C Aspect de surface temps-réel - Utilisation du hardware graphique

- *Real-time Collision Detection for Virtual Surgery* (Computer Animation'99)

- *Perlin Textures in Real Time Using OpenGL* (RR-INRIA #3713)

- *Realistic Rendering of an Organ Surface in Real-Time for Laparoscopic Surgery Simulation* (The Visual Computer)

Real-time Collision Detection for Virtual Surgery

Jean-Christophe Lombardo,
EPIDAURE/SINUS, INRIA, 2004 route de Lucioles,
06192 Sophia Antipolis Cedex, France
Jean-Christophe.Lombardo@sophia.inria.fr

Marie-Paule Cani and Fabrice Neyret
iMAGIS†-GRAVIR / IMAG
BP 53, 38041 Grenoble cedex 09, France
Marie-Paule.Cani@imag.fr, Fabrice.Neyret@imag.fr

Abstract

We present a simple method for performing real-time collision detection in a virtual surgery environment. The method relies on the graphics hardware for testing the interpenetration between a virtual deformable organ and a rigid tool controlled by the user. The method enables to take into account the motion of the tool between two consecutive time steps. For our specific application, the new method runs about a hundred times faster than the well known oriented-bounding-boxes tree method [5].

Keywords: Collision detection, Virtual Reality, Physically-based simulation, Graphics hardware.

1. Introduction

Collision detection is considered as a major computational bottleneck of physically-based animation systems. The problem is still more difficult to solve when the simulated objects are non-convex and when they deform over time. This paper focuses on the specific case of collision detection for a surgery simulator aimed at training surgeons at minimally invasive techniques (ie. laparoscopy).

1.1. Virtual surgery

Non-invasive surgery is rapidly expending, since it greatly reduces operating time and morbidity. In particular, hepatic laparoscopy consists in introducing several tools and an optic fiber supporting a micro-camera through small openings cut into the patient's abdomen. The surgeon, who has to cut and to remove the pathologic regions of the liver, only visualizes the operation onto a screen. Learning to coordinate the motion of the tools in these conditions is a very difficult task. Figure 1 shows a typical tool used for laparoscopic surgery and a view of the control screen during an operation.

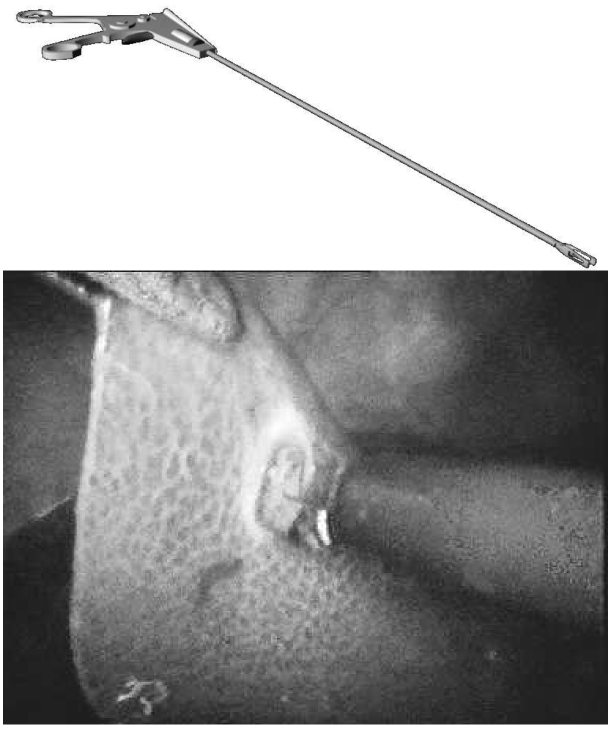

Figure 1. A minimally invasive surgery tool (top). View from the control screen (bottom).

The aim of surgery simulators is to offer a platform enabling the surgeons to practice on virtual patients, thus getting rid of financial and ethical problems risen by training on living animals or on cadavers.

Virtual surgery brings a number of difficulties: It requires both the abilities to interact in real-time with the virtual organs through a force-feedback device and to perform

†iMAGIS is a joint project of CNRS, INRIA, Institut National Polytechnique de Grenoble and Université Joseph Fourier.

a real-time visualization of the deformations. Moreover, the computed images should include as much visual realism as possible (texture of the organs, specular effects due to the optic fiber light, etc). In this context, the time that remains for performing collision detection at each simulation step is extremely small. The remainder of this paper focuses on this specific aspect of the problem. This work is a part of a wider project[1] that studies all the aspects of the problem, including real-time deformable models devoted to the physically-based simulation of the organs [3].

1.2. Collision detection techniques

Due to its wide range of applications, collision detection between geometric models have been studied for years in various fields such as CAD/CAM, manufacturing, robotics, simulation, and computer animation. The solutions vary according to the geometric representation of the colliding objects and to the type of query the algorithm should support. For instance, softwares that maintain the minimal Euclidean distance between the models are often required in motion planning application.

In our background of a surgery simulator, we are interested into methods that detect interpenetrations between polygonal models, since the latter are the most convenient for real-time rendering. We do not need to know the Euclidean distance between non-colliding objects. However, when a collision occurs, the precise knowledge of the intersection region is needed, since it will allow a precise computation of subsequent deformations and of response forces.

Most of the previous work in collision detection between polygonal models has focused on algorithms for convex polyhedra [1, 8]. These algorithms, based on specific data-structures for finding the closest features of a pair of polyhedra, exploit temporal and geometrical coherence during an animation. They are very efficient: the algorithm in [8] runs in roughly constant time even when the closest features change. However, they are not applicable in the case of a surgery simulator, since organs are generally non-convex, and deform over time.

Among the collision detection methods that are applicable to more general polygonal models [10, 2, 12, 4, 11, 13, 5, 7], almost all of the optimizations rely on a pre-computed hierarchy of bounding volumes. The solutions range from axis-aligned box trees, sphere trees [12, 11, 7], or BSP trees, to more specific data structures [2]. All these techniques, which perform very efficient rejection tests, may considerably slow down when objects are very close, ie. when the bounding volumes have multiple intersections. Among the recent approaches for finding bounding volumes that tighter fit the object's geometry, Gottschalk [5] obtains very good results by using hierarchies of oriented bounding boxes instead of axis-aligned boxes. Section 5 will compare our new method with the public domain software package *RAPID* that implements this technique.

A last issue in collision detection is the ability to perform *dynamic* rather than *static* detection [10, 4]: moving objects may interpenetrate between consecutive time steps, so the intersections should be computed between the 4D volumes that represent the solids' trajectories during a time step rather than between static instances of the solids. In the context of a large environment with lots of moving objects, using space-time bounds on the object's motion may lead to the quick rejection of a number of intersection tests [9, 6, 7].

In previous works on laparoscopic surgery [3], a dynamic collision detection was performed by testing for an intersection between the segment traversed by the tool extremity during a time step and the polygonal mesh representing the organ. A bucket data-structure discretizing the organ's bounding box, and storing local lists of polygons, was used to optimize this test. Real-time performances were obtained with a scene consisting in an organ and two tools, when no update of the bucket data-structure was needed. However, each tool was modeled as a single point, which resulted into possible penetrations of the body of the tools into the organ when an unexperimented user was trying to position them. Moreover, considering no update of the bucket structure was very restrictive concerning the possible deformations of the organ.

1.3. Overview

In the context of surgery simulation, the collision detection problem is enhanced by the non-convexity of most of the organs, and by the fact they deform over time. These deformations are far from negligible : laparoscopy typically involves large scale deformations and even topological changes in the structure of the liver since some parts are cut down and removed. In this context, spending time for pre-computing complex bounding volumes does not seem adequate, since this computation will need to be redone at each time step.

A second point is that, even if the number of colliding objects remains small (usually: an organ of interest and few surgical tools), objects usually stay in very close configurations. Collisions or contacts may occur at almost each time step, since the surgeon uses the tools to manipulate the virtual organ. Basically, whatever the method, an intersection test between each tool and the organ will be needed at each time step.

Thirdly, collisions need to be detected even during a fast motion of the tools, otherwise incorrect response forces would fed back to the user. So using dynamic detection, at least for the tools motion seems indispensable.

[1] http://www.inria.fr/epidaure/AISIM

Fortunately, the sum of features of the problem ease its resolution: only one of the objects (the organ) has a complex shape since the tools used in non-invasive surgery can be represented by thin and long cylinders (see Figure 1(a)). Moreover, the tools have a constrained motion since they enter into the patient's abdomen through small circular openings. These two properties enable us to take benefits of the graphics hardware for detecting collisions in real time.

The remainder of this paper develops as follows: Section 2 explains how the graphics hardware may bring a solution to our problem. Section 3 gives a method for performing static collision detection between a tool and the polygonal model of an organ. This method is extended in Section 4 in order to take the dynamic motion of the tool into account. Section 5 presents our results, including a numerical comparison of computational times with the public domain software *RAPID*.

2. Collision detection with the graphics hardware

Our aim is to find a real-time collision detection method that allows us to take the whole tool into account instead of just considering its extremity. Detecting a collision between two objects basically consists in testing if the volume of the first one (ie. the tool, which has quite a simple shape), intersects the second one. This process is very close to a scene visualization process: in the latter, the user specifies a viewing volume (or *frustum*), characterized by the location, orientation and projection of a camera; then, the first part of the visualization process consists in clipping all the scene polygons according to this frustum, in order to render only the intersection between the scene objects and the viewing volume. Specialized graphics hardware usually performs this very efficiently.

Thus, the basic idea of our method is to specify a viewing volume corresponding to the tool shape (or alternatively to the volume covered by the tool between two consecutive time steps). We use the hardware to "render" the main object (the organ) relatively to this "camera". If nothing is visible, then there is no collision. Otherwise we can get the part of the object that the tool intersects.

Several problems occur: firstly, the tool shape is not as simple as usual viewing volumes. Secondly, we don't want to get an image, but we need meaningful information instead. More precisely, we would like to know which object faces are involves in a collision, and at which coordinates. The *OpenGL* graphic library provides features that will allow us to model our problem in these terms. We review them in the next sections.

2.1. Viewing volumes

The most common frustum provided by *OpenGL* are those defined by an orthographic camera and by a perspective camera. In both cases, viewing volumes are hexahedra, respectively a box and a truncated pyramid, specified by six scalar values (see Figure 2).

Moreover, the user may add extra clipping planes for further restricting of the viewing volume, using `glClipPlane()`. All the versions of *OpenGL* can treat at least six extra planes, so the viewing volume can be set to a dodecahedron. However, we must keep in mind that efficiency decreases each time an extra clipping plane is added.

2.2. Picking

The regular visualization process is divided into a *geometrical* part and a *rasterization* part. The geometrical part converts all the coordinates of the scene polygons into the camera coordinate system, clips all the faces relatively to the viewing volume, and achieves the orthographic or the perspective projection in order to get screen coordinates. The rasterization part transforms the remaining 2D triangles into pixels, taking care of the depth by using a Z-buffer in addition to the color buffer.

Computing the first part of the process is sufficient for the applications that only require meaningful informations about visible parts of the scene. A typical example is the picking feature in 3D interaction: a 3D modeler needs to know which object or face is just below the mouse, in order to operate on it when the user clicks. If several objects project on the same pixel, it can be useful to know each of them. In 3D paint systems, the program rather needs to know the texture coordinate corresponding to the pixel which is below the mouse.

OpenGL provides two *picking* modes, that may be selected alternatively to the usual *rendering* mode `GL_RENDER` thanks to the function `glRenderMode()`. For these two modes, no rasterization is performed. Moreover, costly operations such as lighting are usually turned off. The picking modes differ from the informations they give back:

- the *select* mode `GL_SELECT` provides information about the visible groups of faces. A group name is given using `glPushName()` before each group of faces drawing, and *OpenGL* fills an array (provided by `glSelectBuffer()`) during the geometric pass of rendering, writing an entry per group that appears in the viewing volume. Thus one can know the faces that appear on screen. If the window has been reduced to a single pixel around the mouse, one gets the faces that appear below the mouse. If the camera geometry has been set in order to specify a given viewing volume,

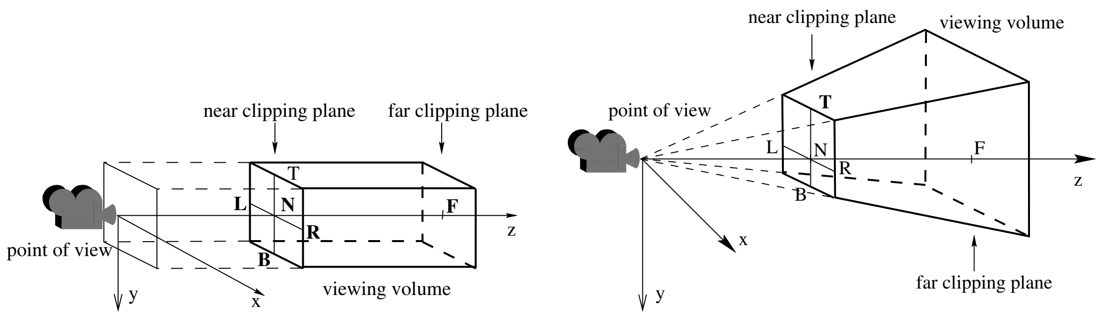

Figure 2. (a) The *OpenGL* orthographic camera (left) and the *OpenGL* perspective camera (right). The viewing volumes, which are either a box or a truncated pyramid, are characterized by the distances to the far and near clipping planes and by the two intervals [left,right] and [top,bottom] which define their section in the near clipping plane.

one gets the faces that intersect this viewing volume. Each entry contains some extra information, e.g. the z min and max inside the group, which can be used to sort or choose between multiple answers.

- the *feedback* mode `GL_FEEDBACK` provides extended information about the transformed and clipped scene. Basically, all the produced data can retrieved. The programmer indicates which kind of information he is interested in (positions, normals, colors, texture coordinates, ...), and provides an array with `glFeedbackBuffer()` that is filled by *OpenGL* during the geometrical rendering pass. In the same way that above, the scene may be clipped to a 1 pixel size window around the mouse, in order to get the geometric data corresponding to the mouse location. A naming mechanism similar to the previous one, using `glPassThrough()`, allows to get in addition the information of the faces (or groups of faces) numbers appearing in the viewing volume.

Since hardware is used to compute transformations and clipping, and since no rasterization is performed (which means that almost all interpolations are suppressed), both picking processes are particularly efficient.

3. Static Collision Detection

Laparoscopic surgery tools can be seen as cylinders of constant section s and of varying length, since user may pull or push them more or less widely into the patient's abdomen. In the remainder of the paper, we call $\mathbf{P_0}$ the fixed point where the axis of a tool starts ($\mathbf{P_0}$ is the center of the small opening the tool passes through), and $\mathbf{P}$ the extremity of the tool. Static collision detection between a tool and the polygonal mesh representing the organ can easily be performed by associating an orthographic camera to the tool.

The camera is positioned at point $\mathbf{P_0}$ and the viewing direction is set to $(\mathbf{P_0}, \mathbf{P})$, thanks to the function `gluLookAt()`. Near and far parameters are respectively set to 0 and to $\|\mathbf{P} - \mathbf{P_0}\|$. The tool section is taken into account by setting the left, right, top and bottom parameters of the camera according to the shape of the real tool extremity. The corresponding code is:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();

// compute distance between
// far and near clipping planes
l = norm(P-Po);

// push the orthographic camera on
// projection matrix stack
glOrtho(-s,s, -s,s, 0, l);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

// move the camera to set eye at Po
// and looking at P
gluLookAt(Po[0], Po[1], Po[2],
   P[0], P[1], P[2],
   up[0], up[2], up[1]);

// redraw the scene with some glNames
// pushed
redraw();
```

For our application, we simply want to detect which faces of the liver are in contact with the tool. Thus we use the *select* picking mode, with one different primitive name per liver face: each `glBegin(GL_TRIANGLES)` is preceded by `glLoadName(t)` where *t* is the triangle number. At the end of the rendering, the first row of the select array contains the number of hit triangles, then for each triangle items consisting in the z min and max and the face number. The exact coordinates of the intersection points could be obtained using the *feedback* mode.

4. Taking the motion of the tool into account

The simple solution presented in the previous section tests the collision between a static position of the tool and the organ at a given time step. This suffers from the classical flaws of time discretization: if the user hands move quickly, the tool may deeply penetrate inside the organ before being repulsed. It may even cross a thin part of the organ without any collision being detected.

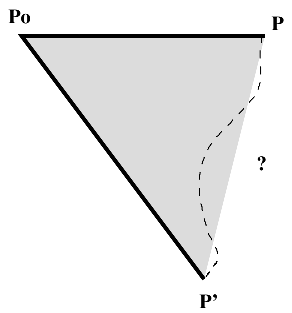

Figure 3. Tool movement between two simulation steps.

In order to avoid these classical problems, we present an extension which takes into account the volume covered by the tool during a time step (we still neglect the dynamic deformations of the organ during this time period). In our model, the tool goes through the patient's abdominal wall at the fixed point $\mathbf{P_0}$, and is able to slide through this point, so its length varies over time. We assume that the active extremity of the tool follows a straight line trajectory from $\mathbf{P}$ to $\mathbf{P}'$. The area covered by the axis of the tool is thus the triangle $\mathbf{P_0}, \mathbf{P}, \mathbf{P}'$ (see Figure 3). Since the tool may be seen as a cylinder of radius s, the volume covered by the tool during the time-interval is obtained by enlarging and thickening the triangle by the distance s. It is thus an hexahedron, as shown in Figure 4. As in the previous section, our aim is to model this volume using *OpenGL* cameras and clipping planes.

The simplest way to do this consists in using an orthographic projection, which parallelepipedic viewing volume corresponds to the bounding box of the hexahedron (see Figure 5): bottom and top, near and far correspond to the hexahedron; two extra clipping planes are used to model the sides $\mathbf{P_0P}$ and $\mathbf{P_0P}'$. However, this naive construction has some flaws. For instance, the orthographic viewing

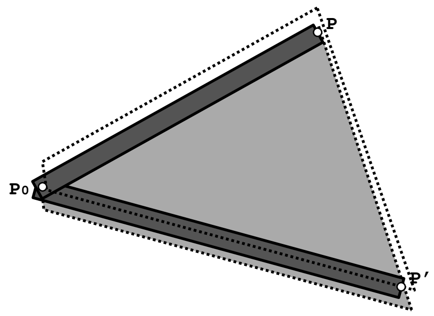

Figure 4. Volume covered by the tool during a time interval.

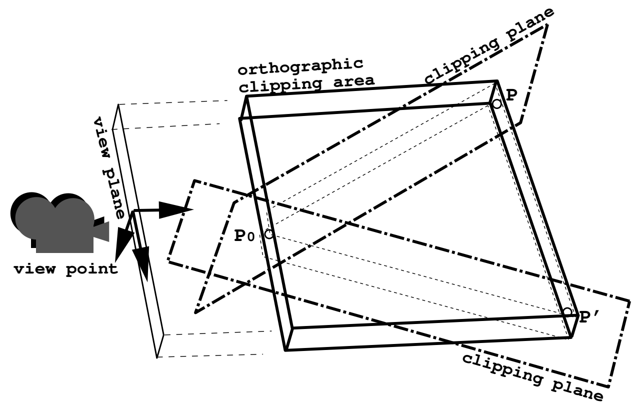

Figure 5. Naive approach using an orthographic camera.

volume will be excessively large when $\mathbf{PP}'$ is far from orthogonal to $\mathbf{P_0P}$ (see Figure 6). The consequence is that a lot of faces will be accepted during the clipping with the frustum, and rejected later during clipping with the user-defined clipping-planes. This increases the cost, since the latter is more computationally intensive than clipping with the canonical viewing volume.

Thus, we construct the test-volume using *OpenGL* in a more complex way, in order to use intermediary volumes that are as small as possible. Our construction is based on a perspective viewing volume whose cone follows the segments $\mathbf{P_0P}$ and $\mathbf{P_0P}'$, as shown in Figure 7. This is done by setting the camera axes to $\mathbf{PP}'$ for the x axis, $\mathbf{P_0P}' \times \mathbf{P_0P}$ for the y axis, and $\mathbf{PP}' \times (\mathbf{P_0P}' \times \mathbf{P_0P})$ for the z axis. As previously, the triangle is enlarged on each side by the tool section s. We set the $(top, bottom)$ interval in the near clipping plane to $2s$. Since the camera is a perspective camera, we have to add two extra clipping planes in order to limit the vertical extent of the volume to

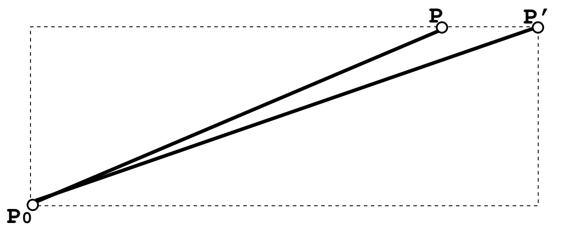

Figure 6. Configuration where the viewing volume is much too large before the addition of the two extra clipping planes.

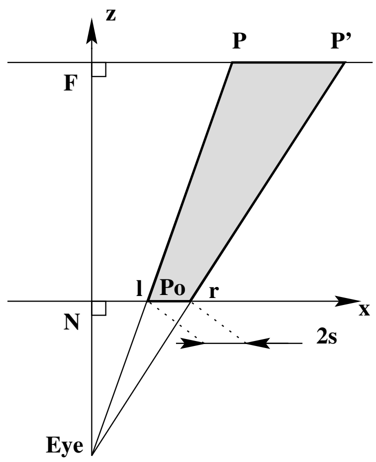

Figure 7. (x,z) plane of the perspective camera

$2s$ everywhere (see Figure 8).

To set the camera to this configuration, the eye position E must be computed from the points $\mathbf{P_0}, \mathbf{P}, \mathbf{P}'$. Let $\mathbf{u}$ be:

$$\mathbf{u} = \frac{\mathbf{PP}'}{\|\mathbf{PP}'\|}$$

We use it to set the left and right limits of the viewing volume in the near and far clipping planes:

$$\mathbf{P_{0l}} = \mathbf{P_0} - s\mathbf{u}$$

$$\mathbf{P_{0r}} = \mathbf{P_0} + s\mathbf{u}$$

$$\mathbf{P_l} = \mathbf{P} - s\mathbf{u}$$

$$\mathbf{P}'_{\mathbf{r}} = \mathbf{P}' + s\mathbf{u}$$

From Thales theorem we get:

$$\frac{\|\mathbf{EP_{0l}}\|}{\|\mathbf{EP_l}\|} = \frac{\|\mathbf{EP_{0r}}\|}{\|\mathbf{EP}'_{\mathbf{r}}\|} = \frac{\|\mathbf{P_{0l}P_{0r}}\|}{\|\mathbf{P_lP}'_{\mathbf{r}}\|}$$

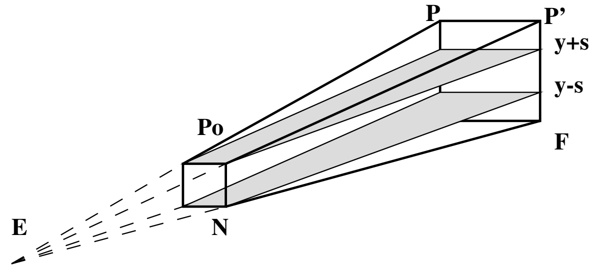

Figure 8. Reducing the viewing volume with clip planes

This yields:

$$\mathbf{E} = \mathbf{P_{0l}} - \frac{\|\mathbf{P_{0l}P_{0r}}\|}{\|\mathbf{P_lP}'_{\mathbf{r}}\| - \|\mathbf{P_{0l}P_{0r}}\|}\mathbf{P_{0l}P_l}$$

Thus we set the *OpenGL* perspective camera parameters to:

$$L = \mathbf{EP_{0l}} \cdot \mathbf{u}$$

$$R = L + 2s$$

$$N = \|\mathbf{EP_{0l}} - L\mathbf{u}\|$$

$$F = \|\mathbf{EP_l} - (\mathbf{EP_l} \cdot \mathbf{u})\mathbf{u}\|$$

$$T = +s$$

$$B = -s$$

We finally add the two extra clipping planes $y = -s$ and $y = s$ depicted in Figure 8. This leads to the following pseudo-code, where `fixed` is $\mathbf{P}_0$, `oldPos` is $\mathbf{P}$, and `newPos` is $\mathbf{P}'$:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
u = (newPos - oldPos)
   /norm(newPos-oldPos);
P0l = fixed - s*u; P0r = fixed + s*u;
Pl = oldPos - s*u; Pr = newPos + s*u;
E = P0l;
E -= norm(P0l - P0r)
   /(norm(Pl - Pr) - 2*s) * (Pl - P0l);
L = dot(P0l-E, u);  R =  L+2*s;
B = -s; T = s;
near = norm (P0l-E - L*u);
far = norm (Pl-E - dot(Pl-E,u)*u);

// define the projection
glFrustum(L, R, B, T, near, far);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

// clipping planes have to be placed in
// MODELVIEW matrix, but we define them
```

```
// if camera referential, so define them
// BEFORE gluLookAt()
GLdouble plan1[4] = {0,1,0,s};
GLdouble plan2[4] = {0,-1,0,s};
glClipPlane(GL_CLIP_PLANE0, plan1);
glClipPlane(GL_CLIP_PLANE1, plan2);
up = cross(E-Pr, E-Pl);
F = (Pl - dot(Pl-E, u)*u);

// move the camera to set eye at E
// and looking at F, with up set up[]
gluLookAt(E[0], E[1], E[2],
   F[0], F[1], F[2],
   up[0], up[1], up[2]);

// activate the clipping planes
glEnable(GL_CLIP_PLANE0);
glEnable(GL_CLIP_PLANE1);

// redraw the scene with some glNames
// pushed
redraw(NULL);
glDisable(GL_CLIP_PLANE0);
glDisable(GL_CLIP_PLANE1);
```

5. Results

We have done a series of cross-tests to bench our collision methods:

- using our liver geometry (1224 triangles) or a simple tetrahedron (4 triangles),
- testing either static collisions with the tool at a time step ('*static*') or collision with the volume covered by the tool during a time interval ('*dynamic*'), as depicted in Figures 9 and 10,
- testing dynamic collision with different numbers of colliding faces (between 5 and 25 for the liver, between 0 and 3 for the tetrahedron).
- comparing our method with the reference software package *RAPID*[2] implementing Obb trees [5],
- running on various hardwares and graphic accelerators.

Figure 11 sums up the comparisons of computational times between our method and the *RAPID* software on various platforms (each given time is a mean value between ten trials of different collision configurations). Since the same compiler (gcc/egcs) was used on all platforms for compatibility reasons, the results cannot be used for a direct comparison between platforms (gcc uses to produce inefficient code on SGI). The meaningful comparison is the ratio between the two methods depending on the graphics and computational performances of the platform[3].

The Obb tree method used in *RAPID* needs precomputing the hierarchical data structure. In our application where the liver deforms over time, RAPID's data-structure would have to be updated at each time step. Since there is no method for doing so to the authors knowledge, we compared our method with the use of RAPID where pre-computations are redone at each time step. Our method then brings an acceleration factor from 150 on high-end hardwares to 12 with a software implementation of *OpenGL* (however,Obb trees would probably give better results if an efficient update algorithm taking advantage of temporal coherence was developed). To be fair, we also computed the acceleration factor without taking RAPID's pre-computation into account. Even in this case which is only applicable to *rigid objects*, our method nearly brings an acceleration factor of five for each collision detection on high-end hardware. All these results are summarized in Figure 12.

6. Conclusion

We have presented a simple and very efficient method for detecting collisions between a general polygonal model and one or several cylindrical tools. Due to its impressive performances, the method is directly applicable in the context of a real time surgery simulator.

Since *no pre-computation* is required, our methods ideally fits to dynamic scenes where objects move and deform over time. As a comparison, the reference code *RAPID*, that is particularly fast, is five times slower assumed that pre-computations are already done, which is not possible for deformable bodies. Our method could thus be useful in many other applications, such as interactive sculpturing where the user manipulates a rigid tool for editing a 3D deformable shape.

The approach could also be generalized to be applied in more general collision configurations: here, one of the colliding objects has a simple geometry. In the general case with complicated shapes, our approach could be used to quickly test the collision between an objet and a non axis-parallel bounding box (or even a bounding dodecahedron) surrounding another object. If the second object is embedded into a hierarchy of bounding boxes, this idea could lead to an acceleration of the general Obb tree method. Lastly, since one of the objects can be a mere soup of polygons changing over time, the method could be applied to the

[2]http://www.cs.unc.edu/ geom/OBB/OBBT.html

[3]Concerning our method, we can note that the relatively bad results on the 3Dfx may be due to the fact that this architecture is not pipelined. On pipelined architectures (Onyx and 4D60), the collision detection time is almost constant when the scene size varies from 4 to 1224 triangles.

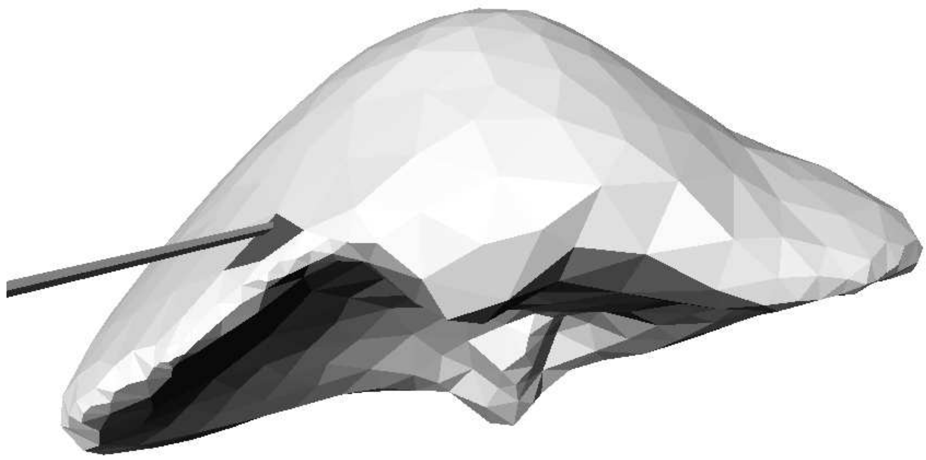

Figure 9. Collision detection between a triangular mesh modeling a human liver and a static position of a tool (which is visualized as a segment).

real-time collision detection between any deformable object (from an elastic surface or volume to a liquid substance) and rigid obstacles embedded into pre-computed hierarchies of bounding volumes.

Moreover, our method is extremely easy to implement (only few dozen lines of codes in an application using *OpenGL* for visualization), portable (*OpenGL* exists on most platforms) and benefits from different graphics hardware as constructors generally offer an optimized implementation of *OpenGL*.

References

[1] D. Baraff. Curved surfaces and coherence for non-penetrating rigid-body simulation. *Computer Graphics*, 24(4):19–28, Aug. 1990. Proceedings of SIGGRAPH'90.

[2] V. Bouma and G. Vanecek. Collision detection and analysis in a physically-based simulation. In *Second Eurographics Workshop on Animation and Simulation*, pages 191–203, Vienna, Austria, 1991.

[3] S. Cotin, H. Delingette, and N. Ayache. Real-time elastic deformations of soft tissues for surgery simulation. *IEEE Transactions on Visualization and Computer Graphics*, (in press), 1998.

[4] A. Garica-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 13(3):36–43, 1994.

[5] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Computer Graphics, Proceedings of SIGGRAPH'96*, pages 171–180, Aug. 1996. A public domain software package is available at : http://www.cs.unc.edu/ geom/OBB/OBBT.html.

[6] P. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[7] P. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996.

[8] M. Lin and J. Canny. Efficient collision detection for animation. In *Third Eurographics Worshop on Animation and Simulation*, Cambridge, England, Sept. 1992.

[9] M. Lin and D. Manocha. Fast interference detection between geometric models. *The Visual Computer*, 11(10):542–561, 1995.

[10] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, Aug. 1988. Proceedings of SIGGRAPH'88 (Atlanta, August 1988).

[11] I. Palmer and R. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[12] S. Quinlan. Efficient distance computation between non-convex objects. In *International Conference of Robotics and Automation*, pages 3324–3329, 1994.

[13] P. Volino, M. Courchesne, and N. M. Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. *Computer Graphics*, pages 137–144, Aug. 1995.

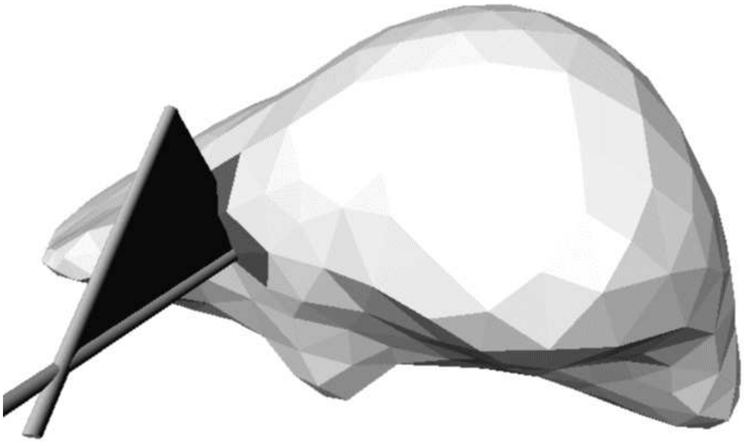

Figure 10. Dynamic collision detection, where the tool motion during a time interval is taken into account (this volume covered by the tool is visualized as a single triangle).

Using our *OpenGL* based method:

processor	R10000	DEC alpha	Pentium2	Pentium2
	195 MHz	500 MHz	333Mhz	333Mhz
graphic	Onyx2 IR	4D60	software	3Dfx Voodoo2
			(Linux Mesa)	(Linux Mesa)
static	0.13 ms	0.09 ms	2.2 ms	1.7 ms
dynamic	0.16 ms	0.11 ms	3.0 ms	2.3 ms

Using the Obb tree method:

processor	R10000	DEC alpha	Pentium2
	195 MHz	500 MHz	333Mhz
Precomputations	24.1 ms	15.7 ms	35.6 ms
static	0.63 ms	0.44 ms	1.0 ms
dynamic	0.76 ms	0.48 ms	1.2 ms

NB: *static* means considering a single position for the tool
dynamic means considering the tool positions during a time interval

Figure 11. Collision detection times

Acceleration factor	Deformable objects		Rigid objects	
	static	dynamic	static	dynamic
SGI Onyx	190	155	4.8	4.75
DEC alpha	179	147	4.9	4.4
Pentium (soft)	16.6	12.2	0.45	0.4
Pentium (3Dfx)	21.5	16	0.59	0.52

NB: *Deformable* objects means considering RAPID's precomputation time,
Rigid objects means igonring RAPID's precomputation time.

Figure 12. Acceleration factor provided by our method w.r.t. *RAPID*

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Perlin Textures in Real Time using OpenGL

Antoine Miné Fabrice Neyret
iMAGIS-IMAG, bat C
BP 53, 38041 Grenoble Cedex 9, FRANCE
Fabrice.Neyret@imag.fr

http://www-imagis.imag.fr/Membres/Fabrice.Neyret/

No 3713

juin 1999

THÈME 3

Rapport de recherche

ISSN 0249-6399

Perlin Textures in Real Time using OpenGL

Antoine Miné Fabrice Neyret
iMAGIS-IMAG, bat C
BP 53, 38041 Grenoble Cedex 9, FRANCE
Fabrice.Neyret@imag.fr
http://www-imagis.imag.fr/Membres/Fabrice.Neyret/

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet iMAGIS

Rapport de recherche n°3713 — juin 1999 — 18 pages

Abstract: Perlin's procedural solid textures provide for high quality rendering of surface appearance like marble, wood or rock. This method does not suffer many of the flaws that are associated with classical image mapped textures methods, such as distortion, memory size, bad continuity through objects. Being based on a per-pixel calculation, they were however limited up to now to non-real-time quality rendering as is ray-tracing. In this paper, we propose a way to implement Perlin texture using a real-time graphics library like OpenGL.

Key-words: image synthesis, virtual reality, procedural texture, Perlin noise.

(Résumé : tsvp)

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Textures de Perlin en temps-réel avec OpenGL

Résumé : Les textures procédurales pleines de Perlin permettent un rendu de qualité pour des surfaces comme le marbre, le bois ou la pierre. Cette méthode ne souffre pas de la plupart des problèmes que rencontrent les méthodes de plaquage de texture classique, comme les distortions, l'occupation mémoire, la mauvaise continuité d'une composante géométrique à l'autre. S'appuyant sur un calcul par pixel, elles étaient toutefois limitées jusqu'à maintenant aux rendus de qualité non temps-réel, comme le lancer de rayon. Dans ce papier, nous proposons une méthode pour qui permet d'implémenter les textures de Perlin en s'appuyant sur une librairie graphique temps-réel comme OpenGL.

Mots-clé : synthèse d'images, réalité virtuelle, textures procédurales, bruit de Perlin.

1 Introduction

Perlin's procedural solid textures are often used to generate complex looking surface appearance such as marble, wood or rock. They have numerous interesting properties, compared to classical image textures:

- They are computed in 3D, not on the surface, which avoids surface parameterization problems that usually produce large distortions on image mapped textures.
- For the same reason, texture features like a vein in marble can easily continue from one element of a composed object to the other, while using classical textures a mapping continuous and coherent through objects have to be found (which is uneasy).
- Almost no memory is used, as the texture values are computed on the fly.
- The resolution is adaptive, each iteration adds increasingly small details and can be stopped once the pixel size is reach. Having a classical image texture both covering a whole surface and having very fine details (to allows for close point of views) can need a lot of memory (e.g. $10,000 \times 10,000$ resolution).
- As they are procedural, no redundant design work is necessary, and no repetition appears. The artist rather controls high level parameters, such as the size of perturbation, the amount of turbulence, the kind of patterns, their size, orientation and location, the range of colors, etc.

However Perlin's textures are based on a per-pixel calculation, thus they cannot be computed in real time, but rather used in a realistic rendering algorithm like ray-tracing. As a consequence, real-time graphics library such as OpenGL only knows image mapped textures.

It would be very interesting to get the quality and ease of Perlin's textures, with the interactive rendering rate of real-time graphics libraries like OpenGL. This would allow for high quality images in a real-time application. This would also provide a large acceleration to non-real-time quality rendering.

Using the numerous features of extended OpenGL such as 3D texture coordinates, multipass, color matrix and look-up tables, we demonstrate in this paper that the Perlin's noise equation can be translated in terms of per-polygon mapped texture rendering.

2 Previous Work

2.1 Perlin's textures

Perlin introduced his model in 1985 [Per85]. Since then it has largely been used in all the existing high quality image synthesis platforms such as Maya, Explore, PowerAnimator or Softimage.

This model contains two ideas:

- It is a *procedural* texture model, which means that the value at a point results from an on the fly calculation.
- It is a *solid* texture model[1], which means that the appearance on the surface reveals data that are defined in volume, as if the object was sculpted in a block of material.

The procedural model is based on *turbulent noise*, that is a continuous self-similar function providing fractal looking patterns. This fractal noise $t(x)$ is defined in 1D as the fractal sum of a simple noise $s(x)$: $t(x) = \frac{\sum_0^n \frac{1}{2^i}.s(2^i.x)}{\sum_0^n \frac{1}{2^i}}$ A value of 4 for n generally gives good results, but one can let the sum add details up to the pixel size. Moreover, a lower value can be used to get smooth patterns. The $s(x)$ *noise function* is both continuous and random, and has a pseudo-period that can be controlled. It is built by interpolating smoothly random values defined on the nodes of a grid. Affine transformations of x allows for controlling the size of patterns (i.e. the frequency of the lowest component), and there location (i.e. the position of a pick in the values relative to a geometric feature). Formulas are identical in 3D, taking x as the vector $\vec{x} = (x, y, z)$. $s(\vec{x})$ is thus a function from $\mathbb{R}^3$ to $\mathbb{R}$, which smoothly interpolate the values given on a 3D grid.

In fact no 3D grid really need to be built, neither infinite array: hashing techniques [Arv90, EMP$^+$98] allows for the simulation of uncorrelated data using a simple small 1D grid of precomputed random values. The hashing of the indices greater than the array size allows for the generation of uncorrelated sequences: e.g.

[1] also called 3D textures, which should not be confused with *volumetric textures* [KK89] that really design 3D expanded objects.

the sequences $\{i\}_i, \{\alpha.i\}_i, \{\beta + i\}_i$ where the parameters α and β are large and generally primes, are uncorrelated once the indices are hashed. A permutation function $\sigma(i)$ of the indices allows for the production of uncorrelated components in multidimensional data. e.g. a 3D value on (i, j, k) is simulated using the 1D index $\sigma(i + \alpha_1 * \sigma(\alpha_2 + \alpha_3 * j + \alpha_4 * \sigma(\alpha_5 + \alpha_6 * k)))$, where the fixed parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6$ are big numbers (generally primes).

To be smooth, the interpolation used should be better than linear. Rather than using cubic interpolation, Perlin change the kind of data to be interpolated: instead of single values, he stores 4 values at the nodes of the 3D grid, that associate a plane to this node (i.e. a quadruplet of random values is used, representing a normal and a height). The resulting value of $s(\vec{x})$ is the trilinear interpolation of the distance of x to the plane defined at each on the nodes on the cell on which x lies.

The turbulent noise function is used as a seed or as a perturbation to give images. E.g. a colormap function can turn the values into colors, threshold functions can generate low or high plates in the curve. The $\vec{x}$ value indexing an image or a simple characteristic curve can be perturbed and turned into $\vec{x} + \alpha * t(\beta * \vec{x})$ were α controls the amplitude of the perturbation and β the frequency of its details[2]. Marble and wood are simulated that way, using a characteristic function that simulates an unperturbed vein for marble (a pick at the location of parallel planes, see figure 1) and for wood (a pick at the location of parallel circles, see figure 2).

2.2 Beyond Basic Graphics Libraries Features

Z-buffer based graphics libraries like OpenGL only implement a restricted set of geometric, photometric and textural representations[NDW93]. E.g. the shapes are only made of triangles, the photometric model is either Gouraud or limited Phong (computed at surface nodes then interpolated on the triangles). The textures are based on the mapping of images using (u, v) texture coordinates given on the surface nodes. These limitations are mainly due to the use of the Z-buffer algorithm, and to the constraints of using hardware (e.g. interpolating and normalizing a normal

[2]A perturbation has 3 components. $t(\vec{x})$ and $s(\vec{x})$ are then vector functions, i.e. they provides 3 uncorrelated noises in each of the 3 dimensions. So α and β can be a number, a diagonal matrix, or a full matrix if anisotropic perturbations are wanted.

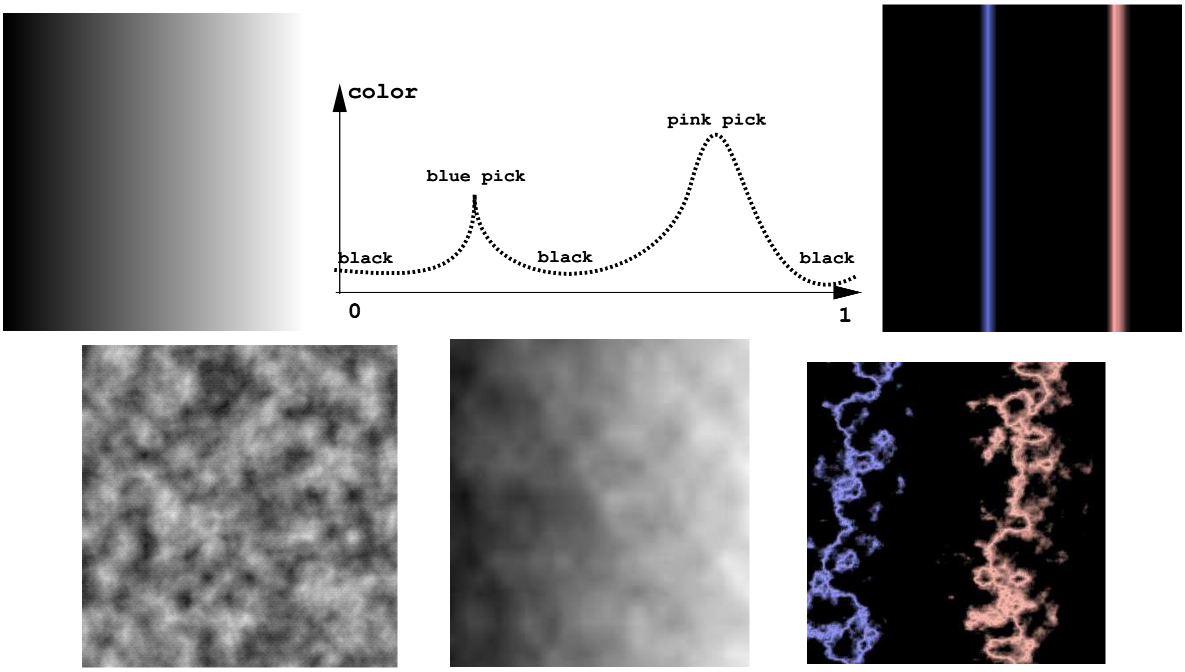

Figure 1: Marble texture: characteristic function $f()$, colormap $C()$, $C(f())$, turbulence $t()$, perturbation of $f()$ by $t()$ (i.e. $f(x+\alpha.t(x))$), result with the colormap (i.e. $C(f(x+\alpha.t(x)))$). NB: these images are produced in real time using our algorithm.

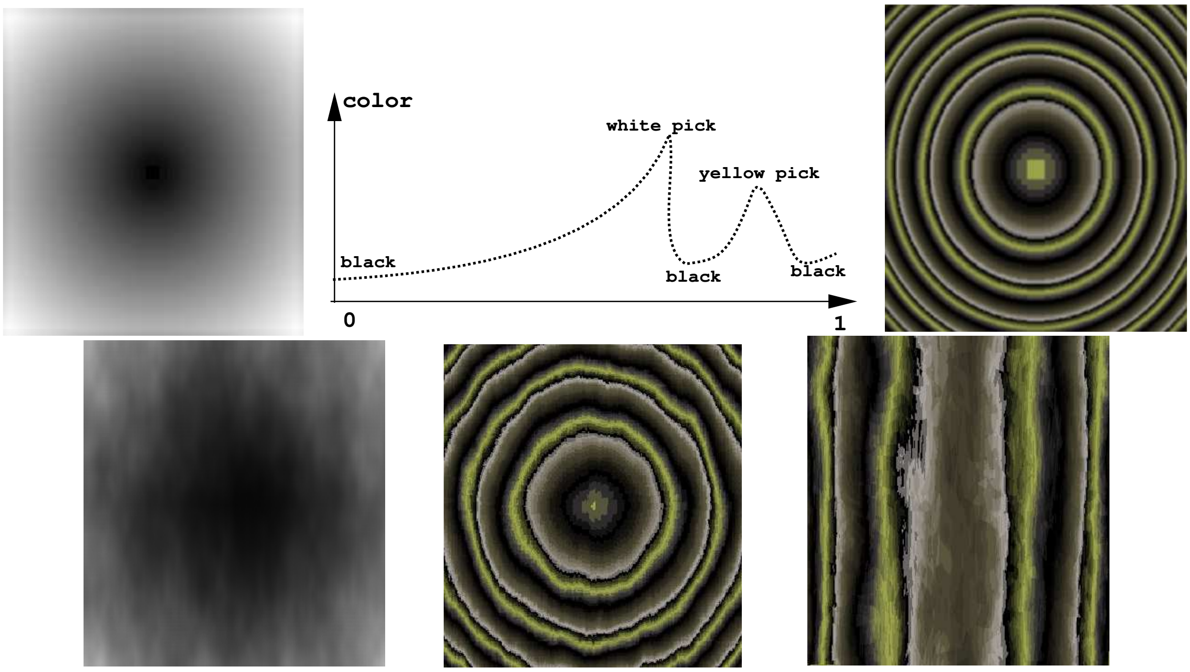

Figure 2: Wood texture: characteristic function $f()$, colormap $C()$, $C(f())$, perturbation of $f()$ by $t()$ (i.e. $f(x+\alpha.t(x))$), result with the colormap (i.e. $C(f(x+\alpha.t(x)))$), same from a side view. NB: this images are produced in real time using our algorithm.

vector along a triangle in order to implement a real Phong shading would need rather more complex electronics). However many open extra features allows for extensions, provided one knows how to translate a problem in terms of the limited grammar provided. E.g. textures coordinates can have from 1 to 4 dimensions, their value is freely determined by the programmer and a 4×4 texture matrix can transform them at rendering time. A 4×4 color matrix can be applied to a resulting color RGBA seen as a 4D vector, multiple color tables can transform the color or alpha value coming from pixels or from a texture, before or after color matrix multiplication, etc. Combinations of transparent layers can also differ for regular compositing by choosing other operators and coefficients than ones of the regular blending equation, and textures are not limited to defining a color: they can be mutiplicative (e.g. to simulate lighting), or contain a Z value.

An important point is to keep in mind that OpenGL ignores the *meaning* of operations and values, it simply *processes* them. Thus, interpolating 1 or 4 texture coordinates along a triangle is an equivalent process, indexing an array with 1 or 4 components is quite similar. It's the user and the programmer who give interpretation to what the texture is attached to, and what the image contents represents. E.g. environment reflections can be obtained by encoding in (u,v) the direction of a reflected ray at a given node, while for the rendering they are (u,v) like any others.

In particular, textures coordinates with more than 2 dimensions can be used in two ways:

- if a 4×4 texture matrix is provided, the coordinates are interpreted like regular coordinates, and transformed using the matrix (e.g. a projection). Then, one may consider the 2 first components of the result to index a texture image, the same way that the 2 first components of the 4D geometric and camera transform are considered to index a pixel on screen.
- by indexing a 3D table with 3 texture coordinates that are a linear transform of the node location, one can display a slice of a volume. The volume is encoded in the 3D texture, and the slice is given by the polygon location in geometric space and texture space. Note that rendering a surface using 3D texture coordinates gives exactly a solid texture as defined previously. Volume rendering can also be obtained by using a sequence of slices and transparent textures.

Using these OpenGL special features or extension, one can implement mirror reflections[NDW93], shadows, Fresnel lighting, bump mapping, volume rendering[WE98], volumetric textures[MN98], and many other effects usually only available in raytracing. Many of them are described in the Siggraph Advanced Graphics Courses [CR98] and on the SGI web site [Gra]. In particular, some clues on how to implement basic Perlin's textures are given in [CR98]: The idea is to define a small random 3D texture, and to map it several times while reducing the size by a factor of two, using the `GL_ADD` additive blending with no blending coefficient (i.e. 1 and 1 instead of $alpha$ and $1-alpha$). More functionality is necessary however beyond this basic solid texture in order to get a fully usable Perlin's texture. In particular, one needs to use this basic 'signal' as a perturbation function as explained before, in order to get the veins of marble or wood. This is described in the next section.

3 Perlin's textures using OpenGL

As we have seen in previous work, the more general Perlin texture equation that gives the color (or any other surface feature) at a given 3D location is modeled by:

$$C(f(T_1.(\vec{x}-\vec{x0})+T_2.\vec{t}(T_3.(\vec{x}-\vec{x1})))) \tag{1}$$

with $\vec{t}(\vec{x}) = \frac{\Sigma_0^n \frac{1}{2^i}.\vec{s}(2^i.\vec{x})}{\Sigma_0^n \frac{1}{2^i}}$ the turbulence function that produces the perturbation,
$f(\vec{x}) : \mathbb{R}^3 \rightarrow \mathbb{R}$ the characteristic function of the material,
$C(x) : \mathbb{R} \rightarrow \mathbb{R}^4$ is the colormap that gives an RGBA value,
$\vec{s}(\vec{x})$ is the pseudoperiodic noise function obtained by the interpolation of the random values given at the nodes of a (virtual) 3D grid.
$\vec{x0}$ and $\vec{x1}$ controls the translation of the characteristic pattern and of its perturbation,
T_1, T_2 and T_3 are matrices controlling the orientation and the directional size of the characteristic pattern and of its perturbation.

In order to simplify the computations, we decompose the transformation T_2 into a rotation R_2 and a scaling S_2, and factorize the R_2 rotation. We note α_1, α_2 and α_3 the diagonal coefficients of S_2. This gives the equivalent expression:

$$C(f(R_2.((R_2^{-1}.T_1).(\vec{x}-\vec{x0})+S_2.\vec{t}(T_3.(\vec{x}-\vec{x1}))))) \tag{2}$$

The point is now to translate this equation in terms of OpenGL (or any other rich graphics library) operations. In 3.1 we see how to generate the pseudoperiodic noise $\vec{s}(\vec{x})$, and in 3.2 how to build from it the turbulence $\vec{t}(\vec{x})$. We explain how to obtain a material such as marble or wood from that, by expressing the characteristic functions $f(\vec{x})$ in 3.3, and the color function in 3.4.

3.1 Generating the pseudoperiodic noise $\vec{s}(\vec{x})$

As suggested in the previous work section, we use a 3D texture containing random values to define the 3D grid. The interpolation to get the values at pixels lying between grid nodes is done by OpenGL, by selecting the magnifying filter `GL_LINEAR` . Despite the fact that it is less smooth than cubic interpolation, it gives correct results. In practice we use a $16 \times 16 \times 16$ random 3D texture. As on one hand we only need intensity values, and and the other hand we need uncorrelated values for the 3 dimensions (i.e. $s()$ is a vector), we will encode all along the process these 3 dimensions into the R,G,B channels. Thus, the 3D texture contains random RGB values.

3.2 Generating the turbulence $\vec{t}(\vec{x})$

The various scales of noise are added using multipass rendering: The (u, v, w) texture coordinates values at the nodes of the object to be rendered are initialized with the translated rotated and scaled geometric nodes coordinates $T_3.(\vec{x} - \vec{x1})$. The current color is used to tune the scaling. It is initialized with α_1, α_2 and α_3 stored in R,G and B, divided by 2 to incorporate an approximation of the normalization by $\sum_0^n \frac{1}{2^i}$ (that is 2 for n infinite), then the object is rendered. To process the other passes, we multiply the texture coordinates by 2, we divide the current color by 2, and we render the scene again. For a correct addition to be done, we choose `GL_ADD` and $(1,1)$ for coefficients in the blend operation. The iteration is repeated as many times as required by the fractal depth n (usually around 4). A very efficient solution to hide the texture repetition consists in rotating the texture at each iteration. As the perturbation size is normally a fraction of the main pattern, the α are much less than 1, so that no overflow will occur.

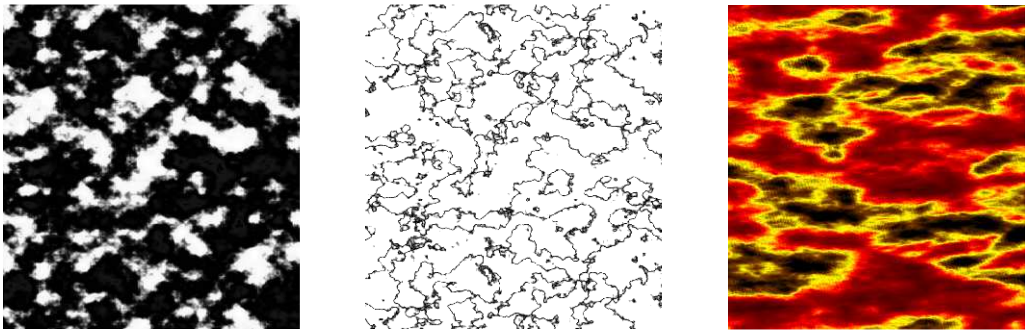

Figure 3: textures with no characteristic function, purely defined by $C(t(x))$.

3.3 Generating characteristic functions $f(\vec{x})$

At this stage, some kinds of textures like grainy rocks or clouds simply need to transform the turbulent noise into color using a colormap (see figure 3). For other materials, the texture is defined by the turbulent perturbation of a characteristic pattern. The perturbation is expressed by $\vec{x}+\vec{t}(\vec{x})$. The characteristic function is $f(\vec{x}) = \vec{x}[0].(1,1,1)$ for marble (see figure 1.1), as ideal features are vertical and parallel. It is $f(\vec{x}) = \sqrt{\vec{x}[0]^2+\vec{x}[2]^2}.(1,1,1)$ for wood (see figure 2.1), as ideal features are vertical concentric cylinders. From that point, we will only deal with these two generic examples.

We already have the perturbing term $\vec{t}()$ computed from the previous section. The evaluation of $f()$ first needs to add to $t()$ the displacement to be perturbed.

Perturbed displacement:

For this we use another 3D texture figuring the identity operator ID_{xyz}, i.e. having u,v,w stored in RGB at each texture pixel location (it is thus a 3D ramp). Then we render the object again, with the additive blend still enabled, after having initialized the (u,v,w) texture coordinates values at nodes with the translated rotated and scaled geometric nodes coordinates $R_2^{-1}.T_1.(\vec{x}-\vec{x0})$ (as required by equation 2. As identity is a separable function, one can also use 3 1D textures figuring ID_x, ID_y and ID_z (in fact it is the same, mapped using only one of u,v or w at a time). This avoids using a 3D texture, but this needs 3 rendering passes. This is useful for low-end graphics cards that do not implement 3D texture in hardware.

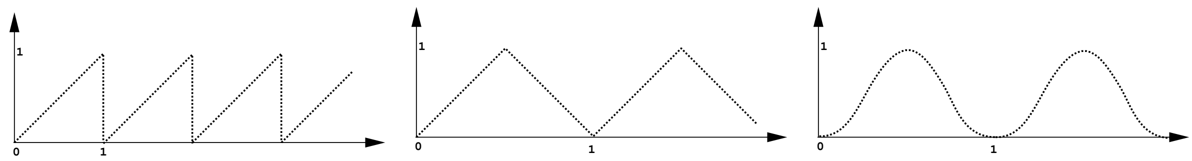

Figure 4: The identity function with modulo, in order to avoid overflow. Two other possibles functions avoiding the modulo discontinuity.

Of course we cannot map the identity in negative values and up to infinity. The function will have a saw tooth periodic shape, that is the same used by OpenGL to repeat texture tiles all along a surface. For some textures, it can be a problem to have the discontinuity due to the modulo. In such cases, we can use a triangular characteristic function instead. A sine can also be used. These ‘identity’ functions are represented on figure 4.

Overflow can occurs when adding the identity function and the perturbation. Since the last has a limited amplitude A that can be known, we weight the identity function so that it remains below $(1-A)$, i.e. we set the current color to $(1-A).(1,1,1)$.

Marble characteristic function:

As seen above, this function keeps only the $\vec{x}[0]$ component and copies it in the others components (we deal with frame transform in paragraph 3.3). This can be done by applying the transform matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

We use the color matrix, that multiplies RGBA pixels values seen as a vector before storing them in the frame buffer. As this should be done only once the iterative sum it computed, and not on the fly, we have to copy the frame buffer onto itself, in order to activate the color matrix transform. This copy should be limited to the object bounding box area.

To avoid touching the other objects on the screen during this operation on the frame buffer, we use the *stencils*: during the very first object rendering we set the stencil in order to mark pixels covered by the object, then during the copy pixels operations we enable the stencil test. The stencil will be reset to zero during the very last rendering pass.

It should be noted that as long as only the first component $\vec{x}[0]$ is kept, we can simplify the previous computations, replacing vector operations by scalar ones, and directly storing the same value in the 3 components RGB to avoid the final copy we need here. Thus we use a luminance 3D random texture, that OpenGL will understand as R=G=B, and a 1D identity luminance texture containing u at each texture pixel location. After rendering the scene, we directly obtain the result without having to use the color matrix and the frame buffer copy. Avoiding the matrix multiplication and the buffer copy a lot of time can be saved, especially on low-end graphics cards.

Wood characteristic function:

For the wood, we have to compute $\sqrt{\mathbf{x}[0]^2 + \mathbf{x}[2]^2}$. We have first to build the squares or the channels R and B. This can be done by copying the screen onto itself while using the blending in a very special way: we as a blend coefficient the image itself, which produces the squares (triangular identity function is used, to get the symmetry around (0,0,0)). Then we use the color matrix to sum the 2 squares and copy the result in all the components, with the matrix

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

This implies that the frame buffer must be copied again onto itself, as explained above (the blend operation being implemented in the pipeline after the color matrix transform, we alas cannot achieve these 2 operations in a single pass). The square root may be computed using a color map that would transform x into $\sqrt{x}$. This can be done in the same pass as the color matrix transform. As we already use a color map for the color, we will rather compile these two maps into one before rendering.

Including the rotation R_2

As the rotation R_2 has to be done before the evaluation of $f()$, as specified by equation 2, actually each of the noise components will be used. Rotation can simply be included in the color matrix, by multiplying this matrix by the rotation ([WE98] does a large amount of geometric transforms using the color matrix, including rotations, and even computes the Lambert shading with it).

To be noted that for most textures, the transform R_2 is the identity matrix (T_2 is only a diagonal scaling matrix): it is rare to want a preferred direction of distortion that is different from the preferred direction of texture 'grain' (i.e. frequency) encoded in T_3.

3.4 Generating of a material with $C(x)$ and $f(\vec{x})$

As seen in the previous work, the color map not only gives colors to the texture, but it really defines its main features, by selecting picks and plates, i.e. particular ranges of values. OpenGL provides for color maps, that can even be used to increase a texture resolution (the interpolation generates values between the texture pixels, which are individually considered in the colormap). We let the user define some key RGBA values that we interpolate to produce the map. In the wood case, we store a key designed for a value c at the location c^2 in order to take into account the $\sqrt{}$ transform. At rendering time the colormap is applied with a copy of the screen buffer onto itself, which can be done on the same pass as the previous color matrix transform.

4 Summary

Here is a summary of the algorithm, for the more complicated case that is the wood texture.

```
# --- Perlin noise pass
current color = (alpha1/2,alpha2/2,alpha3/2)
text coord at each node = T3.(x-x1)
set blend = ADD, coefs = 1,1
enable stencil set
iterate 1 to 4 times:
   render
   multiply text coord by 2
   divide current color by 2
# --- identity to be perturbed added
A = MAX(alpha1,alpha2,alpha3)
current color = (1-A,1-A,1-A)
text coord at each node = inv(R2).T1*(x-x0)
render
# --- prepare the squares
set blend = ADD, coefs = SOURCE,0
enable stencil test
copy frame buffer onto itself
# --- process c(f())
set color matrix to F_wood.R2
set the colormap
enable stencil reset
copy frame buffer onto itself
```

5 Results

As seen in the previous section, the whole process requires 2 to 5 rendering passes, depending on the noise frequency range required, and also 2 block copies in the frame buffer.

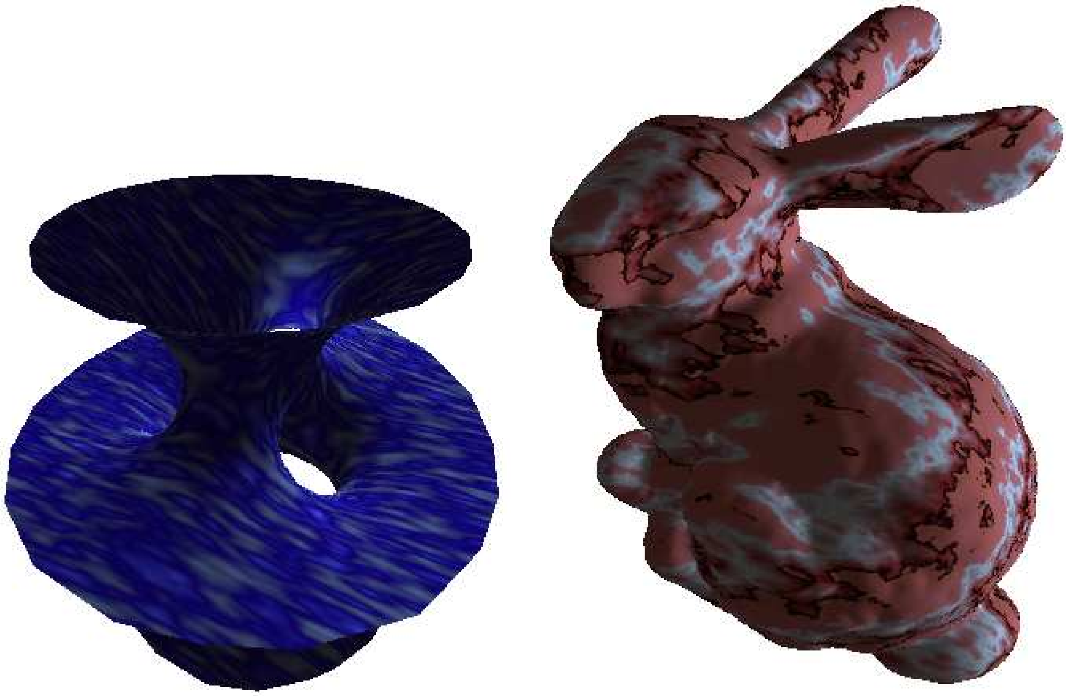

Figure 5: Left: minimal surface, 919 faces, about 35 frames per second. Right: the 'Bunny' big mesh, 70,000 faces, about 2 to 3 frames per second.

For the minimal surface in figure 5.1 containing about 900 faces, we get 30 to 40 images per second on Onyx2 Reality Engine. For the well-know decimation test object 'Bunny' on figure 5.2 containing about 70,000 faces, the rendering is no longer real time, but still interactive with a few images per second. A remind is that simplest scenes are ray-traced in at least ten minutes.

We have also test the program on a low-end architecture: The O2, that does not contain any hardware for 3D textures, color maps and color matrix. We replace the 3D texture by a 2D texture for the tests, which give sometime sufficient effects (otherwise it kills the frame rate !). The geometry on figure 5.1 is then obtained at 10 images per second. On O2 the rabbit already needs 1 second to render with ordinary rendering. With Perlin noise, it needs several seconds.

6 Conclusion and Future Work

In this paper, we have presented a complete solution to the synthesis of Perlin noise in real time on standard OpenGL meshes, using advanced OpenGL features. Although some of these features are not yet implemented in hardware on low-end graphics cards, they are simple and generic enough that they have a chance to appear soon on a wider market. This allows us to add a great amount of realism in real-time for interactive applications, and to greatly speed-up the procedural texture pass for realistic rendering. A library extending OpenGL has been developed from this work, that will soon be made publicly available.

As future work, we consider the translation in hardware of other procedural textures, such as Worley's textures[Wor96] that allows to produce nice cellular shapes such as scales or rocks, also limited for the moment to the context of non real-time quality rendering such as ray-tracing (being pixel-based). The principle of this technique consists in choosing random points on surface or in volume, and to consider nearest neighbor areas (i.e. Voronoï regions), to be combined with lower order nearest neighbors. The per-polygon translation of this, in order to be OpenGL compatible, would consist in using a texture containing a distance map represented by a concentric ramp, centered on each random point, and to use the `MINMAX` blend extension to keep only the min distance value.

References

[Arv90] J. Arvo. *Graphics Gems II*, chapter ,.10, page 396. Academic Press, 1990.

[CR98] Siggraph Course Notes CD-ROM. *Advanced Graphics Programming Techniques Using OpenGL*. Addison-Wesley, 1998. http://www.sgi.com/software/opengl/advanced98/notes/notes.html.

[EMP+98] Ebert, Musgrave, Peachey, Perlin, and Worley. *Texturing and Modeling, a Procedural Approach*, chapter 2, page 66. AP Professional, 1998.

[Gra] Silicon Graphics. Performer white papers. http://www.sgi.com/software/performer/whitepapers.html.

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23(3), pages 271–280, July 1989.

[MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, July 1998. Eurographics, Springer Wein. ISBN.

[NDW93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[Per85] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 169–178. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[Wor96] Steven P. Worley. A cellular texturing basis function. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 291–294. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399

Realistic Rendering of an Organ Surface in Real-Time for Laparoscopic Surgery Simulation

Raphaël Heiss, Fabrice Neyret and Franck Sénégas
e-mail: Fabrice.Neyret@imag.fr

iMAGIS[†] / GRAVIR-IMAG

contact address : Fabrice Neyret
iMAGIS - GRAVIR / IMAG
INRIA Rhône-Alpes ZIRST
655 avenue de l'Europe
38330 Montbonnot Saint Martin
FRANCE

[†]iMAGIS is a joint research project of CNRS / INRIA / UJF / INPG.

Abstract

This paper deals with the rendering issues of the problem of producing a real-time convincing surgery simulation. The main scope of our project is to simulate laparoscopic liver surgery. Nevertheless large parts of the technique apply to other organs. We address three aspects of the appearance of the organ surface: the organ skin texture, the specular highlights, and the reactions of the organ to the instruments.

For this last aspect we address three effects: blood drops rolling on the surface, clear or deep cauterization, and whitening of the surface under local pressure.
To meet the real-time constraint we use advanced graphics features such as multipass rendering, OpenGL texture extensions and lookup-tables. Our target hardware are high-end graphics accelerators such as the SGI Infinite Reality. Nevertheless most of the technique apply to low-end graphics accelerators.

Key words : Real-time rendering, multipass, simulator, medical applications, textures.

1 Introduction

Why simulating ? Laparoscopic surgery is a non-invasive technique that consists of introducing through small holes in the patient's abdomen several micro-instruments and an optic fiber connected to a camera and to a light source (see Fig. 1). This new technique is expanding, but it requires a lot of training for a physician to be proficient. Classical training sessions using cadavers or animals are costly, yield ethic problems, and cannot easily train for particular pathologies. The fact that laparoscopy surgery consists of looking at a monitor while manipulating the scissor-like end of instruments outside the patient should make it easier to replace the patient by a computer with force-feedback devices. There is thus a demand for laparoscopic simulators.

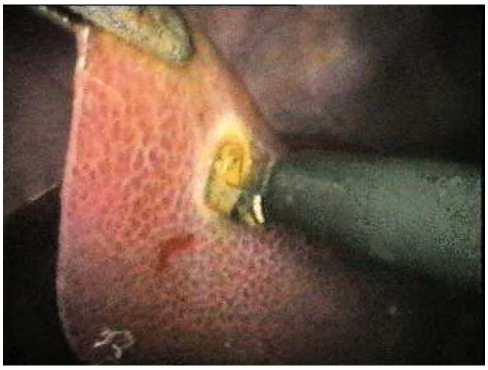

Fig. 1: A real liver laparoscopic image.

Simulating what ? Solving the problem of producing a real-time convincing simulation of the surgery of an organ requires that we address several issues:

- rendering in real-time the look of the organ surface, including the effects of the instruments and the reactions of the living organ;
- managing in real-time the collisions of the instruments with the organ;
- simulating the organ deformation in real-time.

This research is part of a larger project named AISIM (AISIM project). The last two issues are being addressed by other groups within the project. See for instance (Debunne et al. 1999; Debunne et al. 2000; Cotin et al. 2000) for the simulation of deformations and (Lombardo et al. 1999; Picinbono et al. 2000) for the collisions management. In this paper, we only address the first issue.

Why realistic rendering ? In the scope of simulation for training professionals, the realism of the rendering is not a purpose in and of. It can even be dangerous if it provides non-pertinent informations to the trainee that he might use later in real situations. For instance, vessels may be used consciously or not to locate a site, while the vessel locations vary from one patient to the another. The presence of realism addresses three purposes: quality of immersion (it is important that the trainee reasonably trusts the simulation), carrying pertinent information, and providing 3D information that is present in the real situation (no more, no less, same modality).

To find the origin of the 3D sense mentioned above is not trivial, even during actual surgery: the monitor does not carry stereoscopic information, the depth of field of the camera is large, there is almost no shadow because the light source is mounted on the camera, and the shading variations are weak because of high inter-reflections (i.e. the ambient component of the illumination is high).

However, physicians need this 3D information to operate! The fact is that early trainees are depth-blind, and progressively acquire this perception, often without understanding where it comes from. Thus, several secondary clues provide pieces of 3D information. Some deal with the image, some not (e.g. contact feeling, a-priori knowledge of shapes). The firsts include the depth cue given by perspective shrinking of the textured surfaces, the highlights, that provide curvature information, which are also very important for the feeling of contact with the surface, and very thin shadows between distant and close objects.

Because most of the operation focuses on the main organ, we only deal with the first and the second issues.

Rendering what ? For providing 3D clues, we will map undistorted textures on the surface, and simulate the highlights due to the light source, which is a thin ring around the optic fiber.

The pertinent information to carry is the effects of the instruments on the surface and the live organ reactions, which include blood drops rolling on the surface, clear or deep cauterization, and whitening of the surface under local pressure.

To achieve high immersion quality, we will have to take care of realism when rendering these features. As the purpose is to obtain a simulator, all these have to be done in real-time. This suggests that we avoid increasing the geometry (e.g. for the blood drops), and use the advanced graphics features available on the graphics accelerator.

2 Previous Work

2.1 Texturing

Some surgery simulators show organs with constant color, or define colors only at the mesh vertices. The best aspect of organs is obtained by ‘dressing’ the mesh with textures.

Textures have been around for a long time in Computer Graphics (Catmull 1974). However, mapped textures have suffered from mapping distortion for a long time, despite various improvements. This constrains the designers to pre-distort the texture drawing in order to compensate the distortion during mapping. Several solutions to this problem have been proposed:

Direct painting (Hanrahan and Haeberli 1990) consists of painting the texture directly on the final surface and storing the color values in texture space. As the mapping distortion during the storage is the reverse of the one at rendering time, no distortion results (provided the mapping is bijective).

Undistorted pattern mapping using triangular tiles (Neyret and Cani 1999) solves the mapping problem for classes of textures that meet our requirements (isotropic and homogeneous). It consists of defining a small set (four) of compatible equilateral triangular patterns, to be associated to triangular areas defined by a regular triangulation of the surface. Mapping with tiles is interesting because it allows high image resolution using little texture memory, and it relies on samples of materials that are independent from objects shape.

Procedural solid textures, such as the ones introduced by Perlin (Perlin 1985) and Worley (Worley 1996), require no mapping. The second one produces textures made of cells, being based on the 3D Voronoï diagram of a Poisson distribution of points. This matches well the properties of biological tissues.

We combine somehow these three solutions for the various effects we want to obtain. We extend the first one to simulate the evolving effects. We rely on Worley

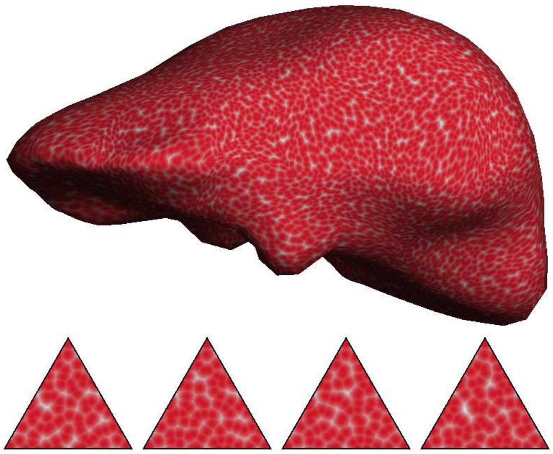

Fig. 2: *Top*: the liver texturing with minimal distortion. *Bottom:* the four triangular tiles used.

textures for the skin aspect. As we cannot afford a per-pixel procedural evaluation in real-time, we will precompute the texture. Fortunately, this has already been addressed for triangular tiles in (Neyret and Cani 1999) (see Figs. 2), thus we will base the skin rendering on this method.

Note that some work has been done on automatic texture acquisition. This may be used to obtain the texture of an organ from a set of photographs. However this reverse problem is especially hard in our case, as both the texture, the exact organ shape, the illumination conditions and the camera location relatively to the organ frame are unknown at the same time. Since satisfactory procedural textures can be produced as well, with the ability of getting compatible small patterns, we tend to prefer the easy solution.

2.2 Highlights

Reproducing the reflects of the light source is important in surgery simulation because they are a clue for depth, orientation and deformations. Organs are wet, and generally covered with a transparent tissue, thus being highly specular.

Specular highlights are generally smaller than the faces of the geometric mesh. Solutions based on Phong evaluation at the vertices thus yield a lot of specular aliasing (i.e. hexagonal spots, flickering shading). Moreover, Phong evaluation only considers a point light source. A convenient solution to these problems is to use reflection mapping (Miller and Hoffman 1984), to allow for reflections of the environment on the object. This fix the highlight resolution problem, and this allows for complex light source. Environment textures are available in hardware using regular textures. It consists of using for (u, v) texture coordinates at each vertex the parametric location on the environment sphere that reflects at this vertex. Various improvements have been introduced to release the limitations of the model (i.e. view dependency, distant environment, motionless environment). For our application the simple solution is almost sufficient, as the environment texture simply represents the aspect of the specular spot, whose location is fixed and centered in the texture since the light and the view-point are merged. However, during surgery the light is not at infinity and its distance change, thus the spot aspect will have to evolve with the distance of the light. Alas, transferring a texture to a graphics board is slow, and doing it at each time step kills performances. We introduce a variant of this technique in order to process distance changes efficiently, and to take the surface roughness into account.

2.3 Advanced rendering features

Graphics libraries such as OpenGL (Neider et al. 1993) provide interesting features (hardware-accelerated on high-end graphics boards) allowing more and quicker effects. For instance, at that time we do not simulate other rendering aspects such as shadows or rough surface. If necessary, solutions could be found using shadow-maps and bump-mapping that can be obtained using advanced rendering features (Siggraph Course Notes 1998; Silicon Graphics b; Silicon Graphics a; Nvidia).

The features we use to get our results and to match the real-time constraints are:

- multipass rendering, that allows us to add several layers of appearance on a surface;
- feedback rendering, which is the classical way to know what location of the geom-

etry is below the pointer;

- texture updating, that allows us to transfer only the part of the texture that has changed from one frame to the other;
- reflection mapping, as stated above;
- color look-up tables, allowing variations of the texture appearance without having to transfer the texture more than once.

2.4 Drops, burnt spots and whitened spots

Little work has been done on drops. Drops used in CG applications are often based on particle systems (Reeves 1983; Reeves and Blau 1985), associating a fake visible object (a disk, a 2D sprite, a line segment) with a physically animated point object. Tears have been simulated in (Fournier et al. 1998). We do not need so much realism in the visual aspect, and we don't want to add hundreds of polygons in our scene, but we need the same kind of animation depending on gravity, shape curvature and friction. For the other reactions of the organ, i.e. burning under cauterization and whitening under local pressure, no real motion occurs despite the evolving appearance, so no physical particle animation is needed. We will achieve the 'fake rendering' of all these objects by directly painting them in the texture at their simulated location, in a way analogous to (Hanrahan and Haeberli 1990), excepted that the painting is temporary.

3 Contributions

3.1 Our model

Our organ surface model consists of three layers to handle the three components of appearance, corresponding to three textures and to three rendering passes (see Fig. 3):

- the first one accounts for the organ skin appearance, including skin color, Lambert shading and skin texture. We rely on Worley patterns and the regular triangular tiles

Fig. 3: The three texture layers: skin, reactions and reflects.

of (Neyret and Cani 1999), as illustrated on Fig. 2. We describe this in section 4.1.

- The second one deals with evolving reactions of the organ skin. These reactions are drawn dynamically in a transparent texture. We do this by extending the direct painting method (Hanrahan and Haeberli 1990), replacing the artist's brush by the simulation of the effects (blood drops, burning, whitening), getting the aspect shown on Fig. 4. The main issues to be solved are listed in sections 3.2 and 3.3; we detail the rendering in section 4.2 and the animation in section 5.
- The third one manages the highlights, using reflection mapping (Miller and Hoffman 1984), that we extended to take into account the variations of the light distance (see Figs. 6). We introduce the issues in section 3.4 and we detail our technique in section 4.3.

We give a pseudo-algorithm of the whole process in Appendix A.

3.2 Suppressing distortions

Distortions are avoided for the skin layer using the triangular tiles method. The reflects layer suffers no distortion, as the parametric space is based on polar coordinates, like the light source shape is. Thus we only have to deal with distortions of the reactions layer, which consists of a regular image texture.

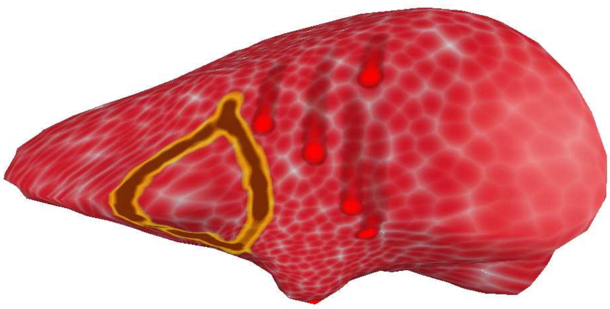

Fig. 4: The three kinds of dynamics effects: cauterization (left), blood drops (middle) and whitening (right).

We build dynamically this image on a way inspired by the direct painting method. But differently to the original method (Hanrahan and Haeberli 1990), we are drawing 'brushes' (i.e. 2D sprites) in the texture, and not simply point painting. Thus we have to take into account the mapping distortion in the neighborhood of the drawing location. This is achieved by considering the local Jacobian of the mapping: for a disk spot to appear on a location of the surface, an ellipse has to be drawn at the corresponding texture location. Details are given in section 4.2.3.

3.3 Evolving effects

The evolving effects are local (drops, spots, marks), and can be though as a kernel around an active spot (drop center, current pressure or heat site) followed by the fading 'memory' of the effect along the previous locations of the active spot. The kernel aspect, the fading nature and speed define the appearance of the effect, while the animation of the active spot defines its change with time.

We model these effects using snakes of sprites, much like the early computer game *Worms* in the eighties, except that our sprites are living in the texture space instead of the screen space (see Fig. 5). We developed an update strategy to minimize the texture transfer to the graphics board, knowing that only the sprite areas change from one frame to the other, and that the sprites overlap. The animation of these snakes is managed by

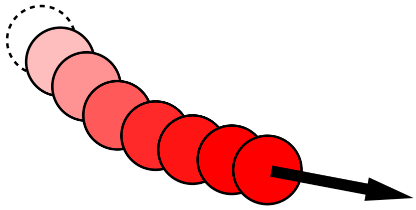

Fig. 5: An effect is represented by a snake of sprites in the texture space.

an automaton which handles the creation of new active spots, the fading of the old ones, the destruction of inactive spots, and collisions rules. Collisions (for drops) are handled using a list for each face that keeps track of the sprites present on the face (storing a sprite list per face is not memory consuming as there are both few faces and few sprites in the simulation). This mechanism also allows us to estimate the decrease of friction for a drop following another one (which will thus finally merge with it). The overlapping (i.e. collisions) between burnt paths should result in a strengthening of the burning. This is achieved by introducing an indirection: we keep in main memory a map of burning intensity, which is transformed into color using a lookup table.

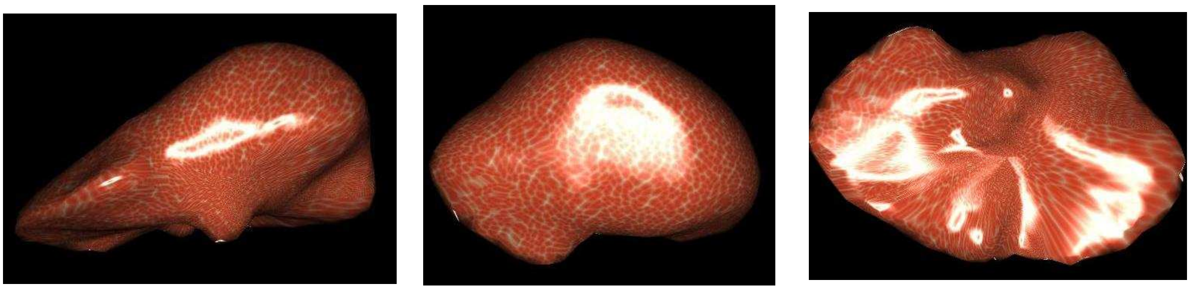

Fig. 6: Ring light source glossy reflection.

3.4 Specular effects

Specular spots on surfaces result from the combined effects of the Snell-Descartes perfect mirror reflection and the distribution of orientations that a rough surface has. We handle these two aspects separately in order to allow the tuning of light distance and surface roughness. We rely on environment textures, than can be hardware accelerated on any graphics board handling textures. As the highlight spot is axi-symmetric, we represent it with the composition of a constant radial ramp by the 1D profile of the reflect spot (see Figs. 14 and 15). This indirection allows us to only update the 1D table when the reflect aspect should change, instead of recomputing and transferring the whole 2D map. The 1D indirection table is implemented using hardware look-up tables. On hardware featuring this facility (such as the SGI Infinite Reality, or PC boards handling palette textures), this allows the real-time update of the highlight appearance.

4 The Three Texture Layers Liver Model

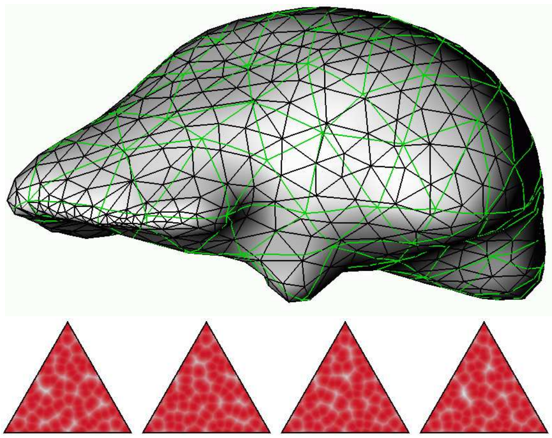

Fig. 7: *Top*: the texture mesh (in green) added to the geometric mesh. The extra vertices are handled as barycentric combination of the main vertices. *Bottom*: the triangular tiles to be mapped on the green mesh.

4.1 Skin Map

For this first layer, we apply the triangular patterns mapping method (Neyret and Cani 1999). The main difficulty lies in the fact that extra vertices are introduced in the geometry, that are the vertices of the texture mesh (figured in green on Fig. 7). This can yield a problem with the simulation of the organ deformation, as these extra vertices

are not known by the deformation model and don't correspond to 3D elements inside the organ volume. Propagating the creation of surface vertices into the volumetric deformation model would increase the model complexity and the resolution time, for a reason unconnected to the simulation quality criteria, which is unacceptable. Instead, we consider these extra vertices as secondary vertices, defined at any time step as the barycentric combination of two or three primary vertices.
For the Worley (triangular) texture, we use only the closest neighbors (i.e. the regular Voronoï diagram of a Poisson distribution of points), and a simple color map to turn the distance into colors (see Fig. 7, Bottom).

4.2 Effects Map

The second layer is a transparent texture, where both the transitory and permanent reactions of the organ skin to the instruments actions are stored. This texture will permanently evolve. The problem is that transferring a texture to the graphics board is slow, so doing it at each time step can spoil the real time simulation. So we have to minimize the amount of data sent to the graphics board, by taking advantage of the locality of changes in the texture. Another issue is to draw the effects in such a way they do not appear distorted despite the mapping is.

4.2.1 Editing of the map

In main memory we maintain separately a transparent texture containing only the permanent effects: at each time step the effect texture in main memory is restored using this one, then all the active sprites are drawn by software in the effect texture. Then the effect texture on the graphics board is updated, by transferring only the parts that have changed as we explain below. If a sprite is not transparent when becoming inactive (essentially for the cauterization effects), it is drawn in the permanent effects texture.

4.2.2 Parameterization

Drawing dynamically the sprites in the texture is very similar to the direct painting methods, for which a designer paints the texture content directly at the surface of the object. In order to apply this method, we need to define a bijective mapping between the surface and the texture space. Several methods exist to deal with the general case. The organ we are working on (i.e. the liver) is close enough to a sphere so that the central projection of a sphere parameterization on the organ surface is bijective, if the sphere center is correctly located (so that the surface is star-shaped relative to this point). This parameterization creates locations with singularities needing special treatment: the poles and the 'date change line' joining the two poles. The 'date change line' treatment is similar to what has to be done in the classical case of torus topology: the part of the drawing that is outside one side has to be drawn on the other side (see Fig.10). The poles are more of an issue. We chose the parameterization in such a way that the poles come to two locations that are far from the potential operation areas (the vessels entry at the bottom of the liver, and the center of the surface on top).

4.2.3 Suppressing the distortions

The basic direct painting method (Hanrahan and Haeberli 1990) automatically cancels the distortions of the map assuming the texture pixels are drawn one by one, which is not our case (moreover, in the original method there are as many vertices as there are texture pixels). We have to pre-distort a sprite that is drawn in the texture in such a way that it will appear undistorted once mapped: a disk on the surface should be drawn in the texture as an ellipse, whose axis and radius corresponds to the eigenvectors and the eigenvalues of the Jacobian of the mapping at this location (see Fig. 8). In fact we don't need to explicitly construct this ellipse: a location U in the neighborhood of the ellipse center Uc in the texture space can be quickly converted into a location P in the neighborhood of the disk center Pc in the surface space using the Jacobian:

$(P - Pc) = J(U - Uc)$. The square distance from P to Pc indicates if the location is inside or outside the disk, and this distance can be used as a parameter of a function controlling the profile of color and opacity in this disk. The Jacobian also provides a bounding box of the ellipse (the two sizes are given by the norm of the two columns times the sprite diameter). So, to draw a sprite in the texture, we scan the bounding box area and proceed for each texture pixel as stated above (the factors in the transform can be computed incrementally).

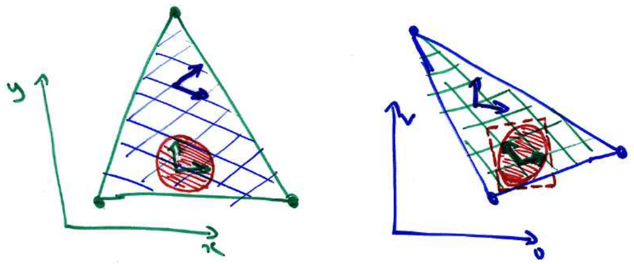

Fig. 8: Due to the mapping distortion, a disk on the surface appears as an ellipse in texture space.

As the mapping is linear within each face, the Jacobian is constant within each face (a part of the disk may exit the face, but we assume that the mapping distortion is not too different for this part of the disk). We could precompute and store the Jacobian for each face (this is not memory consuming as meshes used in surgery simulations are usually simple, due to the cost of simulating deformations). In our implementation at that time we still compute it on the fly, by combining the Jacobian J_{Uu} from barycentric to textural coordinates and the Jacobian J_{Ux} from barycentric to surface coordinates that are both trivial to get ($J_{xu} = J_{Uu}.J_{Ux}^{-1}$). This only needs to be evaluated once per sprite. Moreover, we cache the result in order to avoid recomputing it if the next sprite lies on the same face of the mesh.

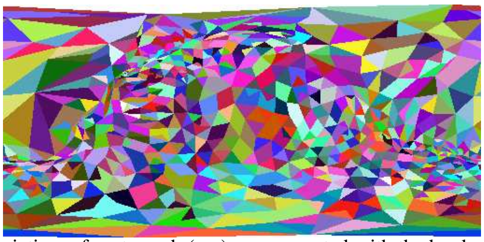

Fig. 9: The table associating a face to each (u,v), precomputed with the hardware using an item-buffer.

4.2.4 Finding a location

When the sprite shows an instant reaction to an instrument action, the location of this action (in particular the face) is known [1]. But during the simulation of the dynamics effects, especially for the rolling of blood drops on the surface, we need to know the location on the surface where a given (u,v) texture location projects to, which corresponds to the reverse mapping. Finding quickly which face contains the given (u,v) value is not easy. A 2D table indicating the subset of faces that intersect each given $[u,u+du]\times[v,v+dv]$ interval would decrease a lot the number of inclusion tests to be done, but it would still be costly to proceed the inclusion test, except if the table is large enough that each cell refers to a single face in almost every case. Once the face number is known, it is easy to compute the exact 3D location corresponding to the (u,v) by linear interpolation of the face vertices. The solution we propose is very close to this, and the suggested table can be built using the hardware: we once render the organ in the texture space (see Fig. 9) using the (u,v) instead of the (x,y,z) as the vertices coordinates, and using for face color the face number (with the shading disabled) [2]. The resulting image is uploaded in main memory before the simulation, then used as a

[1] During our tests we use the mouse, getting the contact location using the feedback rendering mode (*picking*). In our integrated platform the tool is controlled in 3D using a Phantom© arm, and we rely on the (Lombardo et al. 1999) hardware accelerated collision detection technique to get this location.

[2] This technique is called the *item-buffer* and has been used previously many times, for instance in (Baum and Winget 1990) for the computation of form-factors using the hardware. It has been originally introduced by (Weghorst et al. 1984) to accelerate ray-tracing. To be noted that we use it in texture space, instead of geometric space.

table associating a face number to each (u,v). Special care has to be taken for the faces crossing a singularity (a pole or the 'date change line') in order to draw them correctly, as explained on Fig. 10: each pixel should be covered exactly once.

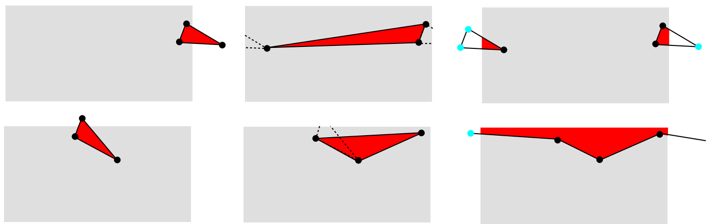

Fig. 10: Wrapping issues on the texture planisphere, at the date change line (top) and at the poles (bottom). We might want to draw a triangle that goes beyond the texture limits (left). The vertices are wrapped using a simple modulo, but they no longer describe the right triangle (middle). We have to clip correctly the triangles against the texture border (right).

4.2.5 Optimizing the texture transfer

The only parts of the effects texture that have changed since the last time step and need to be updated on the graphics board are the areas covered by the active sprites plus the ones of the sprites that have just disappeared (i.e. the background should probably appear). The first idea is to transfer only the bounding box area of these sprites (in OpenGL it corresponds to the *subtexture* feature). But the sprites generally overlap, so that the total number of pixels transfered could be very large (see Fig. 11). The bounding box of all the sprites belonging to a given snake may also be the unit of transfer. But when the sprites lie on the diagonal of this box, a lot of useless pixels are transfered. The optimum is in between: we group the sprites by a given amount, which is estimated to balance the redundant pixels and the useless pixels.

If a snake contains N sprites of size $L \times H$ that have a (dx,dy) offset between each other, then if we make k groups of sprites (assuming N/K is integer), the number of pixels transfered[3] is $k((N/k-1)dx+L)((N/k-1)dy+H)$. The pixel saving between

[3]One group contains $n = N/k$ sprites. If these are evenly spaced, the group is $(n-1)dx+L$ large and $(n-1)dy+H$ high.

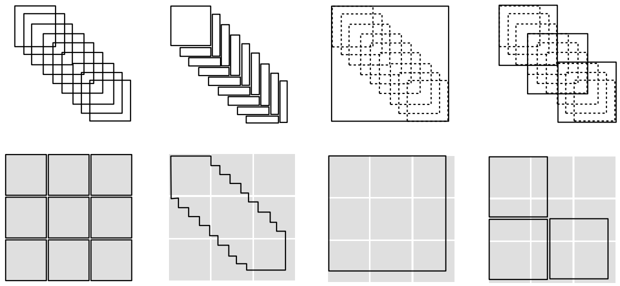

Fig. 11: Minimizing the pixel transfer when drawing snakes. *Left*: as the sprites overlap, transferring successively every sprites is redundant (on the bottom row, the amount of transfered data is figured). *Middle-left*: an optimum pixel set with no redundancy can be defined. Alas, the amount of context switches going with numerous blocks to be transfered and the limitations concerning memory alignment (disadvantaging small dimensions) make this solution non practical. *Middle-right*: transferring the whole snake's bounding box as one single block avoid pixels redundancy, but useless pixels are also transfered. *Right*: transferring the bounding box of groups of sprites is a tradeoff between the amount of redundant or useless pixels. An optimum can be found.

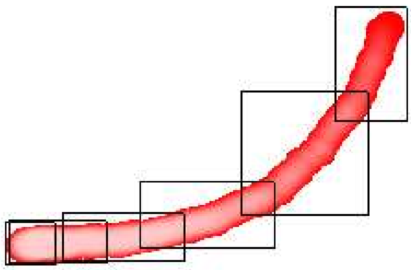

Fig. 12: Near-optimum texture area to transfer.

k grouping and $(k+1)$ grouping is the difference $-N^2 dxdy/k(k+1) + dxdy + LH - (dxH + dyL)$. This gain is positive as long as $\sqrt{k(k+1)} < N/\sqrt{ab-(a+b)+1}$ with $a = L/dx$ and $b = H/dy$. $\sqrt{k(k+1)}$ is close to k for k large enough. Our choice is to take for k the integer part of $N/\sqrt{ab-(a+b)+1}$ (using the mean values for L, H, dx and dy). An example of near-optimum grouping is presented on Fig. 12.

As stated above special care has to be taken when a sprite overlaps the 'date change line'. Such a sprite is split into two rectangles appearing on the two opposite sides of the map. For the optimum transfer evaluation, the parts of the snake that fall before

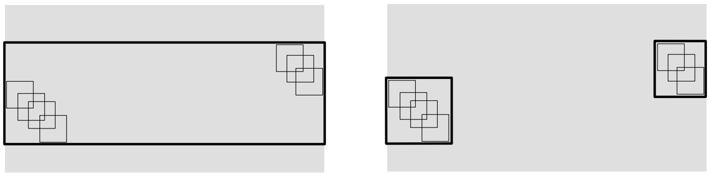

Fig. 13: When a snake is wrapped on the date change line, locality is lost and a bounding box contains a lot of useless pixels (left). Each wrapped portion of a snake should be considered independently (right).

and after the singularity are considered as two different snakes as figured on Fig. 13.

4.3 Reflection Map

On an organ the specularity is generally due to wetness, which yields mirror reflection on very wet locations and rougher reflection on the locations where the wetness layer is thiner, sticking to the surface grain. We could set an image of the light source in the reflection texture and simulate the roughness by accumulating rendering passes using slightly modified normals. This would obviously be costly. We precompute instead the effect of these by setting in the reflection texture (which constitutes the third superimposed layer) the convolution of the perfect mirror image of the light source (i.e. a ring) and a Gaussian 2D kernel figuring the normals variations (see Figs. 14 and 15). The first is scaled inversely with the light distance, while the second is scaled proportionally to the roughness (i.e. the Gaussian standard deviation). Thanks to the radial symmetry, we can even use an 1D radial Gaussian kernel [4].

Alas, this texture has to change with the distance of the light, and should thus be re-computed and transferred at each time step during a camera move, which certainly reduces the frame-rate. We introduce a solution that takes advantage of the spot symmetry: since all the points in the environment texture that are at the same angular distance of the optic axis (located at the center of the map) will show the same highlight intensity, we replace this explicit reflection map by the composition of a map of

[4] This convolution could be computed using the graphics hardware as done in (Soler and Sillion 1998) to get soft shadows.

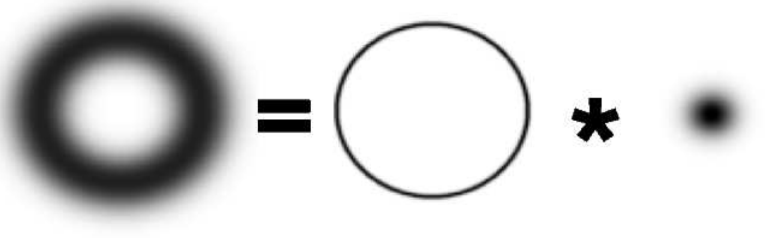

Fig. 14: The reflect texture is the convolution of the image of the source (i.e. a ring) and the roughness signature (i.e. a Gaussian).

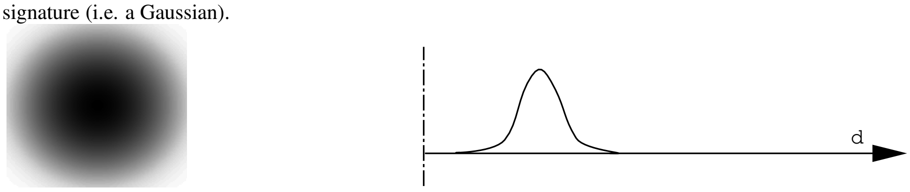

Fig. 15: The 2D radial ramp and the 1D reflect profile (i.e. the glossy ring seen in 1D).

the angular distances (i.e. a radial ramp) and an indirection table (i.e. a look-up table) that converts a radial distance into intensity. That way, the 2D map is constant and loaded once, while the 1D look-up table can easily be updated at every frame. It contains the convolution of the light ring (in 1D a simple peak) by a Gaussian, that is simply a translated Gaussian (as figured on Fig. 15). In fact, the 2D map can have a very low resolution, as the values will be recovered by the texture interpolation. The feature used for the indirection is the *texture color look-up table* of OpenGL, which is hardware-accelerated only on high-end graphics boards (such as the Infinite Reality). On low-end graphics boards, this can be emulated using color index textures: the look-up table is the color palette. To be noted that on the SGI O2 the textures are stored in main memory, thus suffering little penalty for texture loading.

Highlight on drops

In our implementation, we do not treat the highlights on spots in particular. If the reflect layer is drawn after the effect layer then the highlights cover the drops as well, in the opposite case the drops have no highlight. It would be easy to add a small white disk in the blood sprite in order to simulate an ideal drop highlight, but this would'nt correspond to reality as blood appears as flat spots rather than spherical drops. A better solution would be to apply the reflection map on the effect layer using a smaller roughness coefficient than for the skin. As this would require more passes, we preferred to neglect handling specific highlights for drops.

5 Evolving Components

5.1 Textural snakes of sprites

As we have suggested in section 4.2, we represent the various effects by a list of snakes, simulating three kinds of phenomena. A snake consists of a list of sprites. The motion of an effect is suggested by creating a new head and aging the previous sprites. An automaton manages the change in appearance as a sprite ages depending on its kind, and when a sprite should be deactivated. This happens when a sprite is at the tail of a drop, becomes too transparent, or gets the status of a permanent effect, as for a burnt spot. We store in each sprite structure its geometric characteristics (radius, bounding box), its automaton parameters (age, opacity), and a pointer to its location in the texture in main memory. The dynamics characteristics (velocity) only exist for the head, so we store them in the snake structure.

5.2 Automata

The snakes for drops are usually the longest. We design a blood sprite as a red disk with an opacity proportional to $1 - \frac{1}{(r/R)^2}$ where R is the sprite radius and r the distance of the current sprite pixel to the drop center. The aging consists of the multiplication of the global opacity by a fading coefficient. A sprite is deleted when its opacity is too weak or if a maximum length (i.e. age) is reached. In case of collision with another drop, which is tested using the sprite list associated with the face where the head sprite

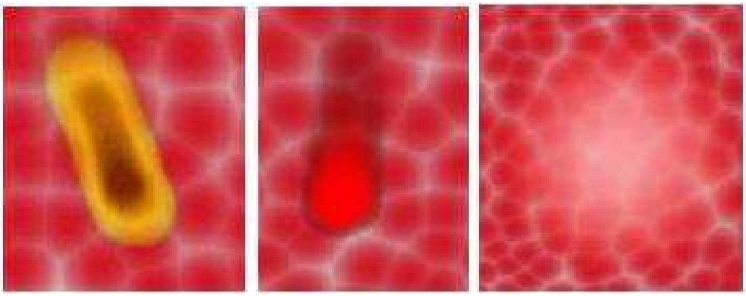

Fig. 16: The three kinds of dynamics effects: cauterization, blood drops and whitening.

lies, the tail of the collided drop is canceled and the radius of the collider increases.

The whitening corresponds to large, semi-transparent white disks. The head size and opacity increases as its cause of creation persists (i.e. contact of the tool with pressure), then the fading occurs by decreasing slowly the opacity and size.

The cauterization works in quite a different way: as the effect of heat depends on the previous state of the skin at this location (e.g. the physician might pass the cauterization tool several times over a location in order to cut the skin), a memory of this state needs to be maintained out of the snakes, otherwise we should keep the burning sprites forever just in case they might have to be reactivated. Thus we store an extra map in main memory containing the amount of heat received in each texture pixel, which is incremented at each new sprite creation. A software lookup table is then used to turn the heat received into color before drawing the sprite, from yellow for a weak burning (used by the physician for marking) to brown-black for cauterization. The snake is of minimal size for this kind as the sprites get a permanent effect immediately.

5.3 Drop motion

The drops have the most freedom: once created as a consequence of the user action, they evolve by themselves, under the influence of gravity. As for the other snakes, the head is the main element: the motion of the drop is simulated by creating a new sprite at the new head location and by fading the previous sprites. While for the other effects the new location of the head is simply the new site where the instrument acts on the organ skin, for a drop it results from Newtonian physics. The issue here lies in the coordination of the different spaces: forces are expressed in the 3D world, while the head sprite belongs to the texture world. To cope with this, we proceed as follow. The acceleration is computed in 3D, taking into account the gravity, the friction and the surface reaction. The evaluation of this last force requires the knowledge of the surface normal at that location. Thanks to the inverse mapping table (cf section 4.2.4), we know the face where the (u, v) location of the sprite lies, which allow us to compute the normal by linear interpolation of the normals at the face vertices. The 3D velocity is then updated according to the acceleration and projected on the surface. The Jacobian of the mapping at this face is then used to convert this 3D velocity into a 2D displacement in the texture space.

Note that the friction is lower on a wet area: a drop coming there rolls faster. Such an area occurs where a drop has recently been, which means that the drop head is colliding with a drop tail that is there. We can simulate this phenomena thanks to the collision management. This is fortunate because this results in the merging of drops, which has the good effect of preventing the growth of the total amount of drops.

5.4 Real-time balance

If the rendering needs more than a 25^{th} of second, the simulation is no longer real-time. This has two consequences: the screen refresh rate is less comfortable, and the simulation time base no longer matches the user time base.

The last issue is the worst, and is corrected in the following way: we measure the delay dt between the two previous frames, and we use this value as the physical time step in the simulation. This allows the simulation to run correctly (e.g. the distance covered by drops) whether the rendering is very quick or very slow.

Another problem occurs due to the events generated by the interface: the processing of an event includes a localization determination that is similar to a *picking* operation (this corresponds to the *feed-back buffer* feature on OpenGL). This requires a partial rendering pass with no effect on the screen, in order to determine which face is below the pointer (either 2D, e.g. the mouse, or 3D, e.g. the Phantom© arm). If too many events per second are generated by the computer, these extra renderings can saturate the pipe-line. We thus have to discard a proportion of events in such a way that this does not occur.

6 Results

We have implemented and tested the proposed solutions on an Onyx2 Infinite Reality2 and on SGI O2, as shown on Figs. 4, 6, 16 and 17. On the IR2, a regular frame rate of 25 fps is obtained while simulating 10 drops of 30 sprites, which allows further extensions. The frame rate is 8 fps on O2 in the same conditions. Without the moving drops, rendering the three layers can be done at more than 50 fps on the IR2 and 14 fps on the O2.

The liver geometric model is extracted from scanner data, and its surface is simplified down to 1200 triangles (the volumetric simulation of deformations cannot handle more in real time). The introduction of extra vertices to superimpose the regular texture mesh used for the skin layer brings the amount of triangles up to 3300, which is still very light for hardware rendering.

First demonstration to the physicians of our research project shows their deep interest and belief that this will lead to a usable training tool.

Evolution of graphics accelerators

New powerful graphics boards such as the Nvidia have appeared by the end of the project. It would be interesting to investigate what could be saved or extended using these boards. Multi-texturing and per-pixel shading would probably allow to simplify the rendering and decrease the number of passes. Bump mapping ability could certainly improve the skin aspect. On the other hand we rely on feedback rendering, subtextures and lookup tables, which are not yet handled by these new boards. Nevertheless, we hope to have proved that the amount and quality of visual effects that can be handled in a real-time simulation strongly depends on the richness of the features offered by the boards, more than the brute amount of polygons per second they can draw.

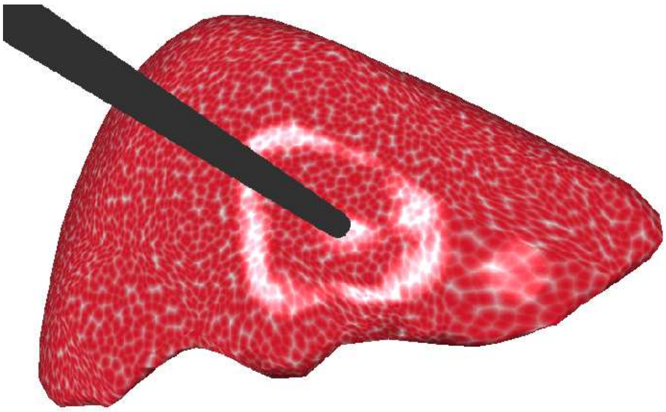

Fig. 17: A specular ring underlying the curvature change due to the contact of the instrument.

7 Conclusions

We have reached our aim of getting a real-time rendering for our laparoscopic simulator including the targeted features. The rendering stage has been integrated with the animation stage, as illustrated on Fig. 17.

Of course the work is far from being finished. We first need to tune all the colors and textures in coordination with physicians[5] in order to match the realism issue needed for the quality of immersion. Then we will have to address the simulation of our first surgery, the cutting and ablation of a part of the liver. This creates new surfaces whose surface area increases with time, yielding non trivial texturing problems. Vessels

[5]For instance, the skin texture used in this paper corresponds to a pig liver, which is the animal used for the training. It should better be human liver skin.

and ‘garbage’ should also appear between the two internal surfaces. Concerning the realism, surrounding organs should be introduced to help believing the simulation (real livers are not floating in the air). Moreover, these organs interact with the main organ. The grain of the liver skin appearing in the highlight could also be simulated. These are some of the goals of CAESARE, the cooperative project continuing AISIM.

Acknowledgments

AISIM was a cooperative research project founded by INRIA from 1997 to 1999. It is continued by the CAESARE cooperative project founded by the French Research Department from 2000 to 2002. We thank Hervé Delingette who has initiated this project, and Jean-Christophe Lombardo who has animated it during the three years. We thank Pierre-Olivier Agliati, Antoine Leroy and Sylvain Trimoreau, who have worked during their training on the integration of the various rendering methods with the simulation platform. Thanks are also due to James Stewart who carefully reread this paper.

A pseudo-algorithm

```
load liver mesh
load texture mesh
parameterize extra vertices with barycentric coordinates
load the four triangular skin patterns
send them to the graphics board
load the (u,v) of the regular nodes relatively to the
triangular tiles
build (u,v) for the effect map (spherical projection)
build the face(u,v) table using the hardware
build the radial ramp texture for the reflection map
send it to the graphics board

repeat forever:

/* rendering */
given the point of view, set the ModelView matrix
draw the skin layer:
bind triangle pattern 1
draw faces textured with it, using (u,v) relative to the tiles
same for pattern 2,3 and 4
draw the effects layer:
blending on
bind the effect texture
draw faces, using built (u,v)
draw the reflects layer:
blending on
set the spheremap (u,v) calculation mode
bind the reflect texture (i.e.  the radial ramp)
build the 1D reflect profile (knowing camera distance and
skin roughness)
send it to the graphics board (as texture lookup or
palette)
draw the faces
```

```
/* simulation */
handle user events, test for collisions with instruments
-> proceed liver distortion (out of the scope of this pa-
per)
-> new sprites are created
update effect texture:
erase sprites in effect texture in main memory
delete dead sprites
add created sprites
paint sprites in effect texture in main memory
send modified parts in effect texture on the graphics
board
execute sprites automata (color and transparency changes)
proceed motion:
for each moving sprite (i.e.  head of the moving snakes)
get face(u,v)
compute N
compute V from Newton laws
compute motion in texture space
update (u,v)
handle collisions
```

References

[AISIM project] AISIM project. Relevant URLs.
AISIM project: *(remark: the version in french is more recent)*
`http://www-sop.inria.fr/epidaure/AISIM/`
CAESARE project:
`http://www-sop.inria.fr/epidaure/CAESARE/`
Real-time simulation of deformations:
`http://www-imagis.imag.fr/~Marie-Paule.Cani/foie.html`
`http://www-imagis.imag.fr/~Gilles.Debunne/`
Real-time rendering:
`http://www-imagis.imag.fr/~Fabrice.Neyret/laparo/`.

[Baum and Winget 1990] Baum, D. R. and J. M. Winget. Real time radiosity through parallel processing and hardware acceleration. *Computer Graphics (1990 Symposium on Interactive 3D Graphics) 24*(2), 67–75.

[Catmull 1974] Catmull, E. E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.d. thesis, University of Utah.

[Cotin et al. 2000] Cotin, S., H. Delingette, and N. Ayache. A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. *The Visual Computer 16*(8), 437–452.

[Debunne et al. 1999] Debunne, G., M. Desbrun, A. Barr, and M.-P. Cani. Interactive multiresolution animation of deformable models. In *10th Eurographics Workshop on Computer Animation and Simulation (CAS'99)*.

[Debunne et al. 2000] Debunne, G., M. Desbrun, M.-P. Cani, and A. Barr. Adaptive simulation of soft bodies in real-time. In *Computer Animation 2000*.

[Fournier et al. 1998] Fournier, P., A. Habibi, and P. Poulin. Simulating the flow of liquid droplets. In *Graphics Interface'98*, pp. 133–142.

[Hanrahan and Haeberli 1990] Hanrahan, P. and P. E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In F. Baskett (Ed.), *Computer Graphics (SIGGRAPH '90 Proceedings)*, Volume 24, pp. 215–223.

[Lombardo et al. 1999] Lombardo, J.-C., M.-P. Cani, and F. Neyret. Real-time collision detection for virtual surgery. In *Computer Animation'99*.

[Miller and Hoffman 1984] Miller, G. S. and C. R. Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environments. In *SIGGRAPH '84 Advanced Computer Graphics Animation seminar notes*.

[Neider et al. 1993] Neider, J., T. Davis, and M. Woo. *OpenGL Programming Guide*. Reading MA: Addison-Wesley.

[Neyret and Cani 1999] Neyret, F. and M.-P. Cani. Pattern-based texturing revisited. In *SIGGRAPH 99 Conference Proceedings*, pp. 235–242. ACM SIGGRAPH: Addison Wesley.

[Nvidia] Nvidia. Developer - white papers. `http://www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame`.

[Perlin 1985] Perlin, K. An image synthesizer. In B. A. Barsky (Ed.), *Computer Graphics (SIGGRAPH '85 Proceedings)*, Volume 19(3), pp. 287–296.

[Picinbono et al. 2000] Picinbono, G., J.-C. Lombardo, H. Delingette, and N. Ayache. Anisotropic Elasticity and Forces Extrapolation to Improve Realism of Surgery Simulation. In *ICRA2000: IEEE International Conference Robotics and Automation*.

[Reeves 1983] Reeves, W. T. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics 2*, 91–108.

[Reeves and Blau 1985] Reeves, W. T. and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky (Ed.), *Computer Graphics (SIGGRAPH '85 Proceedings)*, Volume 19(3), pp. 313–322.

[Siggraph Course Notes 1998] Siggraph Course Notes. *Advanced Graphics Programming Techniques Using OpenGL*. Addison-Wesley. `http://www.sgi.com/software/opengl/advanced98/notes/`.

[Silicon Graphics a] Silicon Graphics. *Way cool, way fast OpenGL rendering techniques*. `http://reality.sgi.com/opengl/tips/`.

[Silicon Graphics b] Silicon Graphics. *Witches Brew: source + docs on Impressive Programming*. `http://toolbox.sgi.com/TasteOfDT/src/exampleCode/WitchesBrew/`.

[Soler and Sillion 1998] Soler, C. and F. X. Sillion. Fast calculation of soft shadow textures using convolution. In M. Cohen (Ed.), *SIGGRAPH 98 Conference Proceedings*, pp. 321–332. ACM SIGGRAPH: Addison Wesley.

[Weghorst et al. 1984] Weghorst, H., G. Hooper, and D. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*(1), 52–69.

[Worley 1996] Worley, S. P. A cellular texturing basis function. In H. Rushmeier (Ed.), *SIGGRAPH 96 Conference Proceedings*, pp. 291–294. ACM SIGGRAPH: Addison Wesley.