A Multiple Shooting Vectorial Algorithm for Progressive Radiosity

Blanca Garcia
Xavier Pueyo

Report LSI-94-5-R
A MULTIPLE SHOOTING VECTORIAL ALGORITHM
FOR PROGRESSIVE RADIOSITY

Blanca Garcia
Departament de LLenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Avda Diagonal,647
08028 Barcelona
Spain
phone :34-3-4016595
fax : 34-3-4016600
email : blanca@lsi.upc.es

Xavier Pueyo
Departament de Informàtica i Matemàtica Aplicada
Universitat de Girona
E.P.S.
Lluis Santaló s/n
17003 Girona
Spain
phone : 34-72-418417
fax : 34-72-418399
email : xavier@lsi.upc.es

December 1993
INDEX

1. Abstract
2. Introduction
3. Radiosity
4. Software optimization for form factors
5. Multiple shooting radiosity
6. Conclusion and future work
7. References
1- Abstract

Radiosity has been largely used in computer graphics for synthesis of realistic images. It is well known that this method has a high computational cost. This paper proposes an algorithm designed in order to take advantage of the use of vectorial computers. More precisely, we use a progressive radiosity approach where instead of shooting one patch at each iteration, we shoot a number of patches which is multiple or submultiple of the dimension of the vectorial processor each time.

2- Introduction

Computer graphics progress has reached a high level of realism during the last ten years. In 1984, the radiosity method was borrowed from thermal engineering to produce realistic images corresponding to ideally diffuse environments [1]. This technique is very expensive because visibility, one of the most time consuming functions in computer graphics, must be performed a lot of times. Many algorithms have been proposed to reduce the time required to obtain an image. Different ways have been investigated: the use of different kinds of coherence widely explored in visibility computation, the proposal of algorithms for approximated solutions, the use of high performance computing resources.

In this paper we propose to use an algorithm designed to take advantage of the features of a vectorial computer taking into account other ways of saving computational resources. With this target, we have chosen the progressive radiosity approach, which is widely extended to produce useful images in a substantially shorter time than classic radiosity. On the other hand, we have considered the feasibility of using surface coherence without lost of efficiency of our vectorial algorithm.

Section 3 introduces the radiosity techniques as well as a its progressive refinement approach. Next, in section 4, we shortly describe how a standard radiosity algorithm has been modified to take advantage of features of vectorization, followed by the results. Section 5 presents the new algorithm designed for vectorial processor and discusses its alternatives. Finally, at the end of this section, we show the results obtained with this specialised algorithm.
3. Radiosity

The radiosity algorithm is a method for evaluating the light intensity at discrete points in an closed environment[1]. This environment is composed by polygons which are split down into patches. Radiosity is computed at the center of those patches. The relationship between the radiosity of a given patch and the rest is given by the following formula:

\[ B_i = E_i + P_i \sum_{j=1}^{n} F_{ij} B_j \]

This equation yields a nxn matrix which can be enormous in a non trivial environment. Due to the strict diagonal dominance of this matrix, there is a cost proportional to the square of the number of patches.

\[
\begin{bmatrix}
1 - P_1 F_{11} & \ldots & -P_1 F_{1n} \\
\vdots & \ddots & \vdots \\
-P_n F_{n1} & \ldots & 1 - P_n F_{nn}
\end{bmatrix}
\begin{bmatrix}
B_1 \\
\vdots \\
B_n
\end{bmatrix}
= 
\begin{bmatrix}
E_1 \\
\vdots \\
E_n
\end{bmatrix}
\]

where

\( B_i \)= radiosity in surface i \\
\( E_i \)= emissivity of surface i \\
\( P_i \)= reflectivity of surface i \\
\( F_{ij} \)= form-factor between surface i and j \\
\( n \)= number of surfaces in the environment

A form factor is given by the geometric relationship of two patches and represents the ratio of energy leaving one and arriving to the other[1]. So, the sum of all the form factors from a given patch to the rest is equal to 1.

The form factor between finite patches is defined by:

\[ F_{A_i A_j} = \int_{A_j} \frac{\cos(\phi_i) \cos(\phi_j) HID}{\Pi r^2} \, dA_j \]
where

The mathematical expression does not take account the possibility of objects hiding all or part of one patch from another. Therefore, it is required determining which patches are visible from each patch, with a function called HID in the formula.

The overwhelming cost of radiosity lies on form factor computation which can be 80% of the total. To handle with this large-scale calculation, several approaches have been described, like the hemi-cube projection approach [6] and the ray-tracing approach [3] for instance. In both cases, form factors are found prior to beginning with radiosity, which leads to a massive CPU and memory costs.

The progressive approach[4] avoids this enormous costs by computing form factors on the fly. Now, instead of keeping the whole system of operations, to complete the energy exchange of the environment, a useful solution is computed. This is obtained by only shooting the energy of a reduced number of patches, those which contribute more to the illumination of the environment.

At any step, given a patch i (chosen by a certain criterium), every $F_{ij}$ is estimated. After used, the form factors are not kept, which prevents from memory problems. In return, if that patch is selected some time afterwards, everything must be done again, although the number of steps is expected to be very low compared to the total number of
patches. This way, form factors from a predetermined patch to the rest are estimated and its radiosity is spread accordingly to these values. In other words, form factors calculation and radiosity are not isolated but combined to get partial results each step.

4. Software optimization for form factors

In this section, the main goal to be achieved was to get a software which estimates the form factors of an environment with the help of vectorization.

As well known, vectorization has three basic rules: a) Only loops that have vectorizable expressions in its body can be vectorized, b) Only inner loops are vectorized, and c) Everything likely to inhibit vectorization must be eliminated from the loop when possible. This includes I/O expressions, function references, return sentences, indirect addressing, loops with goto or conditional branches, recurrences, ... to name the most important.

A number of vectorization techniques have been used to improve a standard progressive radiosity algorithm (focusing on form factors), whose results are presented in this section.

Memory

It is important to know how vectors are allocated in memory. Data access must be as fast as possible in order to make the most of vectorization. The memory of the computer used in our work consists of four sections and sixteen banks distributed as follows:

<table>
<thead>
<tr>
<th>Section 0</th>
<th>Section 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 0</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Bank 4</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Bank 8</td>
<td>Bank 9</td>
</tr>
<tr>
<td>Bank 12</td>
<td>Bank 13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 2</th>
<th>Section 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 2</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Bank 6</td>
<td>Bank 7</td>
</tr>
<tr>
<td>Bank 10</td>
<td>Bank 11</td>
</tr>
<tr>
<td>Bank 14</td>
<td>Bank 15</td>
</tr>
</tbody>
</table>

Fig 2

Consecutive elements of a vector (or matrix) are kept in this order: element 0 in bank 0, element 1 in bank 1, ..., element 16 in bank 0 again, element 17 in bank 1, and so on. This avoids delays on accessing these elements.

When manipulating a vector or matrix, one must be careful if the stride (logical distance between two elements in a vector) is greater than 1, especially if multiple of 2. As well known, any part of memory in any computer is not accessible for a certain time when an element of its is addressed for security reasons. So it is in the banks. If a loop asks for elements allocated in the same bank or section in a row when dealing with a vector, the time of execution will be outstandingly increased.
Implementation

The software used in our experiment estimates the patch-to-patch form factors. The objects of the environment are subdivided into polygons and patches. The user decides how many patches on a polygon will be with the input data when subdivision is performed.

The analytical formula to estimate a form factor can be simplified in such a way that we need only four things: two cosines, a distance and the HID value. HID measures the visibility between two entities within a range from 0 (no visible) to 1 (fully visible) and is very time-consuming. During the execution, the HID function is called very often, and represents about 90% of the total time in the original code although it was inlined.

Objects in the environment are supposed to be convex and determined by planes. We need to gather the relevant information into vectors, that is, the area and center of the patches along with the normal of the geometric plane they belong to.

In general, polygons will have many patches (Fig 4). We can use surface coherence to reduce the number of patches to deal with (Fig 5). In this figure, we can gather the patches whose polygon cannot be discarded using surface coherence. At any step, every operation is performed on a vector of dimension $P$, where $P$ is the final number of patches we have gathered.
Results

Once the software had been optimized, a number of executions were performed to measure the foreseeable improvements. Every trial consisted of a closed environment determined by a big cube and a predetermined number of smaller cubes allocated therein at random. The number of objects depended on both the number of patches we wanted in a certain scene and the subdivision of the polygons of the objects.

On the other hand, we wanted to test how polygon subdivision could affect the results. Following this idea, we tested scenes with the same number of patches in total, but the number of patches in a polygon were changed. The subdivision we tried were four, sixteen and sixty-four patches. This is shown in the charts below. We took account of the time devoted to form factor estimation.

The optimized code produced good speed-ups compared to the original code, around 40%. This includes the time used by HID and the rest of functions. We have observed that the function that estimates the visibility between two patches easily takes the 90% of the total CPU time in any case. This means that this function is the actual bottleneck of the algorithm as well known. Although it has been avoided any reference when possible, it is obvious that an optimization of this function would give us better results consequently.

The inlining of some functions that depend on HID gave a reduction of about 40% in its performance. This reduction remains regardless the number of patches in the environment or the subdivision level. The rest of the code, which uses vectorization, produces a speed-up ranging from 45% to 55%.

This ratio is not affected by the subdivision of the polygon although a slightly smaller reduction can be appreciated when the number of patches on a polygon is bigger. Both programs take account of the coherence to discard patches whose polygons are not visible to each other, that is why there are similar reductions in CPU time; in the case of optimized code, it is a bit more efficient.

The remaining of this section are charts where results can be seen graphically. Pay attention at the differences in the last page, where cpu time increases rapidly when subdivision is small (only four patches a polygon for instance). Recall that the total number of patches is kept equal.
CPU time reduction (vectorized code / original code x100)

<table>
<thead>
<tr>
<th>patch number</th>
<th>1,300</th>
<th>2,200</th>
<th>3,750</th>
<th>5,200</th>
<th>8,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 patches per polygon</td>
<td>53.44</td>
<td>58.22</td>
<td>56.94</td>
<td>56.36</td>
<td>60.06</td>
</tr>
<tr>
<td>16 patches per polygon</td>
<td>59.88</td>
<td>59.39</td>
<td>59.74</td>
<td>61.92</td>
<td>61.86</td>
</tr>
<tr>
<td>4 patches per polygon</td>
<td>61.59</td>
<td>62.82</td>
<td>61.94</td>
<td>64.45</td>
<td></td>
</tr>
</tbody>
</table>

Fig 6

CPU time (seconds) | Time Differences (absolute terms)

patches on a polygon | 4 | 16 | 64

- 5,200 patches
- 3,750 patches
- 2,200 patches
- 1,300 patches

Fig 7
Subdivision of every polygon: 4 patches
- O optimized software
- N no optimized software
Subdivision of every polygon: 16 patches
O optimized software
N no optimized software
Subdivision of every polygon: 64 patches
O optimized software
N no optimized software

Fig 10
5. Multiple shooting radiosity

In this section we present a new approach for solving the radiosity distribution system by using a specialized radiosity algorithm for vectorial computers. In the classical progressive radiosity, only a chosen patch gives its unshot energy to the environment (the shooting patch), as explained in section 4, at each iteration. In general, the selection of this patch is based on the amount of unshot energy it has. The process repeats that step until a certain threshold of unshot energy is achieved.

Given that the more use of vectors and matrices as large as possible, the better in a vectorial computer, it seems clear that we could shoot more than one patch in each iteration of the progressive algorithm. In other words, a group of shooting patches are manipulated simultaneously.

At first sight, two questions arise in this proposal: first, how many patches will be shot at each iteration and, second, how they will be chosen. The decisions about them will produce different performances.

The subdivision of an environment can be done in a rigid way in order to test the feasibility of the proposed algorithm, e.g. we can divide every polygon in a predetermined number of patches, irrespective of its area or emissivity. The subdivision of the scene is straightforward since the number of patches per polygon is predetermined. On the contrary, this arbitrary splitting may not suit the environment because it can overdivide small polygons and big ones may need more subdivision as well as zones with important radiosity gradient. Another way to do it could be more adaptative, following some criteria in order to subdivide the polygon attending to its characteristics. In general this adaptative way of proceeding is more suitable because it takes into account the environment features.

Once the environment has been subdivided, a group of patches has to be selected in order to distribute their unshot energy. Basically, either all the patches in the group belong to the same polygon or they lay on different polygons. The first case has various advantages: if the shooting patches belong to the same polygon and any group of N patches happen to be in the same situation, it is easy to take advantage of the coherence e.g. to discard polygons when form factors are estimated and operations like visibility computation may be performed for N patches in a row. This would save time and resources given that some operations will be performed once instead of N times by means of using surface coherence. In addition to this, no energy among the shooting patches has to be given out since every form factor between patches of a given surface is null. The election of the "shooting polygon", the polygon containing the selected patches, could be based on the unshot energy of their patches which is easy to estimate.

The second case is more complex. As classic progressive radiosity does, the N patches with the highest unshot energy could be chosen. This selection would require a longer search than the first case and they are very likely not to lay on the same polygon. In consequence, coherence would also be more difficult to apply than in the former case. In the normal course of events, there will be exchange of energy among the shooting patches because now their form factors will not be null.
Let us recall how the progressive radiosity is:

```c
main(inputfile, outputfile) {
    read_data(inputfile);
    subdivision_into_patches;
    step=0;
    while (not end()){
        shootingpatch = select_shooting_patch();
        form_factor_estimation(shootingpatch);
        distribution_of_unshot_energy(shootingpatch);
        step=step+1;
        print_results(outputfile,step);
    }
}
```

The multipleshooting algorithm will be

```c
main(inputfile, outputfile) {
    read_data(inputfile);
    subdivision_into_patches;
    step=0;
    while (not end()){
        shootingpolygon = select_shooting_polygon();
        form_factor_estimation(shootingpolygon);
        distribution_of_unshot_energy(shootingpolygon);
        step=step+1;
        print_results(outputfile,step);
    }
}
```

The difference between these algorithms is that in the first case, only one patch is shot while in the second one a polygon is chosen and all of their patches are shot at any iteration.

In the figure below, we show an example with two surfaces S1 and S2 subdivided into a number of patches. The dotted patches have low radiosities while the others have high radiosities. In the first alternative explained above, we would select patches from 1 to 16 because they belong to the same polygon, so taking advantage of coherence. We assume here that the unshot energy of polygon S1 is higher than the unshot energy of polygon S2. On the other hand, if we used the second strategy, we would choose sixteen non dotted patches, for example 1,2,5,6, from 9 to 16 and from 17 to 20. This selection would prevent
us from exploiting surface coherence but the algorithm would converge more quickly.

**Implementation**

From now on, we will explain how we implemented these ideas and what we got as a result.

First, we decided to adopt a simple way of subdivision. Every polygon is divided in a predetermined number of patches or a multiple of this number, which will be called NSUB. Therefore, the environment consists of NGROUPS groups of NSUB patches. These groups are not modified afterwards. The number of divisions of a polygon is decided by the user in the input data. Recall that one polygon may have more than one group. In our subdivision we only have three entities: objects, polygons and patches, although we are really working on the patches most of the time. In order to simplify, only four-vertex convex polygons are accepted.

Another word must be said about subdivisions. The first part of the programme reads the input data and identifies the objects and polygons of the scene, in addition of the number of groups of NSUB patches. It goes without saying that the information of every patch (area, center, etc) will require a NSUBxNGROUPS structure, whatever it is. Every kind of data is kept in different arrays: area (called patcharea), center (called patchcenter), and so on. The data of each of the NGROUPS groups is not kept together, that is in NSUB consecutive positions of such structure. Instead, the stride between two
of them will be \textit{NGROUPS}: the group \textit{j} of the environment will find its members in the positions \textit{j+i*NGROUPS} for \textit{i}=0,..,\textit{NSUB}-1, as figure 12 depicts:

![Diagram of group distribution](image)

\textit{Patch of group \textit{j}}

\textit{Fig 12}

This organization may appear confusing, but it will allow us to get better results. In general, \textit{NSUB} is likely to be smaller, typically 16, 32, 64,.., than \textit{NGROUPS}, which can be sizeable. As we will see below, the same layout will appear on dealing with the structures that form factor require. The routines of the vectorial computer will manipulate a bigger number of elements and, therefore, improve the final performance.

Second, we followed the idea of "shooting polygon", with a group of \textit{NSUB} patches laying on the same polygon and all of them is a group as a result of the subdivision. The selection of the group is based on the area and on the unshot energy of its members. It turns out to be as easy as the search of the shooting patch in the classical progressive radiosity.

We are dealing with a matrix on estimating the form factors and the unshot energy to be added to the environment. This fact will allow us to make use of the mechanisms offered by a vectorial computer.

We will focus on the funcions "select.shooting.polygon", "form.factor.estimation" and "distribution.of.unshot.energy".

"Select.shooting.polygon" finds the group of patches (belonging to the same polygon) with the highest value of the following formula:

\[ \sum_{j=1}^{NSUB} U E_{ij} A_{ij} \quad (i \text{ ranges from } 1 \text{ to } \textit{NGROUPS}) \]

where \( U E_{ij} \) is the unshot energy and \( A_{ij} \) is the area of patch \( j \) and group \( i \)

Once this has been pinpointed, the process goes on only if this unshot energy as a whole is not smaller than a threshold. Otherwise, the algorithm ends. This group will be called the shooting polygon from now on.
"Form_factor_estimation" is responsible for estimating the form factors. The dimension of the data structure will be NSUBxNSUBxNGROUPS:

![Diagram of data structure]

This layout will appear on the arrays we need.

In general, every loop of this function has the same layout a priori:

for (i=0;i<NSUB;i=i+1) for every shooting patch
  for (j=0;j<NSUB;j=j+1) for every division
    for (k=0;k<NGROUPS;k=k+1) for every group operation...

As we know, only the inner loop will vectorize, and this is the longest in this proposal. On implementing the code, some of the loops can be joined.

for (i=0;i<NSUB;i=i+1)
  for (j=0;j<NSUB*NGROUPS;j=j+1)

or even only one loop

for (i=0;i<NSUB*NSUB*NGROUPS;i=i+1)

And, when possible, they have been substituted by a routine of the vectorial computer.

Finally, "Distribution_of_unshot_energy" takes this matrix to shoot the energy of the shooting polygon. To obtain the energy that must be added to any patch, the unshot
energy of the patches of the shooting polygon must be multiplied by the form factor corresponding to that patch (a column of the matrix) and by its reflectivity. Finally, the values in the column must be added to the unshot energy and final radiosity of that patch and set to zero the unshot energy of the patches of the shooting polygon. After that, the main loop begins again.

In terms of matrices, this means a dot product between a column of the matrix and the column of unshot energies (of the shooting patches). This must be carried out for any column of our matrix:

Results

A number of trials were carried out to prove the proposed algorithm. Every experience consisted of a closed environment determined by a big cube and a predetermined number of smaller cubes allocated therein at random (10 cubes to work with 2496 patches and 16 to work with 3072 patches). Three samples of these environments were executed. In our application, the minimum number of patches in each polygon is 16, that is, NSUB is equal to 16.

The vectorized classical progressive radiosity has been executed to compare results. Both processes stop when an equivalent threshold of unshot radiosity is achieved.

The following charts show the results. As it can be seen, the number of shoot patches in the multiple shooting is similar to the number in the simple shooting (classical progressive radiosity) given that NSUB patches are shot at every iteration. This is given in the second column. At this point we should note that even though the number of shoted patches is similar in both algorithms, there are not exactly the same in both cases. This is due to the fact that, in the multiple shooting algorithm, patches are selected by groups.
Multiple shooting is worthwhile seeing the time charts. The first one shows the execution time for each kind of process. The time decrease is quite obvious, not only in this chart but also in the next one, where time is divided by number of iterations. The time of an iteration in the simple shooting represents the time we take to shoot one patch while, in the case of multiple shooting, it is the time required to shoot NSUB patches. We obtain the time for one patch dividing by NSUB in order to compare with the simple shooting. Each patch in the multiple shooting only needs between 0.107 and 0.214 seconds in the shooting process while it ranges between 0.792 and 1.245 in the simple shooting.

This improvement comes as a result of the vectorial behaviour of the proposed algorithm which supported by: a) the use of vectorial functions, which are usually highly optimized, b) the use of large matrices (vectors), which reduces the overhead per element in this structure when operating and c) the data distribution, to make the most of the former two, since memory access undergoes no delay.

Notes in the charts:

(1) The second column is the total number of patches shot in every multiple shooting experience obtained by multiplying the number of iterations by NSUB.

(2) The second column is the time required by a patch to be shot in the multiple shooting obtained by dividing the time of an iteration by NSUB.

<table>
<thead>
<tr>
<th>patches</th>
<th>simple shooting</th>
<th>(1) shot patches in multiple sh.</th>
<th>multiple shooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>2496</td>
<td>10,197</td>
<td>10,432</td>
<td>652</td>
</tr>
<tr>
<td>2496</td>
<td>8,509</td>
<td>8,368</td>
<td>523</td>
</tr>
<tr>
<td>2496</td>
<td>10,176</td>
<td>10,240</td>
<td>640</td>
</tr>
<tr>
<td>mean</td>
<td>9,627</td>
<td>9,680</td>
<td>605</td>
</tr>
<tr>
<td>3072</td>
<td>11,048</td>
<td>11,728</td>
<td>733</td>
</tr>
<tr>
<td>3072</td>
<td>9,390</td>
<td>9,136</td>
<td>571</td>
</tr>
<tr>
<td>3072</td>
<td>10,880</td>
<td>10,464</td>
<td>654</td>
</tr>
<tr>
<td>mean</td>
<td>10,439</td>
<td>10,442</td>
<td>652</td>
</tr>
</tbody>
</table>

Number of iterations

Fig 15
<table>
<thead>
<tr>
<th>patches</th>
<th>simple shooting</th>
<th>multiple shooting</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2496</td>
<td>9,518</td>
<td>1,302</td>
<td>7.31</td>
</tr>
<tr>
<td>2496</td>
<td>7,871</td>
<td>1,073</td>
<td>7.33</td>
</tr>
<tr>
<td>2496</td>
<td>8,051</td>
<td>1,101</td>
<td>7.32</td>
</tr>
<tr>
<td>mean</td>
<td>8,453</td>
<td>1,158</td>
<td>7.32</td>
</tr>
<tr>
<td>3072</td>
<td>13,650</td>
<td>2,516</td>
<td>5.42</td>
</tr>
<tr>
<td>3072</td>
<td>11,493</td>
<td>1,929</td>
<td>5.95</td>
</tr>
<tr>
<td>3072</td>
<td>13,553</td>
<td>2,226</td>
<td>6.08</td>
</tr>
<tr>
<td>mean</td>
<td>12,698</td>
<td>2,223</td>
<td>5.81</td>
</tr>
</tbody>
</table>

Time in seconds

Fig 16

<table>
<thead>
<tr>
<th>patches</th>
<th>simple shooting</th>
<th>time per patch</th>
<th>multiple shooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>2496</td>
<td>0.933</td>
<td>0.124</td>
<td>1.996</td>
</tr>
<tr>
<td>2496</td>
<td>0.925</td>
<td>0.128</td>
<td>2.051</td>
</tr>
<tr>
<td>2496</td>
<td>0.792</td>
<td>0.107</td>
<td>1.720</td>
</tr>
<tr>
<td>mean</td>
<td>0.833</td>
<td>0.119</td>
<td>1.922</td>
</tr>
<tr>
<td>3072</td>
<td>1.235</td>
<td>0.214</td>
<td>3.432</td>
</tr>
<tr>
<td>3072</td>
<td>1.223</td>
<td>0.211</td>
<td>3.378</td>
</tr>
<tr>
<td>3072</td>
<td>1.245</td>
<td>0.212</td>
<td>3.403</td>
</tr>
<tr>
<td>mean</td>
<td>1.234</td>
<td>0.212</td>
<td>3.404</td>
</tr>
</tbody>
</table>

Time in seconds / Number of iterations

Fig 17
6. Conclusion and future work

We have proposed a radiosity algorithm specially suited for vectorial computers based on the concept of multishooting. This has, in a way, been previously used in parallel MIMD and SIMD architectures but, as far as we know, it has not been introduced and experimented in vectorial computers.

The proposed technique has shown to be efficient and reduces the execution time significantly with respect to an optimized standard progressive radiosity algorithm. So, the improvement with respect to a non vectorized radiosity algorithm would be better.

Our future work in this area will focus on studying other criteria of selection of the shooting set of patches and to extend the multishooting algorithm to environments where the subdivision of polygons into patches is not predetermined but adaptative.

Acknowledgements

The authors gratefully acknowledge Salvador Clavé, Project Director of CESCA, for his valuable help in the development of this research.

This work was supported by grants from the Centre de Supercomputació de Catalunya (CESCA) and Universitat Politècnica de Catalunya (UPC) and it was partially developed under grant TIC92-1031-C02-01 from CICYT of the Spanish Government.

References


Recent Research Reports


LSI-93-49-R “A class-based approach to learn appropriate selectional restrictions from a parsed corpus”, Francesc Riba.


LSI-94-2-R “Degrees and reducibilities of easy tally sets”, Montserrat Hermo.

LSI-94-3-R “Isothetic polyhedra and monotone boolean formulae”, Robert Juan-Arinyo.

LSI-94-4-R “Una modelización de la incompletitud en los programas” (written in Spanish), Javier Pérez Campo.


Copies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secretls@lsi.upc.es