ESAIM: Proceedings CEMRACS 1999
URL: http://www.emath.fr/Maths/Proc/Vol.10

3D COMPUTING USING VIRTUAL REALITY DATA.

Stéphane Del Pino[1]

Abstract. Virtual Reality does not give a detailed description of the surface of the objects in the scene it represents. Rather than trying to solve the problem of data translation, we propose to apply a *fictitious domain* like method. We enlighten in this paper the computer science difficulties of such an approach and give some solutions illustrating its efficiency for an unsteady heat equation.

AMS Subject Classification. 65Y99, 65N30, 68N15.

The dates will be set by the publisher.

1. Introduction.

Let $\Omega \subset \mathbb{R}^3$, the aim of this work is to solve the general problem

$$\begin{cases} au - \nabla \cdot \nu \nabla u = f, & \text{in } \Omega, \\ u = u_0, & \text{on } D, \\ \nu \cdot \nabla u = g_0, & \text{on } N, \\ \alpha u + \nu \cdot \nabla u = g_1 & \text{on } R; \end{cases} \tag{1}$$

where $\{D, N, R\}$ is a partition of $\partial\Omega$. We will aim at the case when $u(x) \in \mathbb{R}^m$ such that $a(x) \in \mathbb{R}^{m\times m}$, $\nu(x) \in \mathbb{R}^{3\times m\times 3}$, $f(x), u_0(x), g_0(x), g_1(x) \in \mathbb{R}^m$

In a series of notes [5, 6, 7], J.-L. Lions and O. Pironneau showed that new techniques to speed up computations by domain decomposition arise from *Constructive Solid Geometry* (CSG). In this paper, we will focus on the geometrical aspects of *Virtual Reality* (VR) and the interest of mixing it to a *high* level software language. We will enlighten the difficulties of its implementation and propose an appropriate toolkit. Then we will show the efficiency of the method applied to an unsteady heat equation resolution.

2. Virtual Reality.

To represent the world, VR has to compromise between realistic rendering and speed. In VR one can find two different strategies to achieve this goal.

The first is the one used in 3D video game: surfaces are described by triangulation, and then the rendering is done at the hardware level by video cards able to compute the color of billions of triangles each second. The

Keywords and phrases: fictitous domain, scientific computing, virtual reality, programming language

[1] Laboratoire d'Analyse Numérique, Université Paris 6, 75 252, France
e-mail: delpino@ann.jussieu.fr

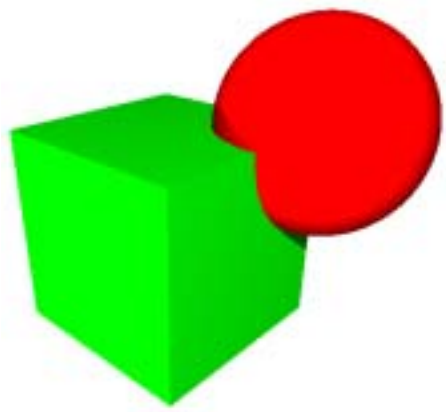


FIGURE 1. *A scene rendered by* `POV-Ray`*, using the* zbuffer *algorithm.*

problem is that such surface meshes cannot be used to generate a 3D grid since they are not conforming and lots of triangles may overlap. So this kind of data cannot be used easily for computation.

The second strategy is the use of CSG, which is a way to build objects of complex geometry using basics shapes (sphere, cube, tore, cone, ...), called primitives, performing simple transformations (rotation, translation ans scaling) and set operations (union, intersection, difference). This method is intensively used in image synthesis. The interest of this approach is that to render the scene, there is no need to compute the results of the CSG operations, and then to compute the object surface. The image is computed using the *zbuffer* algorithm:

```
for(int t=0; t<tmax; t++)
  if (z[t]<=zbuff[x[t]][y[t]]) {
    putpixel(x[t],y[t],z[t]);
    zbuff[x[t]][y[t]]=z[t];
  }
```

`zbuff` has been previously initialized to very high values.

`POV-Ray`[12] is a renderer which is quite popular in the image synthesis community. Its geometrical description is *highly* based on the CSG, even if it can deal with meshes. The reason is that `POV-Ray` is an engine which uses a *human readable language* to perform its rendering and so, it does not generally have to treat meshes which come more easily from *modelers*.

The following code is the description of a `POV-Ray` scene:

```
sphere {<1,1,1>, 0.5     texture {pigment rgb <1,0,0> }}
box    {<1,1,1>, <0,0,0> texture {pigment rgb <0,1,0> }}
```

The scene is a sphere of center $(1,1,1)$ and radius 0.5, and the cube built on vertices $(0,0,0)$ and $(1,1,1)$. The `texture` modifiers gives the colors of the objects (the sphere is green and the box red).

In fact this is not a complete `POV-Ray` file: rendering information are missing (camera, light position, ...). The result of the rendering is given on the figure 1.

`POV-Ray` is not the "standard" VR language. The Web 3D Consortium [1] has defined such a standard: VRML, the *Virtual Reality Markup Language*[4]. The same kind of scene could easily be done in VRML, but VRML's weak point, is that it does not implement all sets operations: *only set union are allowed.*

The reason why `POV-Ray` language has been preferred is that it is stable, actively maintained and used. Moreover `POV-Ray` comes with its sources and can also be used as a post-processing visualization tool applying the patch of Suzuki [10].

3. The freefem3d language

The Freefem project was started in 1995 by Bernardi et al [1] as an education software. It has received a large audience and many users requested a 3D version of this solver.

The actual version, `freefem+`, is a 2D-PDE solver driven by a language which allows to solve more general problems than (1), on a 2D geometry. Moreover, the use of such a high level language which handles loops, variables, control structures, ... Functions such as mesh refinement or interpolation provides a lot of flexibility. Therefore it is a very convenient tool for Domain Decomposition.

Since we want the solver to use geometry data coming from VR, we have to provide to `freefem3d` a different PDE language, trying to be as close as possible to `freefem+` and also similar to C++.

Here is an example of a program using the syntax we chose:

```
vector n=(20,20,20);
vector a=(-1,-1,-1);
vector b=(0,0,0);
function f = sin(x^2);

// Reads the geometry in the file
scene S("geom.pov");

//define a domain
domain Omega(outside(<1,0,0>), S);

// Initialize a Grid using a,b and n.
structmesh C(a, b, n);

// Problem definition.
solve(u) in Omega by C {
  -div(grad(u)) = f;
  u = 0 on C;
  dnu(u) = 0 on <1,0,0>;
}
```

This program solves the following problem:

$$\begin{cases} -\Delta u = \sin(x^2), & \text{in } C \setminus \overline{\Omega}, \\ u = 0, & \text{on } \partial C, \\ \nabla u \cdot n = 0, & \text{on } \partial\Omega. \end{cases} \tag{2}$$

[1] `http://www.vrml.org`

The domain Ω is the union of those objects in the VR scene, whose color is `<1,0,0>`. One has to remark that *union is **always** implicit for objects of the same color*. The color plays the same role as the usual *reference*. One could solve the "internal" problem ($-\Delta u = \sin(x^2)$ in Ω) using the keyword `inside` instead of `outside` in `Omega` definition.

Since the language supports loops, it can be used to solve time dependent problems: the user as to write a time discretization scheme using the language. Moreover, since it can handle the resolution of multiple problems, domain decomposition can also be treated easily. This flexibility will be shown in the Section 6.

4. Implementation.

Solving (1) using VR geometry data and a `freefem` like language requires a lot of computer science investment. To try to simplify the implementation of the code, one has to choose appropriate tools.

4.1. **The programming language.**

As in `freefem+`, the programming language used for the project is C++ [9]. There are many reasons for this. C++ is an *Object Oriented* (OO) programming language, which implies that it provides a framework guiding the developer to produce a more readable and easy to maintain code (using data protection and inheritance mechanisms).

Moreover, using the *expression template* mechanism and *template metaprograms*, respectively introduced by Veldhuizen[11] and Unruh, one can generate codes as efficient as the ones obtained Fortran, while using *operator overloading* which allows to write a code close to a mathematical formulation and therefore easier to produce.

C++ is really adapted to the VR approach: what is a VR scene? It is a collection of *shapes* which can interact using set operations. So, `Shape` is a *base* type. It means that `Shape` is an abstract *class* defining the interface for all its *derived* classes.

Since we want to build a characteristic function using VR, one of the thing that all *kinds* of `Shape` will have in common will be a *member function* that will indicate if a point is inside the `Shape`. Let us call this function `Inside(Point)`[2]. Since it is a *pure virtual* function of the base class, it has to be implemented in all derived classes to allow object *instantiation*. So, if one wants to add a new `Shape` type he will just have to provide an `Inside(Point)` function to his class deriving from `Shape`.

Let us look at the case of the `Union`. The `Inside` function will look like that:

```
const bool Union::Inside(const Pointx& P) const {
  bool inside = false;
  for (int i=0; i<nbShapes; i++)
    inside = (shapes[i].Inside(P) || inside);
  return inside;
}
```

The important thing is that this function is written once for all. When adding a new `Shape` type one does not have to change the `Union`. The efficiency of this method has already been established by in [2] for an electromagnetism problem.

[2] C++ code shown in this paper is not the one used in the software, its purpose is just for explanation. We strongly urge the interested reader to download and check `freefem3d` source code at `http://www.freefem.com`

The second reason to use *C++* is based on the `freefem+` success in dealing with non scalar PDE resolution: with `templates` one can perform a bloc matrix-vector product as *easily* as the scalar one. So, in this sense, writing a scalar solver and taking care of the `template` compatibility will provide a vectorial solver.

4.2. **Implementing languages.**

To exploit VR data and to manage them as in `freefem+`, we have to deal with the implementation of compilers for the programming languages which have been discussed in sections 2 and 3. This is a problem well known in Computer Science.

The VR language that we have chosen to use is `POV-Ray` [12][3]. Since this language is stable, what we need is just to implement a parser which is compatible with the grammar established by `POV-Ray` developers.

On the other hand, we have to develop a completely new language: even though an effort is done to be as close as possible to `freefem+`, the geometry description is so different that incompatibilities necessarily occur. And since the language is new, we have to take care that the *grammar* of the language is correct:

Assume that we want to be able to write:

```
dnu(u) = 0 on <1,0,0>+<0,1,0>;
```

which should be understood as: *Impose a Neumann homogeneous condition on the border of objects whose colors are* `<1,0,0>` ***and*** `<0,1,0>`. The problem here is that the language understands vector algebra. This means that there is a risk that `<1,0,0>+<0,1,0>` will be interpreted as `<1,1,0>`! In this case, the language *grammar* would be bugged, and you have to change the boundary condition definition to something like this:

```
dnu(u) = 0 on <1,0,0>,<0,1,0>;
```

This is simple but as the language grows, more subtle errors may arise, so tools which will help us to create a language are necessary.

The Computer Science community has developed such tools, called *compiler compilers*. One of the most common is `yacc` and its free version `bison` [3]. It allows the description and the validity checking of *LALR(1)*[4] grammar, and provides a syntax to automatically generate *C* or *C++ parsers*. One can use `flex` [8], as a `bison` companion: its goal is to provide a fast lexical analysis of the file.

5. The Solver

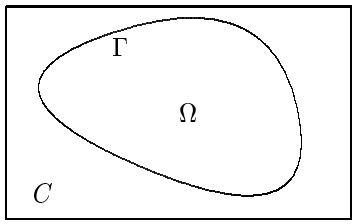


FIGURE 2. The geometry is approached using a fictitious domain like method.

[3]Others languages could be implemented in the future, but actually this language is the closest to our needs ...

[4]*LALR(1)* is the class of context-free grammars that Bison (like most other parser generators) can handle; a subset of *LR(1)*. *LR(1)* is the class of context-free grammars in which at most one token of look-ahead is needed to disambiguate the parsing of any piece of input.

Let Ω a bounded connected domain of $\mathbb{R}^n$ $(n \in 1, 2, 3)$. Let $\alpha, \mu \in \mathbb{R}^{+*}, f \in L^2(C)$ and $g \in H^{-\frac{1}{2}}(\Gamma)$, where $\Gamma \equiv \partial\Omega$. We consider the following problem:

$$\begin{cases} -\nabla \cdot (\mu \nabla w) = f, & \text{in } \Omega, \\ \alpha w + \mu \dfrac{\partial}{\partial n} w = g, & \text{on } \Gamma. \end{cases}$$

Its variational formulation: find $w \in H^1(\Omega)$ such that

$$\int_\Omega \mu \nabla w \cdot \nabla \hat{w} + \int_\Gamma \alpha w \hat{w} = \int_\Omega f \hat{w} + \int_\Gamma g \hat{w}, \ \forall \hat{w} \in H^1(\Omega)$$

can be reframed into: find $w \in H^1(\Omega) \oplus L^2(C \setminus \overline{\Omega})$ such that

$$\int_C \mathbf{1}_\Omega \mu \nabla w \cdot \nabla \hat{w} + \int_\Gamma \alpha w \hat{w} = \int_C \mathbf{1}_\Omega f \hat{w} + \int_\Gamma g \hat{w}, \ \forall \hat{w} \in H_0^1(C) \tag{3}$$

where C, the *fictitious domain*, contains Ω (see figure 2).

Here, Dirichlet boundary conditions are penalized into Robin conditions but several other formulations will be tested. The variational problem (3) is discretized using a Q1 finite element method, and the linear system is solved by the conjugate gradient method.

The solution of (3) is not *unique* so we take the one that is 0 in $C \setminus \Omega$.

To compute the solution of (3), one uses two data-structures returned by the compiler: the characteristic function of Ω and a triangulation for each `POV`-object.

6. Numerical results.

We test the method on the unsteady heat equation:

$$\begin{cases} \dfrac{\partial}{\partial t}(u) - \Delta u = 0, & \text{in } C \setminus \overline{\Omega}, \\ \nabla u \cdot n = 0, & \text{on } \partial C, \\ u = 1, & \text{on } \partial\Omega, \\ u(0) = u_0. \end{cases} \tag{4}$$

where Ω is a table at fixed temperature 1 and C is a room with perfectly isolating walls. So, that $u \xrightarrow{t \to +\infty} 1$.

The `POV-Ray` code associated with the geometry is the following:

```
// Rendering informations
camera { location <7,5,4> look_at  <0,1,0> }
light_source { <7,5,4> color rgb <1,1,1> }
light_source { <0,6,0> color rgb <1,1,1> }
background { color rgb <1,1,1> }

// The fictitious domain
box {
  <-3,0,-3>, <3, 3, 3>
```

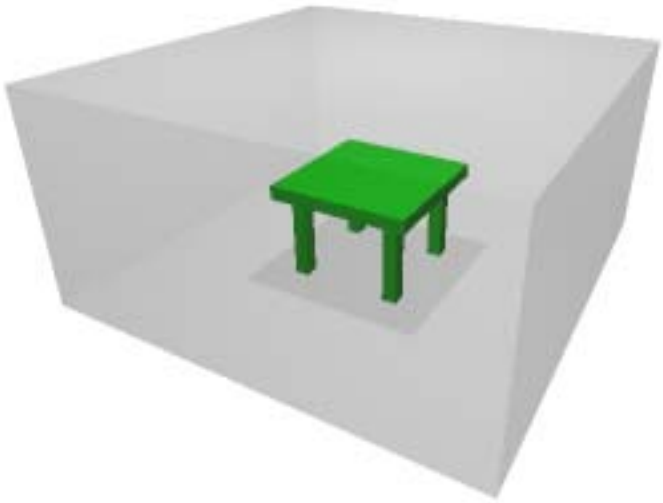


FIGURE 3. *The computational domain: C is the transparent cube, Ω is the table.*

```
      pigment { color rgbf <0.95,0.95,0.95, 0.8> }
      no_shadow
    }

    // The table
    union {
      box { <-1,  1.2,  -1>, < 1, 1.4, 1  > }
      box { <-0.6,1.3,-0.6>, <-0.8, 0,-0.8> }
      box { < 0.6,1.3,-0.6>, < 0.8, 0,-0.8> }
      box { < 0.6,1.3, 0.6>, < 0.8, 0, 0.8> }
      box { <-0.6,1.3, 0.6>, <-0.8, 0, 0.8> }
}
```

The generated scene can be seen on Figure 3. One should note that the first `box` defined is not part of the computation scene: the use of `rgbf` tells to the code that the box is the shape of C, so, to help the user, the code will check that it matches the definition of the mesh.

To compute the solution of (4) the following program is used:

```
vector a = (-3, 0,-3);
vector b = ( 3, 3, 3);
vector n = (64, 32, 64);

structmesh C(n,a,b);
scene S("table.pov");
domain Omega(outside(<0,1,0>), S);

function un_1 = 0;
double i=0; double dt=0.1;
do {
```

```
    solve(u) in Omega by C {
      -div (dt * grad(u))+u = un_1;
      u=1      on <0,1,0>;
      dnu(u)=0 on C;
    };
    function un_1 = u;

    dxplot("u"+str(i)+".dat", u, C);
    i++;
  } while (i<=5);
```

The mesh is $64 \times 32 \times 64$, the initial value is `un_1`$= 0$. One can see that the time discretization using a implicit Euler scheme is performed using the language.

References

[1] D. Bernardi, F. Hecht, K. Otsuka, and O. Pironneau. freefem+, a finite element software to handle several meshes. Downloadable from: `ftp://ftp.ann.jussieu.fr/pub/soft/pironneau/`, 1999.

[2] S. Del Pino, E. Heikkola, O. Pironneau, and J. Toivanen. A finite element methode for virtual reallity data. *C.R.A.S.*, June 2000.

[3] C. Donnelly and R. Stallman. *Bison: The Yacc-compatible Parser Generator.*

[4] J. Hartman and J. Wernecke. *The VRML 2.0 Handbook.* Addison-Wesley, 1996.

[5] J.L. Lions and O. Pironneau. Algorithmes parallèles pour la solution de problèmes aux limites. *C.R.A.S., 327*, Paris 1998.

[6] J.L. Lions and O. Pironneau. Domain decomposition methods for cad. *C.R.A.S., 328*, Paris 1998.

[7] J.L. Lions and O. Pironneau. Sur le contrôle des sytèmes distribués. *C.R.A.S., 327*, Paris 1998.

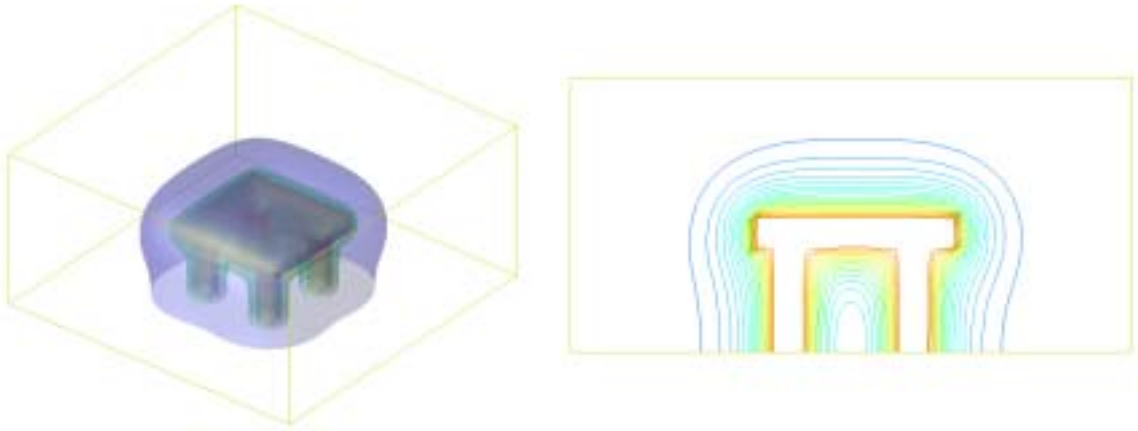
[8] V. Paxson. *Flex: A fast scanner generator.* Free Software Foundation.

[9] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 3nd edition, 1997.

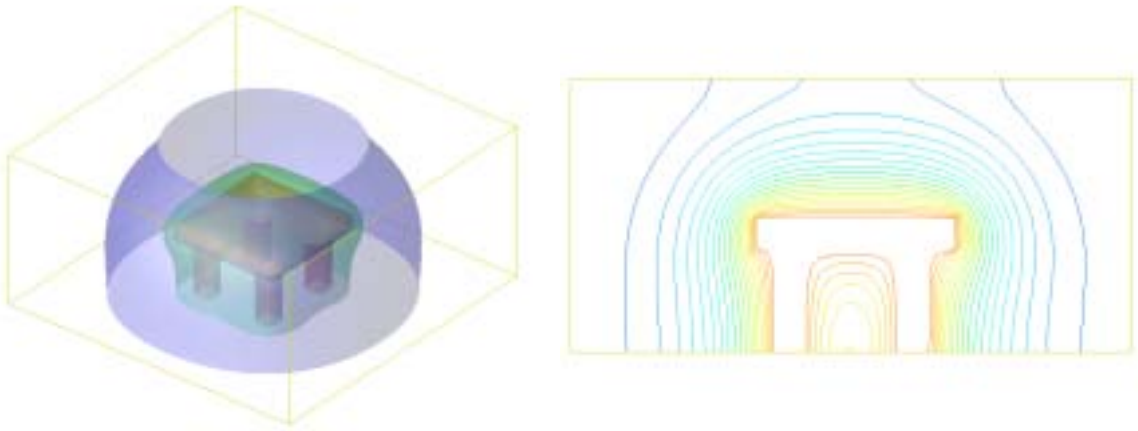
[10] R. Suzuki. A patch to `POV-Ray` for iso-surfaces. `http://www.public.usit.net/rsuzuki/e/povray/iso/index.html`.

[11] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

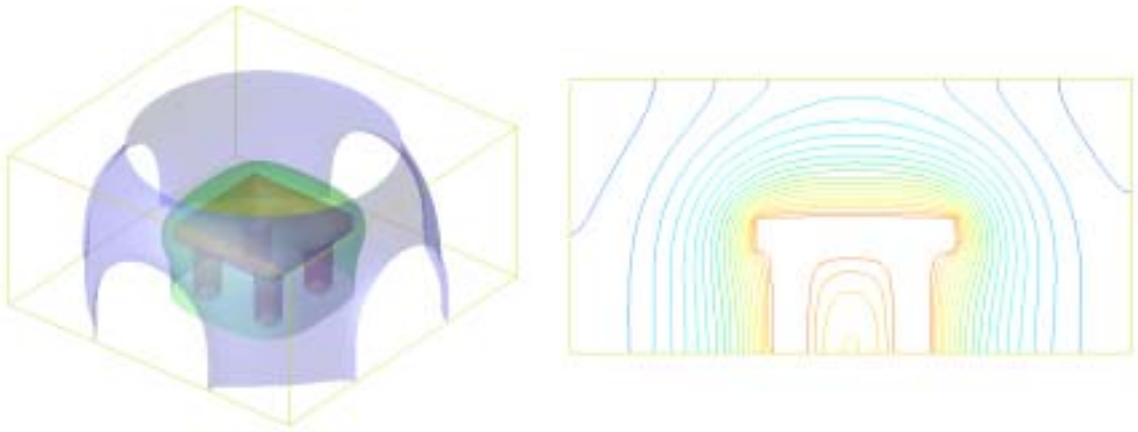
[12] A. Wardley. Persistence of vision, `POV-Ray`. `http://www.povray.org/`, 1995.



$t = 1.$



$t = 4.$



$t = 6.$

FIGURE 4. ***Left:*** *isosurfaces for values 0.05, 0.5 and 0.95.* ***Right:*** *the solution is mapped on the plan passing trough (0,0,0.7) whose normal is (0,0,1). Results are displayed using* OpenDX(`http://www.opendx.org`).