Interactive Relighting of Dynamic Refractive Objects

Implementation Specification

应哲敏
3120101966
Computer Science
yingzhemin@hotmail.com

巩炳辰
3120000140
Computer Science
gbc@zju.edu.cn

June 2015

Abstract

In this documentation we just specify how we implemented the relighting of dynamic refractive objects as described by Sun, X. et al.[2008].[1] All credits go to the authors of the paper. Thanks to thier working.

As the fancy paper was published in 2008, situations have changed a lot. We altered some methods used in the original pipeline. So in our implementation, more workload is done on GPU, which results in a more coherent pipeline.

1 Introduction

1.1 Original pipeline

We briefly describe the original pipeline here, for the convenience of describing our implemented pipeline. The complete original pipeline is depicted in Figure 1.

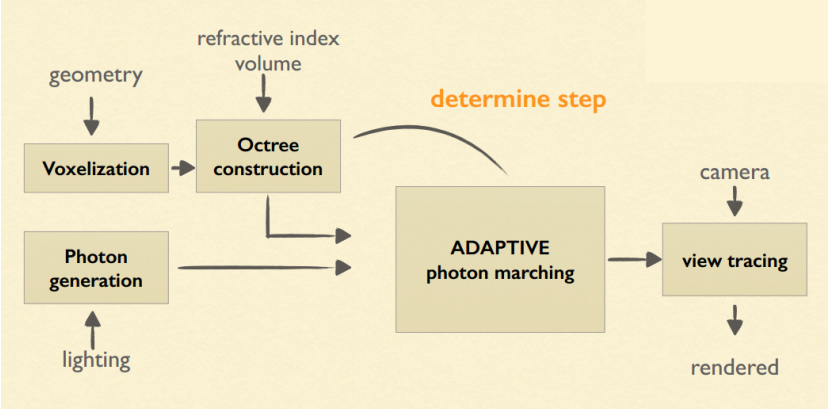


Figure 1: Original pipeline

1. Voxelization

   Converts triangle-mesh surfaces into volumetric data. As the pipeline traces photon path inside the mesh, a voxel-based representation of the index of refraction n, the scattering coefficient σ, and the extinction coefficient κ is required.

2. Octree construction

   Analyzes the index of refraction data and produces an octree describing the regions of space in which the refractive index is nearly constant.

3. Photon generation

   Generates photons with initial positions and diretions, which will traverse the scene later.

4. Adptive photon marching

   Advances each photon along its path through the scene. Step length is chosen adaptively using the formerly constructed octree.

5. Radiance storage

   Deposits radiance into all the voxels that each photon traverses.

6. Viewing pass

   Renders an image by tracing viewing rays from the camera into the scene and calculating the amount of radiance that reaches the camera along each ray.

1.2 Technical details

All the data used in this documentation are obtained in an environment as follows.

- Model: Stanford Bunny with 5002 faces and 2503 vertices.
- Rendered resolution: 1024x576
- Voxelization resolution: 128x128x128
- Generated photons: 20000 - 70000
- GPU card: NVIDIA GT 750M
- Development kits: Visual Studio 2013, OpenGL 4.3, CUDA 7.0
- Operating system: Microsoft windows 8.1 Pro 64 bit

2 Our Implementation

2.1 Voxelization

2.1.1 Voxelpipe

In the original pipeline, voxelization is accomplished by rendering the mesh in a sliced schema. We tried this method, but failed to do this fast enough.

We aimed to accomplish the voxelization in less than 100 ms. Suppose we have 100 ms, then for a voxelization procedure with resolution 128x128x128, we have to render this mesh in an extremely high frame rate up to 1280 fps, without other processing. In our experiment, we achieved to render a planar 500x500 picture at 400-500 fps. And moreover, we do not have a GPU rasterizer, we have to rely on the rendering pipeline of OpenGL and this requires additional interoperation between OpenGL and CUDA, which slows down the voxelization a bit more.

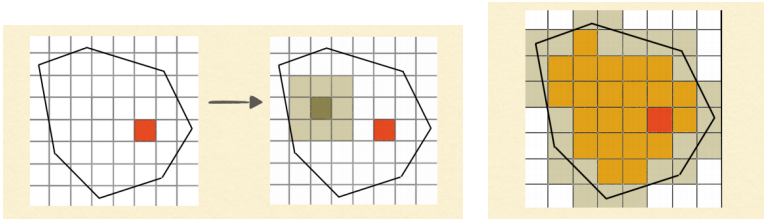
So we turned to other voxelization methods. We found a seemingly powerful tool on NVIDIA Reseach website which claims to be programmable. Roughly speaking, this method finds all voxels overlapped by each triangle on the mesh surface and then leverages the fast radix sorting on GPU to get the voxelized mesh. The details of this 3D voxelization method is described by Jacopo Pantaleoni [2011].[2] We obtained the source code via Google Code. Unfortunately, the author ceased maintenance in 2011, when the version of CUDA was 4.x. We have to modify and adapt the codes to newer version of CUDA.[1]

[1] Related code: FILE: src/voxelization.cu, LINE: 386-701

2.1.2 Voxel hull filling[2]

What was more unfortunate, Voxelpipe only gave us a hollow voxelized surface, but not a solid one. So we decided to fill the mesh. We tried scanline and path finding, both resulted in unsatisfying outcomes. Finally, we devised a diffusion simulation based filling algorithm.

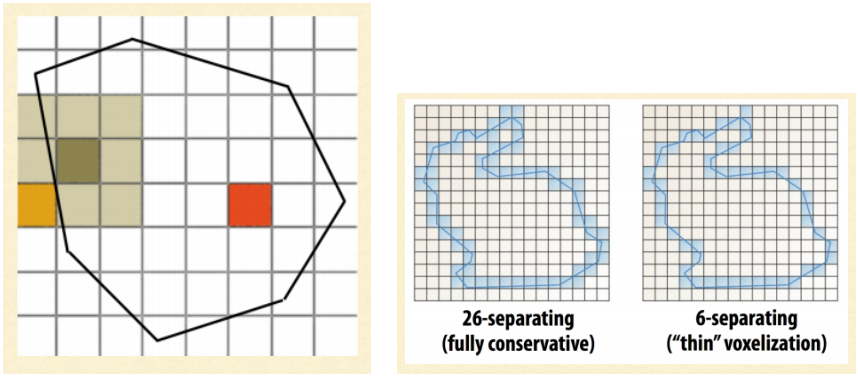
We require the user to choose one internal point of the mesh, like the paint bucket tool in image editing softwares. Alternatively we can design algorithms to find one voxel , which is much more easier then finding all, and frees the users from the choosing operation. We label the chosen internal one by 2, and the formerly obtained voxels by 1, and in the meanwhile, others by 0. In every loop, a 0-labeled voxel looks into the 26 neighbours near it. If it finds at least one 2-labeled neighbour, it labels itself by 2 and exit the loop. Not soon, all internal voxels will be labeled by 2. This procedure is depicted in Figure 2(a) and 2(b).



(a) Choosing and labeling (b) Filled voxels

Figure 2: Filling

In this algorithm, one vital problem is leaking (Figure 3(a)). We solved this problem by using a 26-separating voxelization (Figure 3(b)). And note that, this algorithm exits when the total 2-labeled voxels stops to increase or the loop number reaches a top bounding limit (the half dimension of the voxelized mesh, $\frac{128}{2}$ in this case).



(a) Leaking (b) 26-separating

Figure 3: Leaking and solution

[2]Related code: FILE: src/voxelization.cu, LINE: 192-234

In fact, this algorithm takes $O(n^4)$ time on CPU, but thanks to the warp based threads scheduling feature of CUDA library, one voxel does not stay in an invalid loop for long. We found the filling takes only $\frac{1}{3}$ time of the voxelization time and converges pretty early. The performace is shown in Figure 4 in *ms* unit.

Figure 4: Filling performance

2.1.3 Smooth index of refraction

In the original pipeline, to get smooth variation near the surface of the mesh, it adopts a rather complex super-sampling and down-sampling procedure. We noted that there must be one input into the pipeline. So we did not store the index of refraction in the voxels. In stead, we referd to the original input in later stages in the pipeline. After all, the original input is the most precise one we can get.

And also, we used a so called *refraction map*(Figure 5) in our early development and later we adopted analytic expressions. We found analytic expression is flexible enough to get the desired refraction distribution. And two of its overwhelming advantages are: it is perfectly precise; it is easy and fast to calculate the gradient.

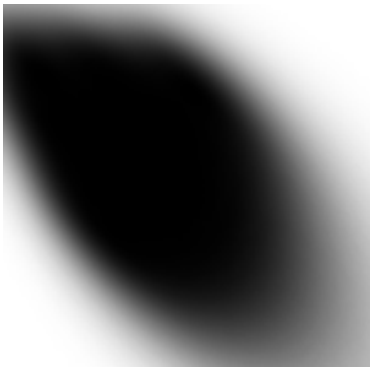


Figure 5: Refraction map

2.2 Octree construction[3]

Our implementation of the octree is totaly the same as the paper describes. What need to be specified is the data storage, since other implementation details are straightforward. The octree is stored in linear layout in the VRAM. This allows us to access any node on any level according to several easily deduced equations. And the numbers are almost all exponentials of 2, which allows us to optimize the codes using bit operations.

Some equations and implementations are as follows.

$$\text{total\_size} = \frac{1}{7}(2^{3*logN+3} - 1)$$

```
int totalsize = sizeof(struct Node)*((1 << ((log_N+1)<<2 + (log_N+1))) - 1) / 7;
```

$$\text{grid\_on\_level\_n} = 2^{3*(logN-n)}$$

```
    threads = (1 << ((log_N - level)<<1 + (log_N - level)));
```

$$\text{offset\_level\_n} = \frac{1}{7}(2^{3*(logN+1)} - 2^{3*(logN-n+1)})$$

```
__host__ inline int GetLevelOriginOffset(int level){
    if (level == 0)
        return 0;
    else if (level > log_N)
        return -1;
    else
        return ((1 << ((log_N + 1)<<1 + log_N + 1))
            - (1 << ((log_N - level + 1)<<1 + log_N - level + 1))) / 7;
}
```

In level n, a voxel with offset *offset*:

$$\begin{aligned} dim &= 2^{N-level} \\ x &= \frac{offset}{dim^2}\%dim \\ y &= \frac{offset}{dim}\%dim \\ z &= offset\%dim \\ \text{level\_origin} &= 8*x*dim^2 + 4*y*dim + 2*z \end{aligned}$$

```
    int x = (id / dim / dim) % dim;
    int y = (id / dim) % dim;
    int z = id % dim;
    int base = (x*dim*dim)<<3 + (y*dim)<<2 + z<<1;
```

[3]Related code: FILE: src/photon_inline.h, LINE: 143-396

2.3 Photon generation[4]

As the paper proposes, photon generation is much similar to shadow map.

We set up an additional frambuffer in OpenGL with color buffer and depth buffer. The camera is put at the exact position of the light, and only the mesh is drawn. As we enabled the depth test, only the faces directly seen by the light will be drawn in the final framebuffer. Here, the small trick is we kept the world coordinates of each drawn vertex in the vertex shader and passed it to the fragment shader. In fragment shader, we used the three RGB channels of color to store the XYZ coordinates. The related GLSL program and the resulted texture are as Figure 6 and Figure 7 show.

```
Vertex shader:
    fs_meshpos = u_mMatrix * vec4(v_position, 1.0);

Fragment shader:
    frag_color = vec4(fs_meshpos.xyz, 1.0);
```

Figure 6: Shaders in photon generation

Figure 7: Rendered texture with world coordinates

World coordinate to color texture rendering is done by OpenGL and later the texture is scanned to generate the photon list. To reduce the number of photons generated, the photons that may not traverse the mesh should be discarded. We only count the colored pixels. When scanning, pixels may be interpolated to produce more photons if the total amount is small or may be down sampled to cut down the total amount if the photons are too many. Together with the position and radiance of the light, we can get the final photon information list with initial position, direction and radiance.

[4]Related code: FILE: src/photon_inline.h, LINE: 1242-1406

2.4 Adptive photon marching

2.4.1 Photon march[5]

To march the photons, we followed the ray equation of geometric optics.

$$\begin{aligned} x_{i+1} &= x_i + \frac{\Delta s}{n} v_i \\ v_{i+1} &= v_i + \nabla s \Delta n \end{aligned}$$

These equations are straightforward and can be easily turned into codes.

2.4.2 Step determination[6]

When determine the step length in photon marching, we encountered some problems to do with the choice between discrete and continuous. To take advantage of the octree, one may let the step length be one or several times of the voxel size. But the direction is hard to determine. If the direction is continuous, then due to discrete float number rounding, the direction varies in a jumping form. And if the direction is discrete, we can use only 26 directions. Both these two situations result in unsmooth results in our experiment.

So we adopted a continuous way for both the step length and the direction. We still leveraged the constructed octree, but only to obtain the largest size of the region in which the index of refraction is almost constant. Then in this region, we calculated the intersection point on one of the region surface with the photon marching ray. Given the direction, one ray can only intersect with three surfaces of the cube, so we used an enumerating calculation as follows.

for: known_index = 0 : 2

$$\begin{aligned} pos\_next_{known\_index} = origin_{known\_index} \\ + Sign(direction_{known\_index} + 1) * region\_size * \frac{1}{2} \end{aligned}$$

for: not_known_index = (known_index + i)%3, i = 1, 2

$$\begin{aligned} pos\_next_{not\_known\_index} = \frac{pos\_next_{known\_index} - pos_{known\_index}}{direction_{known\_index}} \\ * direction_{not\_known\_index} \\ + pos_{not\_known\_index} \end{aligned}$$

We was anxious about the intersection calculation taking long time. But it turned out to be fast. In one test, it measured 283,366 ns, as opposed to the calculation of the index of refraction which took 174,550 ns.

After the calculation of intersection, we used the length of position-intersection vector as marching step length. But this is not the final. This length is compared with the user provided step lower or upper limit and a default lower limit(one voxel). Finally, we multiplied the length with about 1.1 to push the photon a bit further out of the current region.

[5]Related code: FILE: src/photon_inline.h, LINE: 920-946
[6]Related code: FILE: src/photon_inline.h, LINE: 602-700

2.4.3 Rasterization and radiance storage[7]

The original pipeline adopts a rather complex workaround here, due to the unavailability of atomic operations in CUDA and disability to rasterize 3D texture in OpenGL. After refering to the documentations of CUDA, we noticed that GPU of compute capability 2.x or above does have an atomic operation on float numbers.

So we wrote our line rasterizer. The rasterizer is quite naive but is adequate to the line segmentation task. We found the main direction of the line, along which the line goes through most voxels. Then we step through the voxels along this direction, storing values in each traversed voxel.

2.5 Viewing pass

Viewing pass is much similar to adaptive photon marching. The difference is the camera is set at the position of eyes and step length is not adaptive but fixed. Some other implementation details are in the following sections.

2.5.1 Resulted textures[8]

As CUDA can not directly render the results on the display, we have to store the results in several textures and transfer them to OpenGL. We used four textures: color texture for environment map, radiance texture for environment map, color texture for traced view ray and radiance texture for traced view ray. Except the first one, the other three are copied from CUDA to OpenGL to display. Texture mixup is done in fragment shader because in this way we can omit lots of pixels that do not come from the textures.

The reason we distinguished color and radiance is that human eyes are more sensitive to luminance information. So we can take more control on the luminance information by seperating it from color. Another reason is that radiance results must be normalized according to the amount of photons, but the color results need not.

In the fragment shader, Gaussian blur is applied to make the results more smooth.

2.5.2 Single scattering[9]

We treated the single scattering inside mesh and in air as Mie Scattering. The intensity of scattered light is calculated as proportional to viewing angle cosine value.

$$I(\theta) = I_i(1 + cos^2\theta)$$

2.5.3 Color mixing

Color mixing includes two aspects. One is mixing two colors, and another one is mixing color with luminance.

[7] Related code: FILE: src/photon_inline.h, LINE: 762-901
[8] Related code: FILE: src/photon_inline.h, LINE: 1409-1468
[9] Related code: FILE: src/photon_inline.h, LINE: 883-901

Colored light goes through one transparent object and then through another transparent object, what is the resulted color of the light? This problem is actually hard to solve. We first refered to a dedicated paper on this topic, but later found it is too complex.[3] Our second trial was the subtractive color system CMYK, and found the performance suffered. So at last, we just took the minimum of every RGB channel. This is coarse treatment but the result is acceptable.

Color and luminance mixing is more systematic. We converted the RGB value to HSL color space. In HSL color space there is a luminance channel and we just increased or decreased the L channel by some degree. The color is converted back to RGB color space for display. Color and luminance mixing is done in fragment shader.[10]

2.6 Optimization

To improve the frame rate and provide a more smooth interaction, we spent much time on optimization. Two main techniques are using local memory more and making less function call.

2.6.1 Local memory and global memory

If we transfer data from CPU to GPU using *cudaMemcpy* or allocate memory using *cudaMalloc*, the related memory resides in global memory. For every thread, it has its own local memory and local registers. Copy the data from global memory first and operate on the data locally and then copy the data back to global memory, this is much faster than operaing on the global memory directly. Operation and performance differences can be seen in Figure 8 and in Figure 9 (unit is ns). We see the performance is nearly doubled.

```
loop:
    *a = *a + i;
```

```
int tmp = *a;
loop:
    tmp = tmp + i;

*a = tmp;
```

Figure 8: Operate globally and locally

```
initialization: 1,2700
while initialization:
14,4700
intersection: 18,2700
get n: 13,1654
get ds: 7,6608
get gradient: 7,3922
march: 6,5182
rasterization: 5,0214
update: 6,0794
```

```
initialization: 1,4000
while initialization: 7,0572
intersection: 9,5576
get n: 5,8256
get ds: 5,2248
get gradient: 5,8710
march: 5,6066
rasterization: 4,8880
update: 5,4352
```

Figure 9: Preformance difference between two kinds of operation patterns (ns)

[10]Related code: FILE: src/shaders/default.frag, LINE: 57-166

2.6.2 Make less function call on device

We found function calls on device are expensive. Even the function is inlined, it seems that the compile forgets the *inline* keyword. A small experiment is as Figure 10 shows.

```
a = a + 1
    just add: 516
    add by call: 62438
    add by call inline: 67670
```

Figure 10: Preformance difference between three kinds of Add operation patterns (ns)

So we replaced all function definitions on device with macro definitions. This optimization cut down $\frac{1}{3}$ time.

3 Results

Here we present some screenshots of the rendered results.
The Stanford bunny:

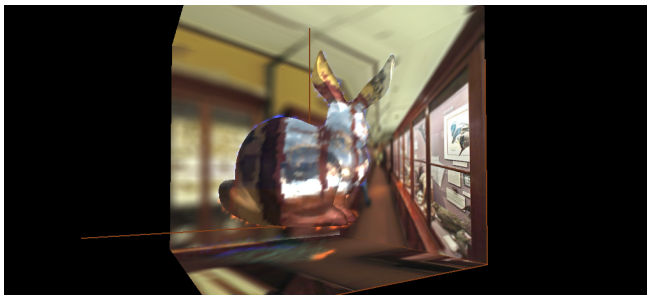


Figure 11

Two balls:

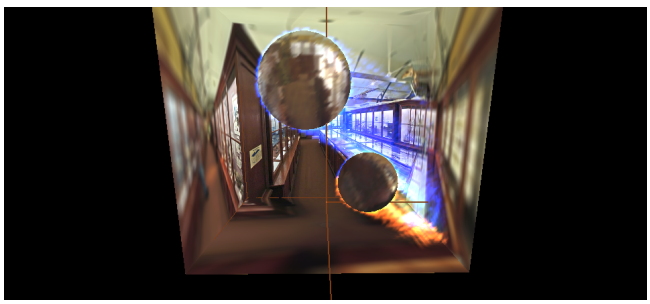


Figure 12

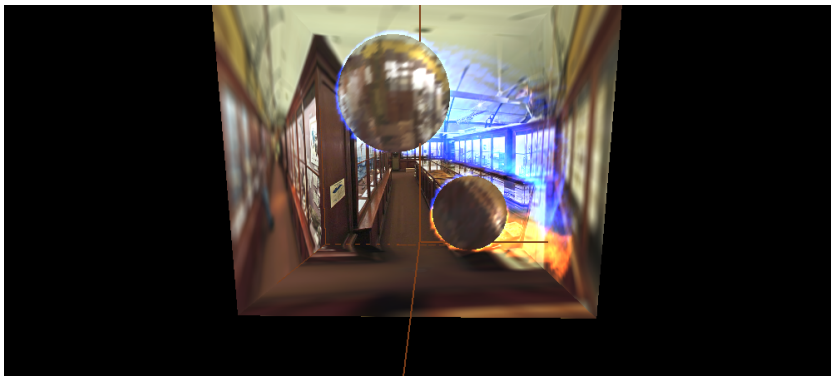


Figure 13

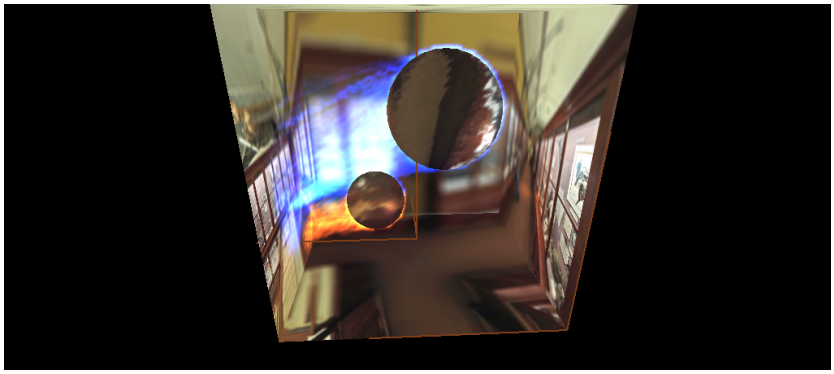


Figure 14

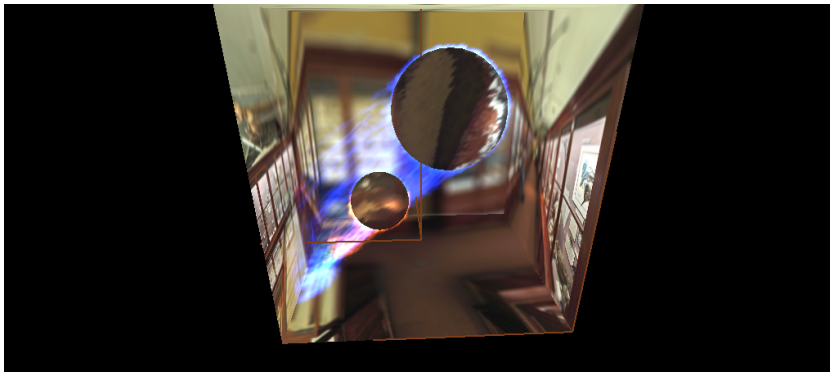


Figure 15

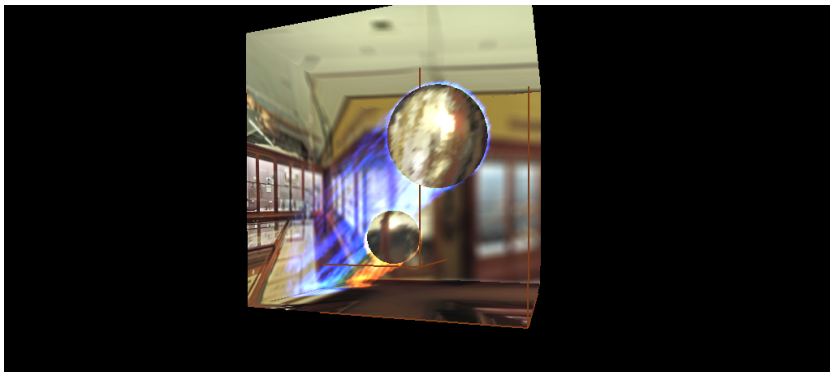


Figure 16

Unfortunately, our visual results are not as good as the paper presents.

The frame rate is about 8-12 fps on the machine specified formerly. As the equipments are much different from those used by the authors of the paper, it is meaningless to compare performance results.

4 Conclusion

In our implementation, only photon generation and rendering is done by OpenGL. Others are done by CUDA. This reduces data transfer dramatically.

Though we implemented the whole pipeline, our rendered result is not so satistying visually. We think much working need to be done to improve the visual results.

References

[1] Sun, X., Zhou, K., Stollnitz, E., Shi, J., Guo, B., "Interactive relighting of dynamic refractive objects," *ACM Trans. Graph. 27, 3*, August 2008.

[2] Jacopo Pantaleoni, "Voxelpipe: A programmable pipeline for 3d voxelization," *High Performance Graphics*, August 2011.

[3] CHET S. HAASE and GARY W. MEYER, "Modeling pigmented materials for realistic image synthesis," *ACM Trans. Graph. 11, 4*, 1992.