to build accurate geometric models [Bier and Stone 1986; Bier 1990]. Nowadays, in applications such as virtual reality and games, cursor snapping may be used to give the user a better sense of the object's shape and dimension due to the additional depth and curvature cues it provides.

Unfortunately, snapping algorithms based on intersection tests performed in object space do not work satisfactorily on geometry changed in the programmable rendering pipeline since the application cannot trivially know how the geometry will be modified [Dir 2002; Dir 2004]. As a result, the snap point in object space usually does not correspond to the same point after the object space transformation by the vertex shader. Furthermore, on image-based rendering techniques such as the *relief texture mapping* [Oliveira et al. 2000], complex models may be rendered almost solely from a set of textures that are processed by a fragment shader and applied on a very simple object (*e.g.*, a cube) that does not keep any resemblance to the final apparent shape of the rendered object.

How can the user snap the cursor to the surfaces rendered on screen and determine the expected 3D position of the cursor in object space? The picking techniques available in the industry standard APIs are generally not suitable for snapping to geometry modified in the graphics hardware.

The Direct3D's D3DX library uses a picking procedure based on intersections tests between the picking ray and the geometry in object space stored in system memory [Mic 2004]. To consider the deformation, the application must transform the geometry using a software version of the deformation code of the shader and then perform the intersection test with the resulting geometry [Dir 2004]. Although this technique is quite accurate, it needs to transform the geometry twice when the cursor is snapping on a surface: one for the hardware version and other for the software version of the shader. This drawback can apparently be overcame by switching off the hardware version of the shader when the cursor is in a *snapping mode*, then using the software version for both snapping and rendering. However, this solution is a hindrance to the efficient usage of geometry deformation in hardware. If the snapping mode is always on, the geometry deformation in hardware will never be used. This strategy also does not work with geometry in which the depth or the normal of the fragments is changed in the fragment processor.

OpenGL provides a picking technique by using a special rendering mode called *selection mode* [Shreiner et al. 2003]. When rendering in such mode, the hardware is capable of returning identification data about the rendered models. Names are assigned to sets of primitives and, after rendering the primitives in the selection mode, the hardware returns a list of names corresponding to the geometry totally or partially contained in the view frustum, together with the minimum and maximum depth value of each set. On the basis of the *selection mode*, Wu *et al.* [Wu et al. 2003] proposed a novel snapping algorithm on the basis of two differential geometry properties: the position of the snapped point provided by the OpenGL API, and the normal vector of the picked surface at this point delivered by the application. Although the algorithm does not require the knowledge of the global representation of the surface, it still demands the application's intervention for completing a snapping action. It is because that the selection mode takes into account deformation of the geometry in hardware, but is not able to provide per-primitive data such as normal vectors and texture coordinates at the picking point.

3 Snapping Algorithm

Our snapping algorithm works in image space for determining the surface's depth and normal after the modification by the shaders in hardware, thus guaranteeing that the snapping will be performed exactly in what the user is seeing, with pixel level precision. It consists of rendering the scene onto an off-screen buffer in an additional rendering pass, or concurrently when working with a multiple rendering target, using a shader that encodes the geometry's normal and depth data in this off-screen buffer. The value of the pixel pointed by the 2D mouse cursor is read back to the application and used for computing the corresponding 3D snap point (position and normal) in object space.

We assume the user has previously selected a particular model of the scene for snapping. If this is not the case the model can be picked by rendering each object with a unique identifier encoded as a color value, then reading back the frame buffer at the pixel pointed by the 2D cursor and decoding the false color to obtain the identifier of the picked model [Shreiner et al. 2003]. As expected, the shaders used to do this must use the same deformation code used when rendering the actual models. Snapping on the selected object is then performed in the following steps:

1. *Render the scene as normal.* The scene is rendered in the frame buffer as usual.

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

Figure 2: False color visualization of the frame buffer data used for extended snapping. (a) Normal map; (b) Depth map; (c) Labeling of faces; (d) Barycentric coordinates.

2. *Isolate relevant pixel*. Since we need the surface data only at the pixel pointed by the 2D mouse cursor, the projection and viewport matrices are set up so that the view-frustum encompasses solely this pixel. This greatly decreases fragment processing.

3. *Render normal and depth map*. The normal vector and depth at the snap pixel are the fundamental frame buffer data needed by a snapping algorithm, as they are sufficient for computing the 3D position of the cursor and a tangent plane at the snap point [Wu et al. 2003]. In order to obtain the normals and depth values of the rendered primitives in a per-pixel basis, we render the scene onto an off-screen buffer using a normal and depth map shader. Although the depth value is already available in the frame buffer, the depth map may be needed because the availability of a reading operation from the depth buffer data depend on the hardware architecture and API (*e.g.*, Direct3D does not guarantee a readable depth buffer).

   The vertex shader must perform the same geometric transformations of the vertex shader used when rendering the geometry in the first pass. Likewise, the fragment shader must apply the same changes in depth and normal as the original fragment shader.

4. *Read off-screen buffer data*. After flushing the rendering pipeline, the off-screen buffer pixel pointed by the 2D cursor is read back and decoded for restoring the normal vector in object space and the depth value in perspective space. The corresponding normal vector in world space is computed by simply transforming the object space normal by the world transformation matrix. Instead, the user can directly encode the world space normals in the vertex shader.

   The 3D cursor's world position is computed by unprojecting the 2D cursor's location back to the world space according to the inverse transformation of the viewport matrix, then extending the resulting vector together with the depth value and transforming it by the inverse of the world, view and projection matrices.

3.1 Mapping Snapped Pixels to Primitives

In applications such as 3D painting and geometry trimming, the user may need to know which primitive the cursor lies on among the primitives that compose the model, or the affine coordinates in the plane of the snapped primitive such as the barycentric coordinates of the snapped point with respect to the triangle that contains it. In order to do so we present an *extended* snapping algorithm that include additional data in the frame buffer besides the data provided by the normal and depth map. This is explained in the following.

3.1.1 Identification of Primitives

For deciding which primitive on a triangular mesh contains the snap point, we assign for each triangle a unique identifier that is encoded as a color value in the frame buffer. In particular, the first vertex of each triangle includes the primitive's identifier as a vertex data that uses a color component semantic. The geometry is then rendered using a flat shading model that uses this false color of the triangle at the first vertex as the false color of the entire triangle. Finally, the 2D pixel containing the snap point is read back from the frame buffer and the color value is decoded for obtaining the identifier of the snapped triangle. Multisampling antialiasing is disabled when using this technique, otherwise invalid identifiers may be generated at the primitives' boundaries.

Normally the encoding of the identifiers should be done in an additional rendering pass besides the normal/depth map rendering due to the state change of the shading model. In order to integrate this with the normal/depth map rendering using the Gouraud interpolation model, the 3D mesh may be submitted to the rendering pipeline as a non-indexed triangle list. The non-indexed mode is required, otherwise the vertices adjacent to two or more triangles could not describe more than one primitive identifier.

3.1.2 Barycentric Coordinates

Besides the identification of the primitive where the snap point lies on, it is often useful to calculate the barycentric coordinates of the snap point with respect to the surface's triangle that it belongs. With such data it is possible to compute primitive's affine attributes that are not explicitly changed by the hardware shaders, such as texture coordinates for 3D painting. In fact, if the normal data is not changed in the vertex or fragment shader, an encoding of the barycentric coordinates can substitute the normal map rendering.

Our strategy consists of rendering a map of barycentric coordinates in a similar way as we render the normal map. Let us consider the barycentric coordinate pair (f,g) given by the equation $V_1 + f(V_2 - V_1) + g(V_3 - V_1)$, where V_1, V_2, V_3 are the triangle's vertices (triangle of reference). We encode f and g as two color components of each vertex and let them be linearly interpolated across the triangle. In special, we assign 1 to the first color component of V_2 (which represents f) and 1 to the second color component of V_3 (which represents g). The remaining color components are set to

null, including all color components of V_1. Finally, we read back the false color values from the frame buffer and decode them to obtain the (f,g) pair, where f is the value that controls the weight of V_2; g the weight of V_3, and $1-f-g$ the weight of V_1.

Similarly as in the identification of primitives, the rendering of the map of barycentric coordinates can be integrated with the normal and depth map rendering, assuming a frame buffer with higher precision that is capable of storing the required additional data.

In Figure 2 we show a visualization in false colors of the contents of the frame buffer for the extended snapping algorithm. In Figure 2(a) we show the normal map rendering with the X,Y, and Z components of the normal vectors in world space encoded in the $[0,1]$ range of the R, G, B color components, respectively. Figure 2(b) is the depth map and Figure 2(c) corresponds the identification of primitives by assigning a unique color for each face. Figure 2(d) presents the barycentric coordinates, with the vertices' weights of the second and third vertices of each triangle encoded as red and green intensities.

3.2 Optimization

Our basic snapping algorithm was intended to read the frame buffer for every frame, which could result in poor rendering performance due to the CPU stalls. Fortunately, the nominal frame rate required to maintain the sensation of interactivity of the snapping operation is a fraction of the actual real-time rendering frame rates. Therefore, we can increase the overall performance by simply decreasing the frequency of frame buffer accesses that are performed in sequence by the snapping algorithm as the 3D models are rendered.

The frequency may be controlled by the user according to the combination of the required degree of cursor positioning precision, the average magnitude of the displacement vector of the cursor and the frame-to-frame coherence of the snapped object. In our experiments we obtained reasonable results starting from a minimum of 20 Hz.

In order to maintain the smoothness of motion of the snapped cursor between frames of frame buffer accesses, we used a technique similar to that proposed by Wu *et al.* [Wu et al. 2003]. During the interval in which the frame buffer is not accessed, the 3D cursor snaps to the last computed tangent plane that corresponds to the last frame buffer access of a snap point. Though the last snap point computed by the image space snapping is always within the bounds of a triangle, the 3D cursor position may snap outside the triangle if the cursor displacement vector has a magnitude that is greater than the dimensions of the initial snapped primitive. However, the cursor lies always on the plane that contains the triangle, which helps to maintain the sense of interaction even on scenes rendered with very low frame rates.

4 Implementation and Tests

The algorithm was implemented in C++ using Direct3D and tested on a AMD Athlon 64 CPU with a NVIDIA GeForce FX 5900 XT GPU. The basic algorithm was first implemented using a frame buffer format of 8 bits per color component. However, due to the need of higher precision for the depth data on most scenes, we also implemented the algorithm using a floating point frame buffer of 32 bits per color component. We used this last format for all the tests. The improvements suggested in Section 3.1 were also integrated in this implementation. Thanks to this format, there was no need of using multiple additional rendering passes. In special, the orientation data was packed in the red color component using 8 bits for each vector component; the identifier of the primitives was packed in the green color component; the Barycentric coordinates in the blue component; the depth data in the alpha component.

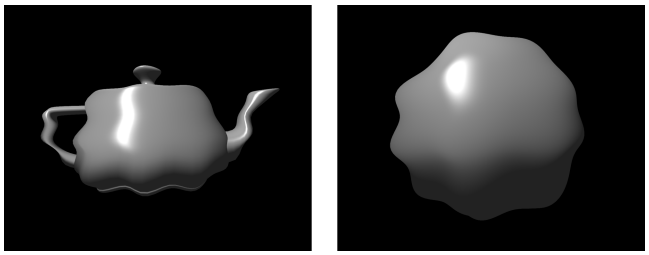

Figure 3: The test models: the deformed teapot and sphere.

scene	no snap	sw	hw	hw at 60 Hz
Low tess. teapot	2,86	3,60	3,22	2,90
Sphere	2,63	6,74	3,19	2,71
High tess. teapot	5,99	23,10	7,55	6,33

Table 1: Average rendering time (in ms) for the three test models using the basic snapping algorithm.

In order to determine the efficiency of the algorithm we compared it against the software picking technique used by the D3DX library of the Direct3D API. When using the D3DX library, geometry deformation was performed in software only. We did not use per-fragment deformation since the D3DX algorithm could not handle such case. The models used for testing were a teapot and a sphere deformed by sine functions over time and rendered as non-indexed triangle lists (Figure 3). The sphere was defined by 4,512 triangles and 2,307 vertices. The teapot was tested with a low tesselated version (2,256 triangles, 1,178 vertices) and a high tesselated version (16,256 triangles, 8,710 vertices). Although non-indexed triangle lists were mandatory for the extended snapping algorithm, indexed geometry rendered as triangle fans and strips can be used for the simple snapping algorithm, thus increasing performance.

In the first test we measured the performance of the basic snapping algorithm capable of returning a 3D position and its surface's normal only. This is usually sufficient for real-time applications such as games, as the user simply wants to select the models. The test results, measured as the total rendering time in milliseconds, are shown in Table 1. The column named "no snap" shows the rendering time without using any snapping algorithm but still using the deformation shader. The column "sw" contains the measurements obtained with the D3DX algorithm. The "hw" column show the rendering time obtained when using our algorithm without the optimization proposed in Section 3.2. The column named "hw at 60 Hz" shows the results when using such optimization with a snapping frequency of 60 Hz.

Our algorithm performed better than the D3DX algorithm for all models, even when not using the granularity control. The main bottleneck of the D3DX algorithm was the need to recalculate, for every frame, the model's vertex buffer in software based on the original geometry. On the other hand, CPU stalls were not an issue when using our algorithm. We think this is because the fragment processing was limited to only one pixel, being the remainder discarded in the clipping stage.

The second test used the extended snapping algorithm that returns the identification of the snapped faces along with the barycentric

scene	no snap	sw	hw	hw at 60 Hz
Low tess. teapot	2,86	3,60	3,28	2,93
Sphere	2,63	6,74	3,33	2,72
High tess. teapot	5,99	23,10	8,02	6,46

Table 2: Average rendering time (in ms) for the three test models using the extended snapping algorithm.

coordinates of the snap point. The results are shown in Table 2 using the same conventions of the previous table. The results for the "sw" and "no snap" modes are showed again for comparison. Performance dropped slightly in this case due to the additional computations performed in the GPU. Nonetheless, the results obtained with the hardware snapping are still better than the software version.

5 Conclusion

We presented a technique for snapping to geometry deformed on graphics hardware. The geometry can be affected by both changes in vertex position and orientation in the vertex shader, and per-fragment changes of normal and depth in the fragment shader. The technique works in image space and is thus capable of snapping the cursor to the geometry actually rendered instead of the geometry originally submitted to the rendering pipeline [1].

Firstly, we proposed a basic snapping algorithm that uses a normal and depth map rendering to calculate the 3D position and the corresponding surface's normal vector of the snapped cursor after the deformation of the model on the GPU. We also present an extended version of the algorithm that is capable of identifying which face the cursor snaps to, and the corresponding barycentric coordinates of the point of snapping. We further improve the efficiency of the algorithm by decreasing the frequency of frame buffer accesses and snapping the cursor to the tangent plane computed from the last access to keep the sensation of a smooth cursor motion.

According to the performance tests, the algorithm is more efficient than the standard snapping technique based on intersection tests. This is true even when the proposed optimization of decreasing the frequency of frame buffer accesses is not used. Our algorithm depends mainly on the processing power of the GPU and it is not negatively influenced by vertex data transfers between the CPU and the GPU as in the standard technique in software. Therefore, it can be used for real-time selection of deformed models in games (*e.g.*, skinned meshes, terrains, displacement mapped models), for interactive 3D painting and surface trimming. Figure 4 illustrates a potential application of our algorithm in 3D painting. In this case, it is a freehand drawing with the use of a mouse on a teapot deformed by the vertex shader.

We believe that, as the graphics hardware continue to evolve, the flexibility of the GPUs will allow users to increasingly take advantage of the deformation of geometry in hardware in such a way that it will soon become too difficult to efficiently emulate such transformation in software. In fact, performing tasks such as displacement mapping, deformation of higher-order primitives and per-fragment perturbation of normal vectors and depth in software would require the emulation of a significant part of the rendering pipeline on the CPU. The standard snapping algorithms in software are unfeasible in these cases, while a technique based on image space, as the one presented here, can handle them straightforwardly.

[1] Videos and sample code of the snapping technique are available at http://www.dca.fee.unicamp.br/projects/mtk/batagelo/index.html.

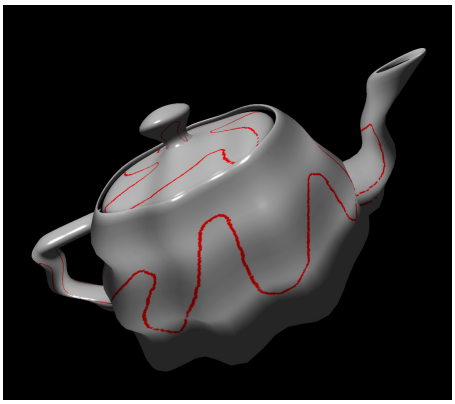

Figure 4: 3D painting on a deformed teapot using the extended snapping algorithm.

Acknowledgment

This project was supported by the National Research Council (CNPq) under the grant number 141685/2002-6 and The State of São Paulo Research Support Foundation (FAPESP) under the grant numbers 1996/0962-0 and 03/13090-6.

References

BIER, E. A., AND STONE, M. C. 1986. Snap-dragging. *ACM SIGGRAPH Computer Graphics 20*, 4, 233–240.

BIER, E. A. 1990. Snap-dragging in three dimensions. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, Snowbird, Utah, 193–204.

DIRECTX NEWSGROUP THREAD. 2002. *SkinnedMesh Pick Problem*, February. Internet newsgroup: microsoft.public.win32.programmer.directx.graphics.

DIRECTX NEWSGROUP THREAD. 2004. *Determining Which Skinning Method to Use*, May. Internet newsgroup: microsoft.public.win32.programmer.directx.graphics.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley Publishing Co., Reading, MA.

MICROSOFT CORPORATION. 2004. *DirectX 9.0 Programmer's Reference*, October.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *Proceedings of SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH, New Orleans, LA, ACM.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2003. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, 4th ed. Addison-Wesley Pub Co. ISBN 0321173481.

WU, S.-T., ABRANTES, M., TOST, D., AND BATAGELO, H. C. 2003. Picking and snapping for 3d input devices. In *Proceedings of SIBGRAPI 2003*, 140–147.