Hierarchical View-dependent Structures for Interactive Scene Manipulation

Normand Brière and Pierre Poulin

Département d'Informatique et de Recherche Opérationnelle *
Université de Montréal

Abstract

The result of a scene manipulation is usually displayed by re-rendering the entire image even if the change has affected only a small portion of it. This paper presents a system that efficiently detects and recomputes the exact portion of the image that has changed after an arbitrary manipulation of a scene viewed from a fixed camera. The incremental rendering allows for all visual effects produced by ray tracing, including shadows, reflections, refractions, textures, and bump maps.

Two structures are maintained to achieve this. A *ray tree* is associated with each pixel and is used to detect and rebuild only the modified rays after an optical or geometrical change. A *color tree* represents the complete color expression of a pixel. All changes affecting the color of a pixel without changing the corresponding ray tree require only re-evaluation of the affected portions of the color tree.

Optimizations are presented to efficiently detect the modified structures by the use of strategies such as grouping similar information and building hierarchies. Pruning and weighted re-evaluation of information are also considered to manage the memory requirements.

The incremental rendering is done efficiently and accurately and is suitable in an interactive context.

CR Categories and Subject Descriptors: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.
Additional Key Words and Phrases: scene editing, interactive system, rendering, image quadtree, color tree, ray tree

1 Introduction

Computing the image of a 3D synthetic scene is a complex process, especially when shadows, textures, bump maps, reflections, and refractions are desired. A simple yet powerful algorithm producing such effects is ray tracing [17]. Unfortunately, its computational requirements are generally too high to be considered suitable to calculate intermediate images resulting from an interactive scene manipulation. In fact, ray tracing is mostly used for high quality final rendering, ranging from minutes to hours of computing.

*C.P. 6128, succ. Centre-Ville, Montréal (Qc) Canada H3C 3J7 { briere | poulin }@iro.umontreal.ca

One important reason why ray tracing is rarely considered within an interactive tool is that most interactive systems recompute the entire image of the scene without considering any incremental rendering from the previous image. The fastest rendering technique is usually the projection of wireframe models without any line removal. Unfortunately, it can only convey information about shape, and complex objects are difficult to interpret. The Z-buffer algorithm is quite fast and treats visible-surface determination. It becomes even more competitive by integrating the technique with hierarchical models and treating the image hierarchically [7]. Shading and shadowing for directional and point light sources, as well as textures and filtering can be simulated [3] [18] [14], but with a significant impact on performance and memory usage. However, other important visual phenomena such as reflection and refraction can only be approximated by textures, and this with great effort and potential artifacts [5].

For interaction, one can also benefit from re-ordering the rendering with respect to the phenomena being manipulated, or from choosing between different levels of object complexity [4]. In specific contexts, some manipulations have been optimized by preprocessing. Hanrahan and Haeberli [10] edit material properties on a preprocessed sphere. Interactivity is obtained but the manipulation is done from a different visual context than the scene itself. Litwinowicz and Miller [12] interactively distort a texture directly on a preprocessed projection of the uv coordinates of the texture parametrization.

Instead of rendering the entire scene, other researchers have considered updating only the elements affected by a change. For each surface, Cook [2] conserves in a tree structure (shade tree) the symbolic evaluation of the illumination model. If a manipulation does not modify the shape of the tree, then the local illumination may be updated by simply evaluating the shade tree. The evaluation of a tree may not be faster than the calculation itself, but if the tree is replicated at each pixel and compressed according to the parameter currently adjusted, then the evaluation of the smaller tree is faster. By expressing a RenderMan shader by source code instead of a symbolic tree, Guenter *et al.* [8] have defined specialized shaders.

In these techniques, the preprocessing is usually applied on the first visible surface. Séquin and Smyrl [15] preserve in a tree the color expressions of all intersections obtained by ray tracing. The image is updated by traversing these color trees with modified parameters. They only consider changes that do not alter the shape of the trees, which leads to significant time savings because this avoids recomputing visibility. They also propose several compression techniques to reduce the memory usage and improve on the display time. Systems such as *Atlantis* from Abvent and *IPR* from Wavefront extend these color trees by adding a fixed number of extra ray generations to allow a user, for instance, to make reflective a previously non-reflective object.

Murakami and Hirota [13] extend these previous techniques to handle also changes in visibility for a scene rendered from a static viewpoint. A ray is indexed by the list of regular voxels it traverses. Any change to the scene is associated with its affected voxels, which in turn determine the potentially affected rays. They also

Color Parameters
surface color
ambient, diffuse, specular coefficients
surface roughness
proportion of reflection, refraction, transparency
proportion plastic-metallic
light color and intensity
texture parameter
bump map
surface parametrization
Optical Manipulation
add/delete a reflection, refraction and transparency attribute
change a refraction index
add/delete/change a bump map of a reflective or refractive object
Geometrical Manipulation
add/delete/transform an object
add/delete/transform a light source
add/delete/transform a displacement map

Table 1: Color and ray tree dependent manipulations

use a clever hashing scheme to identify quickly the rays affected by a given voxel. However the visibility determination is performed with respect to the affected voxels rather than the transformed objects. Therefore, all intersections between a ray and the objects in the affected voxels must be precomputed and saved, or recomputed each time a voxel is affected. Increasing the number of voxels reduces this visibility determination, but at the cost of storing many voxels, and also of handling more entries in the hashing table. Jevans [11] removed the previous dependency upon image resolution by storing instead in each voxel the identification of limited regions potentially affected by this voxel. However more unaffected rays can thus be wrongly identified as affected, and the visibility is done with respect to the entire scene.

In this paper, we present a system for the manipulation of a scene viewed from a fixed camera. The visual effects in the images of the scene can include the richness of all those produced by ray tracing, including textures, shadows, reflections, refractions, and bump maps. The central concept behind this system is the ability to efficiently detect and recompute only the modified image portion. Allowed changes are of any kind, whether modifying a surface shading parameter, a texture, making a surface reflective or refractive, or transforming the geometry of any object.

To achieve this, some concepts from the systems described above are unified into two tree-like structures: the *color tree* and the *ray tree*. In the next section, we describe these two structures and how they are used. In section 3, efficiency issues are addressed for the detection and updating of these structures. In section 4, we consider trade-offs between memory requirements and computing efficiency. Finally, we give some typical results from using our system and conclude by summing up our work and by listing potential extensions and applications.

2 Scene Manipulation

Changing some portion of a synthetic scene can affect different properties treated at various stages of the rendering algorithm. To efficiently update the image affected by a specific change may thus require more than one data structure. All possible changes can be divided into two categories. A *color change* may modify the color of at least one pixel without affecting the scene visibility, while a *ray change* (optical or geometrical) may change this visibility.

In an interactive context, the user selects a group of objects (*selection*) and applies a given change to it. In the next subsections, we explain the particularities of each manipulation and describe the two structures used to incrementally render the current image. Please refer to figure 1 for a graphical representation of the concepts described in this section.

2.1 The Color-tree Structure

In ray tracing, the color of a pixel is computed by following a ray through the scene. At each intersection, the color returned depends upon the surface reflection/refraction model which specifies how much of the light reaching this point is sent back along the ray direction. We store the color of a pixel as an expression tree (figure 1 color tree) in which each leaf is a constant or a pointer to a parameter, and each node is an n-ary function. If a color subtree does not contain any parameter, it is replaced by a constant leaf.

One can view our color tree as a union of concepts such as shade trees [2], texture trees [16], and parametrized rays [15]. Indeed, a color tree corresponds to the entire symbolic evaluation of a pixel color without preserving information about the visibility.

A color change is the most simple and efficient change to handle. It corresponds to a change that does not alter the visibility in the image. Re-rendering the modified scene consists of simply re-evaluating the color expression for each pixel. The refresh rate therefore depends upon the number of nodes in each color tree, which is a function of the number of light sources and interreflection/interrefraction combinations. In most cases, the display time is uniform and quite fast. All the shading parameters listed in the top section of table 1 are associated with color changes.

Any shading parameter may be a variable or more generally, a texture function controlling this parameter. At a given intersection point, such a texture function may depend upon that point or upon the surface parametrization at that point. Since a texture function can be expressed as a tree structure, it is integrated as a subtree of our color tree. A change of any texture parameter is thus also resolved by evaluating the color tree.

Finally, if the user manipulates a higher-level property such as changing a procedural texture to another, or changing the illumination model itself, the associated color subtrees are appropriately replaced.

2.2 The Ray-tree Structure

In order to compute visibility changes efficiently, an extended structure is used, namely, the ray tree. It is similar to Murakami and Hirota's ray set [13], but without information related to voxels. The *ray path* of a pixel is the ordered list of objects encountered by a ray originating from the eye position through this pixel. The ray tree of a pixel (figure 1 ray tree) is the geometrically-specific information of the ray path. Each node of the ray tree contains intersection point dependent information (PDI): the intersection point, its normal, and surface parametrization. Nodes for the eye and for each light are considered global and are not duplicated. A *ray segment* joins two consecutive points of a ray tree. A node of a ray path is only a pointer to an object from the list of objects defining the scene. Rebuilding a portion of a ray tree causes an immediate update of the corresponding color tree.

An *optical change* occurs when a ray tree is modified without changing the scene geometry. The middle section of table 1 lists some optical changes. An optical change is effective at a node when its object pointed to has been modified by an optical change. The intersection point at this node is guaranteed to be valid. However new reflected or refracted rays may have to be re-shot from this point and intersected with respect to the entire scene.

A *geometrical change* occurs when the geometry of the scene has changed. Some of these changes appear in the bottom section of table 1. A general geometrical change is handled in three steps. First the selection is removed from all ray trees. The resulting ray trees

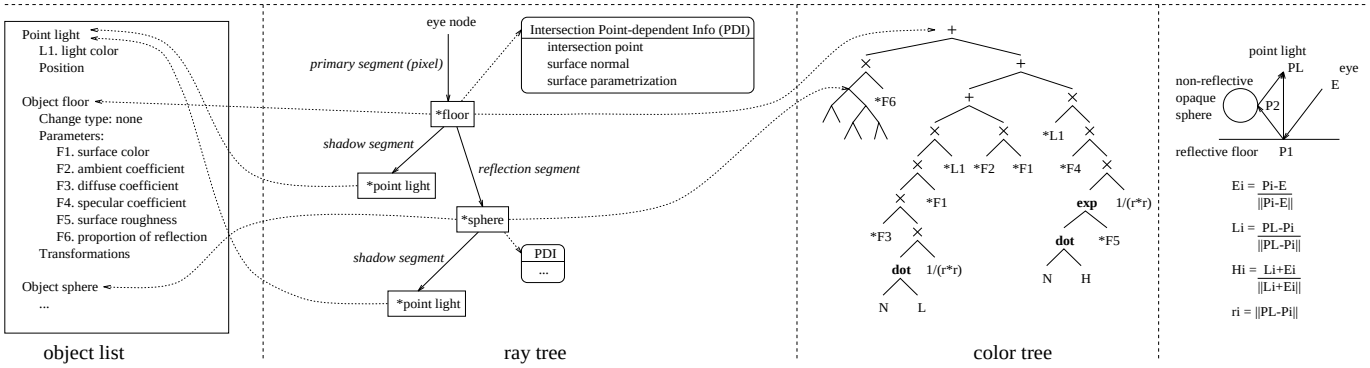


Figure 1: Color and ray trees

will not contain any reference to the selection. Then only the selection is rendered while it is manipulated by the user. Finally when the user releases the selection, the ray trees are updated. A *ray test* is a visibility determination with respect to the selection only. A *ray update* is a visibility determination with respect to the entire scene (except for primary rays which need only a ray test), but only for the affected pixels. This is typically a small fraction of the image.

To remove the selection from the ray trees, all of their nodes are visited. If the corresponding object at a node is contained in the selection, then the entire subtree is no longer valid. It is replaced by a new ray subtree traced from the intersection point of the previous node in the same direction as the incoming ray segment. The intersections are computed with respect to the entire scene, but without the selection itself.

The intermediate rendering does not modify the ray trees in any way since at that time they do not contain any reference to the selection. All ray trees are simply traversed and only intersection tests between the segments and the selection are performed. If a ray segment intersects the selection then the intersection point is computed and all emanating rays (shadow, reflection, refraction) must be reshot from that point with respect to the entire scene, including the selection.

Adding the selection to the ray trees corresponds to rendering it only once while updating the modified ray trees.

3 Hierarchies and Groups

The color and ray trees allow incremental ray tracing of the scene during manipulation. In most cases, this is already an appreciable gain because a structure test is much faster than its update. Also, the visibility determination is typically the most expensive operation and none are performed for a color change.

Unfortunately, traversing all structures to determine what may be affected quickly becomes a bottleneck. In this section, we propose some optimizations to cull many unnecessary tests. Section 5 will demonstrate the efficiency of these optimizations.

3.1 Selection-dependent Preprocessing

The image represented by its color and ray trees is built as a *quadtree*. Typically, the user adjusts the scene by applying several consecutive color or optical changes to the same selection. The quadtree hierarchical structure can greatly speed up such subsequent changes. Indeed, because the selection does not change, the portion of the image dependent upon the selection remains the same. A flag is kept in each *region* (quadtree element) that indicates if the region is dependent or not upon the current selection. The preprocessing is performed at the first change and the flags are not reset as long as the user applies color or optical changes to the same selection. Because of the hierarchical nature of the quadtree, the rejection of independent regions is very fast.

Figure 2 b) shows with white contours the regions dependent upon the central sphere: the primary ray segments intersecting the central sphere, the interreflections on the floor and the left sphere, and the refraction in the right sphere. Here, only shadows from opaque objects are handled. Note that this preprocessing is also applicable for removing the selection from the ray trees.

3.2 Groups and Atoms

It is also possible to speed up the selection-dependent preprocessing by using the concept of groups. A *group* is a region for which each ray path is identical. As such, their corresponding ray trees only differ by their PDIs, and the subregions of a group are also groups themselves.

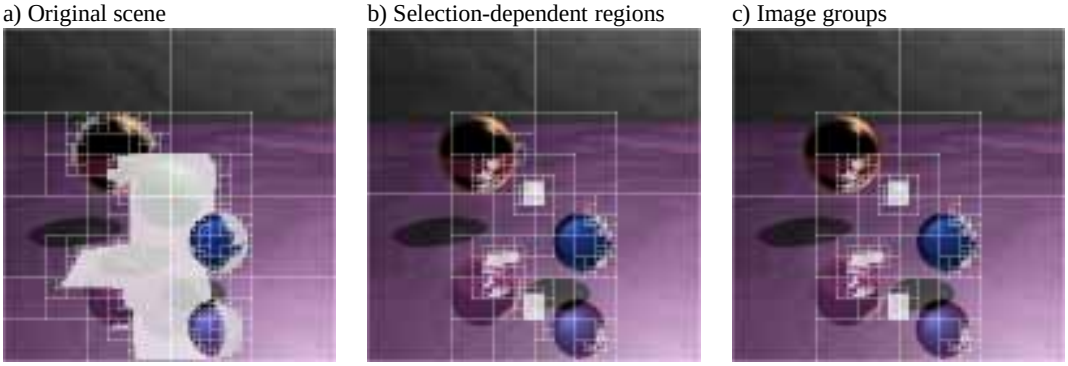
A flag in each region indicates if the region is a group by a pointer to its corresponding ray path. If the region is a group, only the ray path needs to be tested. After optical and geometrical changes, ray paths may change and therefore some image groups must be updated. However, because these changes are propagated from the bottom of the quadtree to its top, the number of regions to be updated is minimal. Figure 2 c) shows with white contours the groups in the image.

In a scene composed of many visible objects, the number of groups can be rather large and their efficiency thus much reduced. If the scene is constructed as a hierarchy of objects, it is possible to consider an *atom* to be at a higher level than the leaves in the model hierarchy. By raising the atom in the hierarchy, the number of groups and ray paths should globally decrease, but if the user tries to modify a subatomic object, the system will have to consider the entire atom as modified. For instance in figure 4, the bust of Beethoven, boat, individual trees, and cows could each be considered an atom.

Many regions contain typically a limited number of ray paths, but because of the distribution of these ray paths within a region, the quadtree subdivision might produce a large number of smaller groups, each in the worst case consisting of a single ray path. To reduce these situations and decrease further the number of groups, the groups are generalized to *path groups*, p-groups, where p indicates the number of different paths in a region. A group is then a 1-group.

Determining if a p-group is dependent upon the selection should be faster as there are no subregions to test, but it may require testing up to p paths. The user can control a maximum value m_p for p so the gains in culling of p-groups are not outweighed by maintaining and testing the list of ray paths.

a) Original scene b) Selection-dependent regions c) Image groups



d) Geometrically-dependent regions e) Scaling down the selection f) Using up to two tunnels per region

Figure 2: A simple scene

3.3 Color-tree Evaluation

For a color update, the user typically adjusts the same parameter frequently. In order to speed up the color update, color expression trees can be compressed according to the currently-selected parameter by maintaining at each node the evaluation of its subtree. In particular, the entire color expression of a pixel independent of the current parameter is temporarily replaced by a constant color value. This occurs when the pixel depends upon the selection but not on the selected shading or texture parameter.

This optimization is present in Séquin and Smyrl [15] in which a subtree is simply replaced by its value. The color trees could be also compressed physically, at the cost of rebuilding them each time another parameter is selected. This is also similar to compressing the shaders in Guenter *et al.* [8].

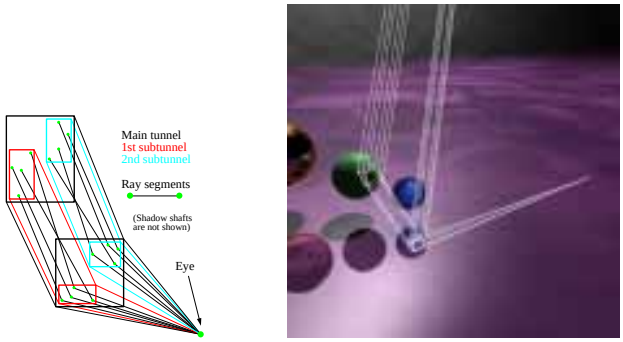


Figure 3: 2D and 3D representations of a tunnel

3.4 Bounding Ray Trees

Testing each ray tree for intersection with the selection becomes prohibitive if our goal is interactive manipulation. In order to greatly speed up ray tests, we use a hierarchy of bounding volumes for the ray trees. If the selection does not intersect such a bounding volume, then no ray test has to be performed for the enclosed ray trees.

3.4.1 Tunnels

For each region of the image, a volume that encloses every point of all its ray trees is kept. This region's main volume, called a *tunnel*, is built as a union of convex volumes, each called a *section*. Figure 3 shows a 2D representation of a tunnel and its 3D counterpart for a particular region (a group) and viewpoint from the scene depicted in figure 2. The tunnel encloses the eye (right apex), points on the floor, reflection points on the sphere, and two branching sections going towards the light above.

The first section of a tunnel encloses the eye point and all points directly visible from the eye through this region (first-generation ray segments). For each point light source, there is another section enclosing the same first-generation intersection points and the point light source. All the reflected ray segments of the second generation are enclosed by another section, and similarly for the refracted ray segments of this second generation. The two sets of endpoints of these secondary segments each generate another section respectively with each point light source. This process continues recursively for the next-generation segments.

A special situation occurs when the region is a group. In this case, all n ray trees have the same path and each section always encloses exactly n ray segments. These tunnels are usually thinner as the rays tend to remain closer to each other. However, if the region is not a group, some sections enclose ray segments whose endpoints do not belong to the same objects. Such a section may be much larger than necessary as some ray segments are not correlated. However,

if a region forming a p-group consists of p tunnels associated with each p different ray paths, then many of those undesirable sections are eliminated. If the region has more ray paths than the maximum number m_p, the ${m_p}^{th}$ tunnel contains all ray trees belonging to the $p - m_p + 1$ remaining ray paths.

3.4.2 Sections

A section bounds a set of ray segments. We adopted the *shafts* introduced by Haines and Wallace [9]. A shaft is built using two aligned boxes with each bounding plane passing through a face of a box or through a pair of relevant edges, each of them belonging to a different box. This construction has the property that if a shaft is built with two given boxes, then it encloses any subshaft built using two subboxes. A pyramid shaft is a particular shaft where all points at one end are the same. It is used for sections whose apex is the eye or a point light (figure 3).

Rather than testing for an intersection between the shaft and the selection itself, we use a bounding box around the selection. The test is based only on trivial rejections between the vertices of these two bounding volumes and their supporting planes. This is simpler and faster to compute, although if no trivial rejection has occurred, the test proceeds as if there were an intersection. For an exact test between a box and an arbitrary convex polyhedron, see Greene [6].

Figure 2 d) shows with white contours the regions updated when a geometrical change is applied to the selection (the central sphere). The updated portion of the image is larger than the minimal one in figure 2 b) because the bounding volume of the selection is used for the intersection test. This appears first in the projection of the selection in the image, and also in larger shadows and reflections of the selection. Moreover, this test is limited to trivial rejections only. The selection is scaled by $\frac{1}{4}$ in figure 2 e). The corresponding smaller updated regions are displayed with white contours. One can see that some irrelevant regions are located at the silhouette of the reflective sphere because of some incoherently large sections. Allowing up to two tunnels per region eliminates some of these sections (figure 2 f).

3.4.3 Updating the Tunnels

Adding an object to the ray trees may change some tunnels. This is also the case after an optical change. The tunnel of a region with at least one modified ray tree must be recalculated. However, the modification is propagated from bottom to top and no tunnel is rebuilt needlessly.

A region is formed by four subregions (quadtree structure), so a tunnel of this region is formed by its four subtunnels. At the lowest level, the ray segments form the shaft which is built using the two bounding boxes of their endpoints. At a higher level, the bounding boxes used for the shaft bound the respective subboxes of the four subshafts. Each level is thus updated in constant time.

4 Spacetime Considerations

In previous sections, we described the structures and optimizations allowing us to manipulate a scene interactively. However, we did not discuss memory requirements. Memory usage is the major drawback of our scheme as each pixel contains a lot of information.

In order to manipulate a scene at various image resolutions given limited memory, we present various trade-offs between the space required by the full structures and the computing time necessary to rebuild them.

In this section, we discuss strategies to lower the memory requirements, based upon controlling the number of different information pieces. This leads to a more general approach based upon the relative importance of an information. Note that none of these memory strategies affect the resulting images; they only influence what information will need to be recomputed. This makes the system flexible according to the space and time constraints.

4.1 Eliminating Information

All color trees, ray trees, and tunnels represent the largest portion of the memory usage in our system (table 2). Fortunately, we can exploit the locality involved with most changes.

By limiting to m_c the number of color trees, and if a color change does not update more than m_c pixels, then the refresh rate is not affected. However, if the number of updated pixels is larger than m_c, the excess pixels can be updated using the (slower) ray tree evaluation.

Similar to the color and ray tree reduction, a maximum number of tunnels can be specified by the user in order to control their memory usage. The time for reconstructing a tunnel becomes a function of the number of its immediate subtunnels already constructed, and the number of subtunnels to reconstruct. So the worst case of rebuilding a tunnel depends upon its level in the hierarchy.

4.1.1 Color Trees

Another solution specific to color trees prunes color subtrees by replacing them by their corresponding functions rather than their expansion in a tree. A procedural texture is an example of such a function. However when changing one parameter of this texture, the smaller trees thus obtained may be slower to evaluate as the function calculation is usually slower than its subtree evaluation. The user could also select a subset of parameters which are subject to change, and thus replace any subtrees in which no such parameter appears by a constant. Changing another parameter will involve however a ray tree evaluation.

4.1.2 Ray Trees

The ray trees represent another important portion of the storage used. All of this information can be removed in order to reduce the memory usage, requiring any intersection point, normal vector, and surface parametrization to be recalculated on demand. However, these quantities are not usually expensive to recompute, because the object to intersect with is already known. So the visibility calculation for the intersection point is done with respect to that object only. Although it is faster to preserve all this information, another space-time trade-off is possible by keeping only a certain number of ray trees. A deleted ray tree may be recalculated from a pointer to its ray path, which is stored in a global list.

4.1.3 Tunnels

Because of their hierarchical nature, many lower level subtunnels are rarely accessed. Also the lower we get in the structure, the more such subtunnels there are. By simply eliminating the tunnels at the two lowest levels, one obtains a reduction of a factor of about 16 in the number of tunnels. Moreover, this contributes to improving the global performance, as testing a lowest-level tunnel is more expensive than testing the few enclosed ray trees themselves.

4.2 Shadow Counters

For shadows cast by opaque objects, it is not necessary to know in which order objects are blocking the light, but only if the light is visible or not. Therefore, instead of constructing the list of ray segments starting from a given intersection point up to a point light, we use only a counter to indicate the number of blocking objects.

When an object is removed from the ray trees, the counter for a light at a ray node is decremented if the object was a blocker. If the

Scene	Color trees	Ray paths	1-groups	Ray trees	Tunnels
Figure 2 (simple)	66,000 32.5 MB	125 —	4,500 —	66,000 5.3 MB	10,900 5.4 MB
Figure 4 (complex)	66,000 20.5 MB	5,300 —	12,100 —	66,000 3.7 MB	10,900 4.3 MB

Table 2: Statistics on memory requirements

counter reaches zero, the illumination must be recomputed; otherwise, nothing has to be recomputed. When an object is added to the ray trees, some counters may be incremented. If it is the case for a counter previously at zero, the illumination from that light (now in shadow) must be recalculated.

Still, l such counters must be associated with each intersection point for a scene with l lights. To control the space used by the counters, we set a maximum of $b+1$ bits for each counter, hence handling up to 2^b-1 blockers. The *extra* bit represents overflow and it is set only when more than 2^b-1 blockers lie between the light and the intersection point. When removing a blocker, if any shadow counter is decremented to zero and its extra bit is set, we know that the point is still in shadow, but not the number of blockers. It is therefore necessary to re-shoot a ray towards the light with respect to the entire scene to update the counter value. If the value of b is zero, the extra bit becomes a simple flag indicating if the light is visible or not. Because the shadow counters are stored in the ray paths, a small value of b reduces the number of ray paths, but the increased cost of having to re-evaluate more often shadow rays.

If there are semi-transparent blockers among opaque blockers, a counter between two consecutive transparent blockers (along a shadow segment) is used.

4.3 Information Weight

A color subtree, a ray tree and a tunnel do not have the same space and time requirements. The specification of a weight with each piece of information, which indicates its relative importance among the others, is a more general approach.

The weight can be a function taking into account (1) the memory size needed by the information, (2) the time needed to recalculate it, and (3) its latest access time. The system will thus give preference to remove information that is more space intensive, that is faster to rebuild, and that has been inactive for a long time. This weight function provides a way to remove the less important information and to preserve the rest.

The memory size of each type of information depends upon the implementation but is simple to estimate. The time to recalculate a piece of information can be measured empirically, or estimated by various means. For instance, due to its hierarchical nature, the time needed to compute a tunnel corresponds to the computing time of its subtunnels plus its own computing time. For tunnels, we can also consider its surface area as a weight factor since the probability of intersecting an object is proportional to this area.

An information's inactive time is not measured in absolute time. Indeed, the wait between two successive manipulations should not influence the inactivity time. We suggest considering instead the number of changes since the latest access to the information.

5 Results

This section provides some statistics on our current implementation. The scene on which the manipulations were applied appears in figure 4, with the original image (a) before any manipulation and the resulting image (b) after all modifications. All times are in seconds and were gathered on a Silicon Graphics Indigo2 Extreme R4400, running at 150 MHz, with 128 MB of RAM.

The original scene consists of about 16,500 objects (13,500 polygons). It takes 1,100 seconds to preprocess the original image at a 256×256 resolution, Simply ray tracing the same scene requires 640 seconds. Statistics about the number of different pieces of information as well as their respective memory requirements are provided in table 2. To compare, we also give the same statistics for the simple scene of figure 2. One can notice that although simpler in terms of geometry, the visibility complexity of figure 2 is higher than the one of figure 4, which is illustrated by larger memory needs for its color trees, ray trees and tunnels.

The top section of table 3 shows statistics on color changes. The modifiable parameters that are integrated in the color tree are all texture parameters defining a constant color, except for the color of the light source. Traversing the image quadtree to identify the pixels dependent upon the selection takes from 1 to 2 seconds. Using groups for this selection-dependent preprocessing (SDP) leads to 20-40% savings of the first color updates. Computing all colors from the ray paths takes less than 10 seconds. If all ray trees are used instead of recomputed from the ray paths, this time goes down by 40-70%. If the color trees themselves are used, this time is reduced by 90-95%. The four color changes in table 3 are all updated under half a second. The savings due to compression of the color trees are less significant (1-20%).

The optical change in the middle section of table 3 displays a similar behavior than first color changes with respect to the use of groups for the selection preprocessing (25%).

Some statistics on geometrical changes are given in the bottom section of table 3. Changes are dependent upon the number of rays that must be shot and how expensive they are to intersect with respect to the entire scene. The use of tunnels greatly reduces all geometrical changes, especially when only a fraction of all ray trees are allowed to reside in memory. This is due to the fact that many ray trees do not need to be recomputed from ray paths because simply culled by the tunnels. Indeed, tunnels culled between 65-90% of the ray tests. The hierarchical nature of the image quadtree and tunnels shows that almost no performance is lost, even when using 10% of all ray trees.

6 Summary and Conclusion

In this paper, we presented two tree structures allowing an incremental recomputation of the image after any modification of a scene viewed from a fixed camera.

The *color tree* preserves the entire expression leading to the final color of a pixel. Any changes affecting the value of parameters in these trees, such as shading and texture parameters, are quickly displayed by re-evaluating only the subtrees dependent upon the modified parameters.

By storing the image in a quadtree of regions, a preprocessing step identifies by a flag each region within which at least one color tree is affected by a given color change. This preprocessing is, in addition, sped up by the notion of groups, where regions are formed by pixels with identical ray paths. So, an entire group can be eliminated by testing a single ray path.

The *ray tree* preserves only the visibility specific information of the rays generated from a pixel. Any changes affecting a ray tree are handled by re-shooting rays from the previous valid intersection point. Any modification in a ray tree is directly updated in its corresponding color tree. All optical and geometrical changes can be handled with this structure.

After a geometrical change, it is possible to avoid testing each individual ray segment with the current selection. To do so, the ray trees are combined into tunnels formed by a union of shafts. The ray segments modified by a geometrical change are therefore quickly

a) Before changes b) After changes c) Higher resolution

Figure 4: A more complex scene

Color Change	SDP without groups	SDP with groups	Only ray paths	1% ray trees	10% ray trees	All ray trees	Color tree	Compression
leaves color left tree	1.5	1.1	4.65	4.45	1.26	1.26	0.28	0.23
highlight left sphere	1.5	1.1	0.94	0.66	0.52	0.52	0.12	0.09
marble color on Beethoven	1.91	1.43	8.36	7.78	5.09	4.91	0.35	0.32
ground color	2.17	1.59	7.60	7.28	4.50	2.49	0.47	0.44
Optical Change								
index of refraction	1.53	1.12	8.55	8.51	8.47	8.47	—	—

Geometrical Change	removing	rendering		
		all ray trees (0 / 1 / 2) tunnels	10% ray trees (0 / 1 / 2) tunnels	1% ray trees (0 / 1 / 2) tunnels
transform central sphere	51.0	37.8/ 8.8/ 7.8	82.9/ 11.4/ 7.8	88.8/ 16.2/ 12.4
move left colored cow	111.0	38.4/ 13.3/ 12.2	79.2/ 13.3/ 12.3	85.0/ 17.0/ 14.5
move refractive sphere	12.6	39.4/ 6.0/ 5.7	85.6/ 6.2/ 5.7	90.5/ 6.9/ 6.4
flying furthest tree	10.0	38.5/ 16.0/ 13.8	76.0/ 16.0/ 13.8	80.5/ 22.0/ 19.0

Table 3: Statistics on changes (times in seconds)

detected by intersecting the selection with the hierarchical structure. Any change in the ray trees is updated in a bottom up fashion, from the modified ray trees up only to its bounding tunnels.

As a result, the images are usually updated in less than a second for most color changes. Optical and geometrical update times depend upon the number of rays shot, and upon the complexity of the object (selection or scene) to intersect with these rays. However the number of new rays is usually a small fraction of all rays necessary to render the image.

The question of high memory requirements is addressed by pruning color subtrees and eliminating tunnels and ray trees. This information can be efficiently recomputed on demand by keeping a pointer to its corresponding ray path. Also, an adjustable weight function based on memory size, recomputation time, and age of information, helps to determine the best information to keep within the available memory space.

The main conclusion we can draw from our scheme and its current results is that the entire visibility could be handled efficiently using only ray paths. This information does not require so much memory for the important gains it provides. All remaining memory can be used to speed up specific changes, by building various structures such as color trees, ray trees, and tunnels. Their respective memory spaces can be managed adaptively according to local changes.

We expect the benefits of efficient incremental re-rendering to lead to more advanced interactive systems, since processing time can be concentrated on the phenomena the user is interested in, rather than on redundant rendering.

7 Future Work

The current system suggests some interesting avenues to investigate. It could be easily extended to render efficiently animated sequences from a fixed camera when a limited number of objects are moving. It should be possible to exploit time coherency from our knowledge of all motions.

The hierarchical structures, ray tracing rendering and weight functions provide essential information about what is changing with respect to all previously computed information. All this knowledge makes the system a potential candidate for well balanced workload distribution in parallel processing, and memory management. For instance, the use of a large storage device provides an alternate solution to the memory usage for higher image resolution. Such a virtual storage has typically slower access time that can however be factored in the information weight when it is removed from the prime memory. So faster access memory acts then as cache which we could manage accordingly.

The structures can also be used for other purposes. The similarity between image formation and light propagation [1] suggests to use the ray paths only for light preprocessing in order to handle the high memory requirements to reduce the aliasing effects. The incremental updating for changing scene geometry should avoid much unnecessary lighting recomputations. Furthermore, the beam-like shape of tunnels suggests a way to estimate the contribution of participating media, coherent ray tracing, and image filtering.

Acknowledgments

We would like to thank Chris Romanzin and Neil Stewart for their help. We acknowledge financial support from NSERC, FCAR, the Université de Montréal, and Taarna Studios.

References

[1] James Arvo. Backward ray tracing. SIGGRAPH 86 Tutorial notes on Developments in Ray Tracing, August 1986.

[2] Robert L. Cook. Shade trees. Proceedings of SIGGRAPH 84. In *Computer Graphics*, 18, 3 (July 1984), pp. 223–231.

[3] Franklin C. Crow. Shadow algorithms for computer graphics. Proceedings of SIGGRAPH 77. In *Computer Graphics*, 11, 2 (July 1977), pp. 242–248.

[4] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. Proceedings of SIGGRAPH 93. In *Computer Graphics* Proceedings, Annual Conference Series, August 1993, pp. 247–254.

[5] Ned Greene. Applications of world projections. Proceedings of Graphics Interface 86, (May 1986), pp. 108–114.

[6] Ned Greene. Detecting intersection of a rectangular solid and a convex polyhedron. In Paul Heckbert, editor, *Graphics Gems IV*, pages 74–82. Academic Press, Boston, 1994.

[7] Ned Greene and M. Kass. Hierarchical Z-buffer visibility. Proceedings of SIGGRAPH 93. In *Computer Graphics* Proceedings, Annual Conference Series, August 1993, pp. 231–240.

[8] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. Proceedings of SIGGRAPH 95. In *Computer Graphics* Proceedings, Annual Conference Series, August 1995, pp. 343–350.

[9] Eric Haines and John Wallace. Shaft culling for efficient ray-traced radiosity. In *Eurographics Workshop on Rendering*, 1991.

[10] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. Proceedings of SIGGRAPH 90. In *Computer Graphics*, 24, 4 (August 1990), pp. 215–223.

[11] David A. Jevans. Object space temporal coherence for ray tracing. Proceedings of Graphics Interface 92, (May 1992), pp. 176–183.

[12] Peter Litwinowicz and Gavin Miller. Efficient techniques for interactive texture placement. Proceedings of SIGGRAPH 94. In *Computer Graphics* Proceedings, Annual Conference Series, July 1994, pp. 119–122.

[13] K. Murakami and K. Hirota. Incremental ray tracing. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, June 1989, pp. 17–32.

[14] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. Proceedings of SIGGRAPH 92. In *Computer Graphics*, 26, 2 (July 1992), pp. 249–252.

[15] Carlo H. Séquin and Eliot K. Smyrl. Parameterized ray tracing. Proceedings of SIGGRAPH 89. In *Computer Graphics*, 23, 3 (July 1989), pp. 307–314.

[16] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley Publishing Company, 1992.

[17] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[18] Lance Williams. Casting curved shadows on curved surfaces. Proceedings of SIGGRAPH 78. In *Computer Graphics*, 12, (August 1978), pp. 270–274.