# Raytraced rendering using t-buffer and shadow-buffer

Nikunj Raghuvanshi and Sanjay G. Dhande

Department of Computer Science, IIT Kanpur,
Kanpur - 208016, India

{nikunj, sgd}@iitk.ac.in

## Abstract

Ray-tracing is one of the most researched areas in 3D photo-realistic image synthesis. We propose the T-BUFFER algorithm, which is a modification to the conventional ray-tracing algorithm and compares very favorably with it in terms of performance, which is achieved primarily through reduction in the number of unsuccessful intersection tests. Our major contribution is the integration of a shadow-buffering technique which, in conjunction with the t-buffer, can create ray-traced renderings with reasonably accurate shadows much more efficiently than the common approach, while giving the user control over the quality of shadows and the related computational efficiency. We support the above claims with empirical evidence which suggest an approximately linear gain over the original ray-tracing algorithm.

We also describe REALITY - a complete rendering engine which has been developed for the purpose of empirical comparison between the T-BUFFER algorithm and the classical ray-tracing algorithm. Both algorithms were implemented in a common code base to facilitate direct comparison. The engine also implements the above mentioned shadow-buffering technique to accelerate shadow calculations. We present a thorough comparative analysis of the original algorithm and the T-BUFFER algorithm, based on the rendering times of REALITY and show that the results strongly corroborate with the linear gain expected theoretically.

**Keywords**: rendering, photo-realism, ray-tracing, t-buffer, shadow-buffer

## I. Introduction

In recent years, ray-tracing has emerged as one of the most preferred techniques for generating photo-realistic images of 3D scenes. The original algorithm as proposed by Appel[1], can generate renderings of reasonably complex scenes, with point light sources and the shadows cast by them. The basic idea behind APPEL's algorithm is to back-trace light rays from the scene that fall on the observer's eye, which is similar to reversing the process as it actually occurs in nature. Also, “shadow rays” are shot from points on the objects towards the light source to find if they are occluded from the light source by an intervening object, or the object itself (*self-occlusion*). This algorithm generates quite convincing shadows but the results are not photo-realistic, mainly because the algorithm is not capable of implementing a more accurate global illumination model. See [9] for

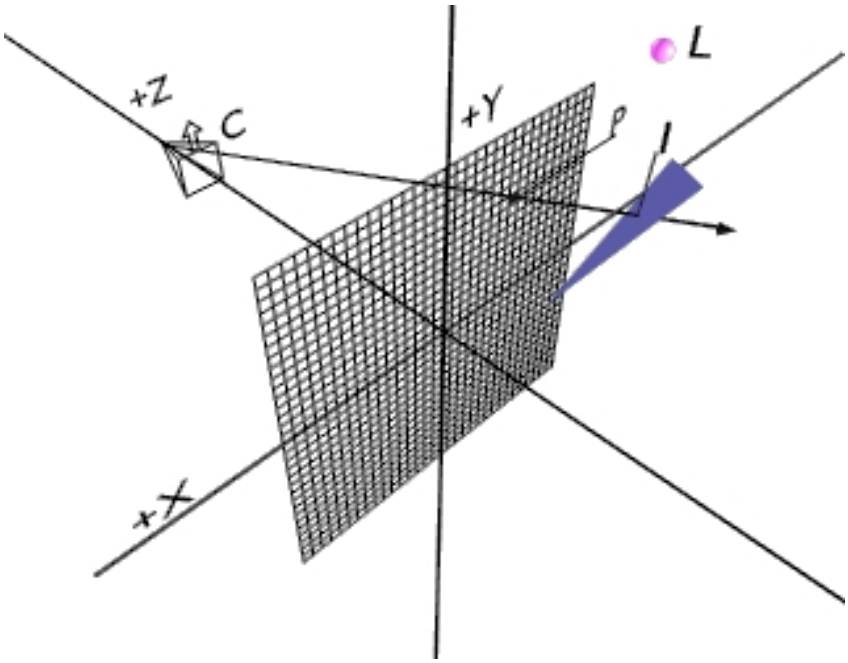


**Figure 1**: The basic principle of ray-tracing. Rays are shot through the viewport and intersections found with the objects in the scene. The corresponding pixel is painted with the object's color, subject to the illumination due to the light source L.

a more accurate account of the problem of global illumination and KAJIYA's mathematical abstraction of the problem.

This algorithm was extended by Whitted[2] and Kay[3] to include specula reflections and the complex optical phenomenon of refraction. WHITTED's algorithm shoots secondary rays from the object's surface, corresponding to specula reflections and refractions. This amounts to calling the algorithm recursively for these secondary rays. This algorithm is referred to as *recursive ray-tracing*.[1] The results of this algorithm are quite convincing since the illumination model is capable of modeling most of the optical phenomena quite accurately. However, the image quality comes at the cost of a large computational complexity. The recursive ray-tracing algorithm is very slow, the main bottleneck being the large number of intersection calculations performed for the rays, those shot from the observer (primary rays) and those shot from the objects (shadow rays and secondary rays). The T-BUFFER algorithm that we propose here, tries to reduce the number of primary rays and shadows rays by avoiding futile intersection tests altogether. That is, while the traditional ray-tracing algorithm shoots a ray and tests it for intersection with all the objects, the T-BUFFER algorithm shoots only those primary rays which are bound to intersect an object (which may/may not be the closest object intersected by that ray). This is accomplished by considering an object at a time and ray-tracing all the primary rays that may intersect it, and noting the results in a t-buffer. As will be elaborated later, the t-buffer is managed in such a manner that once all the objects have been considered, it holds all the visibility-related information that is needed by the succeeding steps of the algorithm. A similar idea is used in the "ZF-buffer" algorithm[6] to reduce the number of intersection tests. Our major contribution is the extension of the same idea as the t-buffer to the calculation of shadows for a scene. The key point to observe is that if a light source is replaced by a viewer, then non-visibility from the light source's viewpoint and shadowing are equivalent problems. Therefore, the algorithm uses a "shadow-buffer" to find shadowed areas in the scene. This way, there is a lot of gain in terms of performance for ray-traced rendering with shadows. We discuss the

[1] Throughout this paper, references to "the traditional ray-tracing algorithm" or "the classical ray-tracing algorithm" are to be interpreted as the recursive ray-tracing algorithm as referred to here.

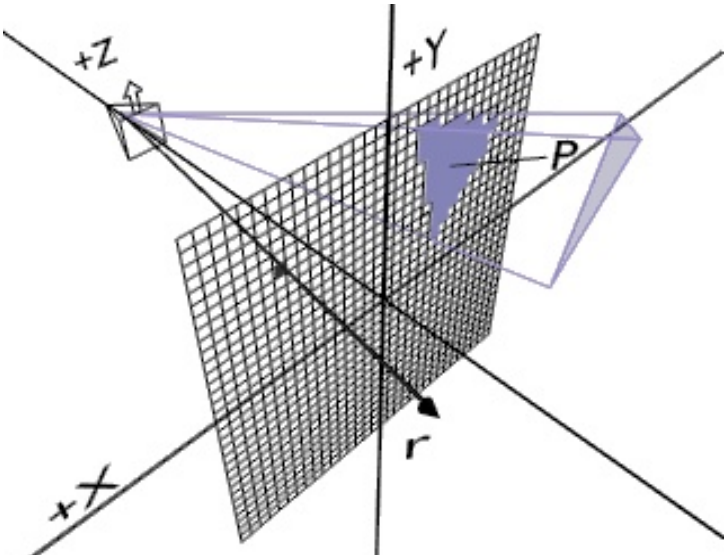


**Figure 2:** The ray marked **r** does not intersect any object in the scene. The T-BUFFER algorithm avoids shooting such rays. Only rays passing through the projection of the object on the viewport, **P** are tested for intersections.

working and implementation aspects of the t-buffer and the shadow-buffer in detail in sections 2 and 3 respectively. In section 4, we illustrate the working of the T-BUFFER algorithm through a set of simple examples. We discuss the empirical results obtained and their analysis for various scenes in section 5. We conclude with a brief statement of the main results and possible further extensions to this work in section 6.

# II. The t-buffer

The traditional ray-tracer works by shooting rays from the observer's viewpoint into the scene, in order to back-trace the light rays from the scene that reach the observer. This idea is illustrated in Figure 1. The observer/camera is located at point $\vec{C}$. The ray intersects the viewport (shown as a grid of pixels) at pixel *P* and the object *O* at point $\vec{I}$ . There is a point light source at $\vec{L}$. The color of pixel *P* is partially determined by factors like the material of *O* and the position vectors $\vec{L}$ and $\vec{C}$. A major part of the computational effort is involved in finding the closest object hit by the ray and the corresponding point of intersection $\vec{I}$ .

The traditional ray-tracer does a large number of futile intersection tests while rendering a scene. To see why this is so, consider a simple scene with a single triangle as shown in Figure 2. A traditional ray-tracer would shoot all the rays which pass through some pixel on the viewport and test them for intersection with the triangle. Its easy to see that a majority of these intersection tests will be futile (an instance is the ray marked 'r' in the figure). The T-BUFFER algorithm, as will be demonstrated, ensures that these futile tests are completely avoided for primary rays by shooting only those rays that lie inside the projection of the triangle on the viewport (the region marked 'P' in Figure 2).

## A. *Structure of the t-buffer*

The T-BUFFER algorithm uses a data-structure called the *t-buffer*, which is a two-dimensional array with the same dimensions as the viewport such that for every pixel on the viewport, there

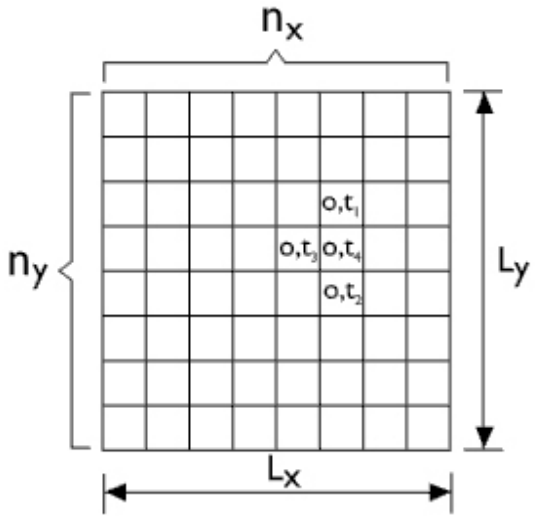


**Figure 3:** Organization of the t-buffer. The length and breadth, $L_x$ and $L_y$ respectively are identical to that of the viewport and so are the resolutions, $n_x$ and $n_y$. Each cell stores information for the corresponding pixel on the viewport. Some of the entries are shown.

is a corresponding location in the t-buffer. At each location in the t-buffer, a buffer item is stored which is a tuple consisting of a distance value, called the *t-value (t)* and an object pointer *(O)*. The organization of the t-buffer is illustrated in Figure 3. A tuple (*O*, *t*) stored in the t-buffer is interpreted as follows - if a ray is shot through the viewport, intersecting it at a pixel which corresponds to the given buffer-item, then the nearest object hit is *O* and the intersection point is at a distance *t* from the observer along the direction of the ray.

## B. *Filling the t-buffer*

The T-BUFFER algorithm attempts to collect all the information required by the t-buffer in an optimal manner. It is clear that once the t-buffer is correctly filled, further intersection calculations are not required for primary rays. The naive approach would be to shoot a ray and perform intersection calculations with all the objects in the scene. Out of all the intersection distances, take the minimum, identify the object hit, and fill the corresponding entry of the t-buffer with the collected data for the particular pixel. As discussed in section 2A, this approach is identical to the traditional ray-tracing algorithm and amounts to a lot of wasted intersection tests.

We now describe the functioning of the T-BUFFER algorithm. To optimize the number of intersection calculations, we shoot only those rays that are *bound to hit an object*. Figure 4 illustrates the idea. To fill the t-buffer, we consider each object in turn (suppose *O* is the object under consideration) and shoot *all* the rays which intersect it. This amounts to shooting all the rays that pass through pixels belonging to the region R in the figure. Now we consider all these rays in turn. Suppose $\vec{r}$ is the ray under consideration which passes through the viewport at pixel P and goes on to intersect object *O* at a distance *t* along the direction of the ray. The t-buffer location corresponding to pixel *P* is looked up. Suppose the t-value stored there is *t'*, the buffer-item being *(O', t')*.

If *t'<t*, there must be some other object *O'* which is hit at a smaller distance than *O*, when a ray is shot from the observer's location passing through *P*. This implies that *O'* occludes *O* from the viewer for this pixel. Therefore, the buffer-item is left unmodified. If, on the other hand, *t'>=t*, it implies that *O* occludes *O'* and hence, the current buffer item is over-written with *(O, t)*. This particular case is illustrated in Figure 4, since the object *O* obscures *O'* for pixel *P*, the present contents of the t-buffer are overwritten with a tuple corresponding to *O*.

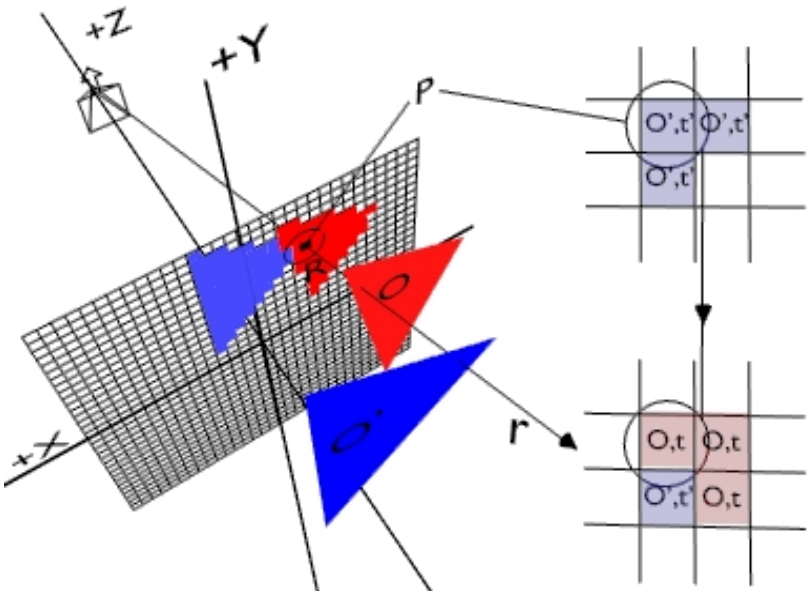


**Figure 4:** Working of the t-buffer. Object O' is considered first and the t-buffer filled likewise. When O is considered, some of the entries of O' are over-written since O' is farther from the viewport than O for the corresponding pixels. One such pixel (P) is illustrated.

It is clear from the above argument that once all the objects have been considered, the t-buffer contains only the minimum distance of intersection for each pixel on the viewport and hence the correct object visibility information for the scene under consideration.

### C. *Utilizing the t-buffer*

The T-BUFFER algorithm first updates the t-buffer according to the scene, as described in the previous section. Once all the values have been updated, it uses the information gathered to accelerate further processing for primary rays. The algorithm considers each pixel on the viewport in turn and shoots a ray through it. For that pixel, it reads the contents of the corresponding location in the t-buffer. This directly gives the closest object that is intersected by the ray and the intersection distance. From this information, and the direction of the ray being shot, the point of intersection can be found through a trivial vector calculation. Using the point of intersection and the actual object hit, the surface normal of the object at that point is found. This information is combined with further processing of the algorithm to yield the color that the particular pixel should be painted in. A more elaborate description of this process is given in section 4, through illustrations for a simple scene.

## III. The shadow-buffer

In this section, we present our major contribution - the incorporation of a shadow-buffer developed on the lines of the t-buffer, to accelerate the shadow calculations for ray-traced rendering. The problem of shadow calculation in ray-tracing can be stated as - Given a light source at $\vec{L}$ and a point $\vec{P}$ on an object O, find if the straight line from $\vec{L}$ to $\vec{P}$ intersects an object O' at a point, say $\vec{I}$, such that,

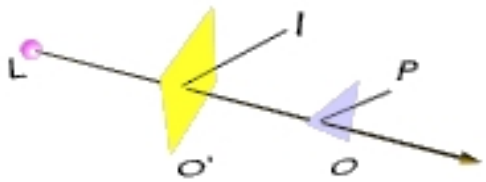


**Figure 5:** Object O' occludes O from the light source L for the ray considered.

$$\left|\vec{I}-\vec{L}\right| < \left|\vec{P}-\vec{L}\right| \;\; and \;\; \left(\vec{I}-\vec{L}\right)\bullet\left(\vec{P}-\vec{L}\right) > 0. \tag{1}$$

Figure 5 illustrates the situation that is described above. This condition tests if a ray of light starting from the light source at $\vec{L}$ is able to reach point $\vec{P}$ without obstruction. If the above condition holds, light cannot reach from the light source to $\vec{P}$ and hence, the point is occluded from the light source and receives no light directly from the light source. Otherwise, the point is directly illuminated by the light source.

## A. *The traditional approach*

The straight-forward approach for finding if condition (1) holds would be to shoot a ray from point $\vec{P}$ towards point $\vec{L}$ (or vice-versa - which is equivalent, as can be easily verified). Stating mathematically, the ray would be,

$$\overrightarrow{v(t)} = \vec{P} + t \cdot \frac{\vec{L}-\vec{P}}{\left|\vec{L}-\vec{P}\right|}, \;\; t > 0. \tag{2}$$

Now, the ray $\overrightarrow{v(t)}$ is tested with **each** object in the scene to find if it intersects the object. Suppose the ray does intersect an object with the intersection distance being $t'$ (say). Then we test if,

$$0 < t' < \left|\vec{L}-\vec{P}\right|. \tag{3}$$

As discussed previously, the truth of this condition is equivalent to $\vec{P}$ being in shadow with respect to the light source at $\vec{L}$. Thus, if this condition fails for all the objects in the scene, $\vec{P}$ is not in the shadow of the light source at $\vec{L}$.

## B. *Shadow-buffer – the motivation*

The calculation of shadows as discussed above involves a lot of intersection tests because for a single point on an object, it requires intersection tests with all the objects in the scene. The principal aim of the shadow-buffer is to reduce this computational overhead to accelerate shadow calculations. The basic idea behind the shadow-buffer is derived from the fact that if the light source is treated as a viewer, then the part of the scene that is visible to it is outside the shadow cast by it, the rest being in the shadow. The idea of using the light source as the center of projection for finding shadows is in fact quite old, and is discussed in [4], although not in

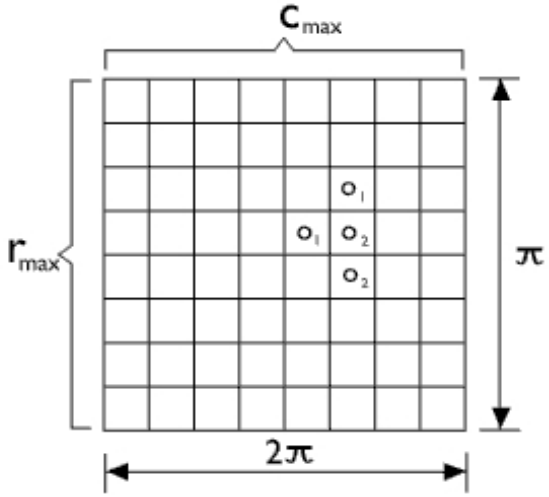


**Figure 6:** Structure of the shadow-buffer. Note that it is almost identical to the t-buffer accept that the dimensions and resolution are different from the t-buffer. Also, the resolution of the shadow-buffer is a variable, as opposed to the t-buffer, for which the resolution has to match that of the viewport. Higher values of resolution tend to give more accurate shadows but at a higher computational cost.

context of ray-tracing. Stretching the above analogy a bit further, while a viewer has a limited field of view, a point light source "sees" in all directions. The shadow-buffer uses these facts to reduce the complexity of shadow calculation, utilizing the same basic principle as the t-buffer, thus avoiding a lot of intersection tests while testing if a point is in the shadow of a given light source.

## C. *Structure of the shadow-buffer*

The shadow-buffer, as illustrated in Figure 6, is a two-dimensional array with dimensions $(r_{\max}, c_{\max})$ where an element with the index (r,c) in the array corresponds to a point on a unit sphere centered at the origin. The polar angles of the point are given by,

$$\theta = \frac{r}{r_{\max}} \times \pi,\ \phi = \frac{c}{c_{\max}} \times 2\pi. \tag{4}$$

Each element of the shadow-buffer is a tuple (*o*,t) consisting of an object pointer *o* and a distance value t, analogous to the t-buffer. Supposing the shadow-buffer is B, the set of objects is O, the light source is at $\vec{L}$ and the direction unit vector is $\hat{d} = (\cos\theta\cos\phi)\hat{x} + (\cos\theta\sin\phi)\hat{y} + (\sin\theta)\hat{z}$ ($\theta$ and $\phi$ as calculated above), the contents of the array are interpreted as,

$$\begin{gathered} (B[r,c] = (o,t)) \Rightarrow \vec{L} + t \cdot \hat{d} \text{ lies on } o \\ \text{and} \\ \forall o' \in O, \left(\exists t', \vec{L} + t' \cdot \hat{d} \text{ lies on } o'\right) \Rightarrow t' \geq t. \end{gathered} \tag{5}$$

That is, if a ray of light leaves the light source with the direction vector being $\hat{d}$, the nearest object it hits is *o*.

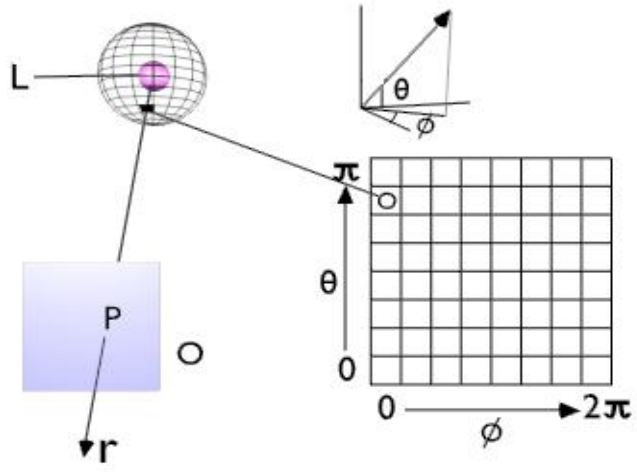


**Figure 7:** Interpretation of the contents of the shadow-buffer. Any ray **r**, from the light source at **L** is mapped onto a location **l** of the shadow-buffer which contains a pointer to the object **O** that is the closest hit by the light ray. To check if a point **P** on an object **O** is not in the shadow, we just need to check if the location **l** has the object pointer stored as **O**.

## *D. Utilizing the shadow-buffer*

The function of the shadow-buffer is to provide all the shadow information there is in the scene, with respect to the light source it is associated with. Note that the shadow-buffer stores information only for one light source, therefore we require as many shadow-buffers as there are light sources in the scene. As will be shown shortly, once the contents of the shadow-buffer are correctly computed, consistent with (5), only shadow-buffer lookups need to be performed while testing for shadows. Moreover, as one may notice, the contents of the shadow-buffer are *independent of the location of the viewer*. This has an important consequence - for renderings of the same scene from different viewer positions/orientations the contents of the shadow-buffer need not be recomputed as the locations of the shadows do not change until there is a change in the relative position/orientation of the light source and the objects. This reduces the computational complexity of multiple renderings with fixed light sources by a large factor.

Now we describe how shadow-buffer lookups may be utilized to extract the shadow information for a scene. The process is illustrated in Figure 7. Suppose we are interested in finding if the point $P_o$ is shadowed with respect to the light source at $\vec{L}$.[2] Suppose, we represent this by the predicate $inshadow\left(P_0, \vec{L}\right)$. $B_L$ is the shadow-buffer associated with the light source at $\vec{L}$ and $\hat{x}, \hat{y}, \hat{z}$ represent the standard Cartesian unit vectors. First, we compute the direction vector from the light source to the given point (shown as **r** in the figure),

$$\hat{d} = \frac{\vec{P_o} - \vec{L}}{\left|\vec{P_o} - \vec{L}\right|}. \tag{6}$$

Next, we find the cell in the shadow-buffer corresponding to this direction vector. The row and column of the cell (r and c respectively) are given by,

[2] The subscript 'o' in $P_o$ denotes that the point P lies on the object named o.

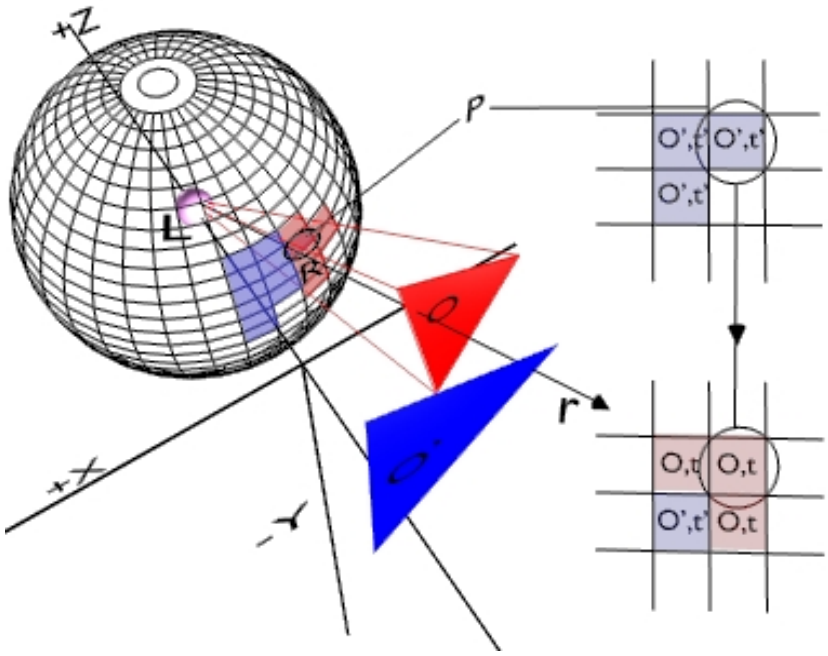


**Figure 8:** Filling the shadow-buffer. Object O' is considered first and the shadow-buffer filled likewise. When O is considered, some of the entries of O' are over-written since O' is farther from the light source L than O for the corresponding pixels. One such pixel, P, is illustrated.

$$\begin{aligned}
\theta &= \arcsin\left(\hat{d}\square\hat{z}\right) \\
\phi &= \arccos\left(\frac{\left(\hat{d}-\left(\hat{d}\square\hat{z}\right)\hat{z}\right)}{\left|\left(\hat{d}-\left(\hat{d}\square\hat{z}\right)\hat{z}\right)\right|}\cdot\hat{x}\right) \\
r &= \frac{\theta}{\pi}\times r_{\max} \\
c &= \frac{\phi}{2\pi}\times c_{\max}\ .
\end{aligned} \tag{7}$$

Finally, we access the object pointer stored at the location $B_L[r,c]$, r and c having been calculated as above - only if it is the same as *o*, is object *o* illuminated at the point $P_o$ by the light source at $\vec{L}$. Stating mathematically,

$$B_L[r,c]=(o',t'),\ o\neq o' \Leftrightarrow inShadow\left(\vec{P_o},\vec{L}\right). \tag{8}$$

## *E. Filling the shadow-buffer*

In this section, we describe the method used to fill the shadow-buffer so as to fulfill the requirement set in relation (5). Observing that the question of illumination of a certain point by a given light source can be reduced to the visibility of the point from the light source, as discussed in section 3B, it is clear that a technique similar to the t-buffer can be applied for the shadow-buffer as well. The algorithm is illustrated in Figure 8. The shadow-buffer is visualized as being wrapped around a unit sphere centered on the light source, L. The algorithm considers

all the objects in the scene one at a time. Suppose the currently chosen object is *o* (the red triangle in the figure). All the rays from the light source which can *possibly* intersect with *o* are shot and the tuple (*o*, t) is stored at B[r,c], where r and c are the corresponding row and column in the shadow-buffer, computed from the direction vector of the ray as shown in relation (7). Note that in the actual scene, some of these rays may be obstructed by other objects. Therefore, overwriting the present contents of the shadow-buffer in all the cases would lead to incorrect results. This can be accounted for in this way - while storing the tuple (*o*, *t*), we find the t component of B[r,c], the tuple already stored in the shadow-buffer. Suppose its value is *t'*. Now, two cases arise -

1. If $t' < t$, some other object which was considered in one of the preceding iterations is closer to the light source than *o*. Hence, (*o*, t) is ignored and not stored at all

2. If $t' \geq t$, overwrite the tuple stored at the shadow-buffer location B[r,c] since we have just found an object closer to the light source, for the current row and column under consideration. This is the case illustrated in Figure 8 - since the red triangle ($o$) is closer to the light source than the blue one ($o'$) for the ray **r**, the present contents of the shadow-buffer (shown top-right) are overwritten with the tuple for *o* (shown bottom-right)

The above method ensures that once all the objects have been considered, the tuples stored in the shadow-buffer correspond to the object nearest to the light source, in various directions in which light rays leave the light source. Thus, the above method ensures that once all the objects have been considered, all the tuples stored in the shadow-buffer satisfy the criterion set forth in (5), and hence contain all the shadowing information for the given light source.[3]

# IV. The algorithm through illustrations

In this section, we describe the complete T-BUFFER algorithm through a series of illustrations and show how the information present in the t-buffer and the shadow-buffer is utilized in an actual run. We also show the outputs given by REALITY, the ray-tracing engine we have developed for testing the algorithm and the effect of various parameters on the actual rendering. For simplicity of illustration, we assume only one point light source and a very simple scene, consisting of three stacked spheres placed on an infinite ground.[4]

## *A. Scene 1*

The configuration of REALITY for this run is as follows –

[3] At this point, one may note that the distance value - t, stored in the shadow-buffer is not required in any later steps of the algorithm. More specifically, none of the steps described in section 3D require the t-value. Therefore, once the shadow-buffer has been filled with the correct object pointers, all the t-values can be discarded, thereby reducing the memory requirements of the shadow-buffer by half.

[4] The case of multiple light sources is a straight-forward extension of the algorithm and is not discussed here for the sake of simplicity of illustration.

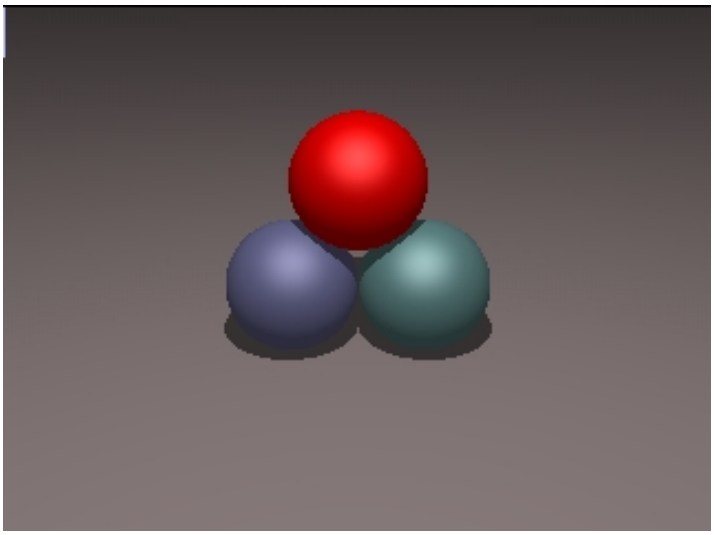

**Figure 9:** Output of REALITY for a simple scene with textures, reflections and shadow-buffer disabled. Note that the scene does not seem realistic, which illustrates the importance of reflections and textures for realistic image synthesis.

| Feature | Status |
|---|---|
| Shadows | ON |
| Inter-Object Reflections | OFF |
| Textures | OFF |
| t-buffer | ON |
| Shadow-buffer | OFF |

Figure 9 shows the output of REALITY for this configuration. The outputs of the traditional algorithm and the T-BUFFER algorithm are identical for this case, which is expected since the t-buffer doesn't introduce any aliasing other than that inherent in ray-tracing itself. Now we describe the steps the T-BUFFER algorithm takes to do the rendering of the scene shown in the figure.

1. **Filling the t-buffer**

   i. Initialize all the entries of the t-buffer to the tuple (*null*, *infinity*) so that the object pointer is set to null and the intersection distance is set to infinity. Consider all the objects at a time. In this particular scene, there are only four objects - the ground and three spheres.[5] Suppose the ground is considered first

   ii. Trace primary rays through all the pixels on the viewport which lead to an intersection with the ground and fill the corresponding t-buffer entries with the corresponding intersection information as discussed in section 2B

   iii. Process the spheres in a similar fashion. Since the intersection distance for any point on any of the spheres is smaller than the value already present in the t-buffer at the

[5] Note that the sky is not treated as an object and hence no explicit intersection tests are done for it. If a ray doesn't intersect any object, it is assumed to be going towards the sky and is treated accordingly.

corresponding location (if it corresponds to one of the intersections with the ground considered in step ii), all t-buffer entries which correspond to the spheres are filled with the corresponding information, overwriting the earlier ground entries present at those locations. Intuitively, this ensures that the spheres occlude the ground from the viewer, which one would naturally expect. Obviously, some entries for the spheres may also over-write each other depending on the viewer's position. Now the t-buffer contains complete visibility information for the scene under consideration

2. **Rendering**

i. Consider each pixel on the viewport at a time. Suppose P is the pixel under consideration
ii. Look at the t-buffer location corresponding to P and find the object pointer, *o* and the t-value, *t*
iii. Calculate the intersection point $\vec{I}$ from the above information[6]
iv. Calculate the surface normal $\vec{n}$ of *o* at the point $\vec{I}$
v. **Illumination and Shadow Computation:** Shoot a ray from $\vec{I}$ towards the light source, go through all the objects sequentially and find if it intersects any of the objects, before it reaches the light source
   a. If yes, $\vec{I}$ is in the shadow of the light source, and the light source's direct contribution to the color is 0. Therefore, the specular and diffuse contributions are 0. Denoting the specular and diffuse contributions by $C_s$ and $C_d$ respectively, $C_s = C_d = 0$
   b. Otherwise, knowing the position of the light source and $\vec{n}$, find the specular and diffuse components of the color contribution from the light source – $C_s$ and $C_d$ respectively.[7] The actual calculation of these is quite common in graphics and are calculated using the assumptions of a lambertian surface and will not be discussed in detail here. Briefly, Lambert's law is used while calculating $C_d$ and PHONG's illumination model[10] is used for calculating $C_s$
vi. Calculate the ambient contribution to the color $C_a$ (assuming some constant amount of ambient light)
vii. Return the color for pixel P as $C_{total} = C_a + C_d + C_s$

## *B. Scene 2*

The configuration of REALITY for this run is as follows –

[6] If the direction vector corresponding to P is $\hat{d}$ and the viewer is at $\vec{C}$, then the intersection point is given by $\vec{I} = \vec{C} + t \cdot \hat{d}$.

[7] All the color contributions discussed here eg. $C_s$, $C_d$ etc., are treated as vectors in 3D RGB space and added accordingly.

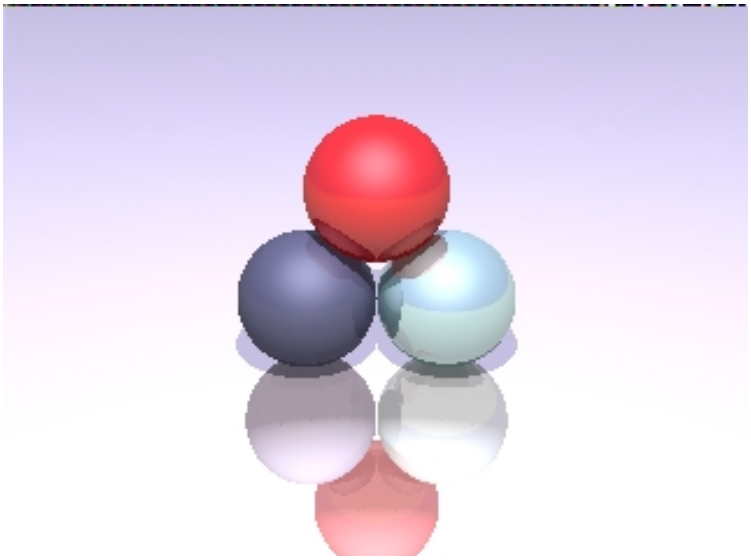

**Figure 10:** Output of REALITY for a simple scene with textures and shadow-buffer disabled. Note the enhancement in rendering-quality due to the addition of inter-object reflections. Specifically, note the inter-reflections among the spheres.

| Feature | Status |
|---|---|
| Shadows | ON |
| Inter-Object Reflections | ON |
| Textures | OFF |
| t-buffer | ON |
| Shadow-buffer | OFF |

Figure 10 shows the output of REALITY for this configuration.

1. **Filling the t-buffer**

   - This step is carried out in a manner identical to the previous illustration

2. **Rendering**

   i. The initial steps are the same as steps 1 through 5 of the algorithm described in the previous illustration. Therefore, we assume that the diffuse and specular color contributions – $C_d$ and $C_s$ respectively, have been calculated already

   ii. Calculate the inter-object reflection contribution as follows –
      a. Find the reflection of the primary ray $\vec{r}$ about the surface normal $\hat{n}$
      b. Call the classical ray-tracing algorithm with $\vec{I}$ as the initial point and $\vec{r}$ as the direction vector of the ray. Suppose the color contribution returned is $C_r$

   iii. Calculate the ambient contribution to the color $C_a$ (assuming some constant amount of ambient light)

   iv. Return the color for pixel P as $C_{total} = C_a + C_d + C_s + C_r$

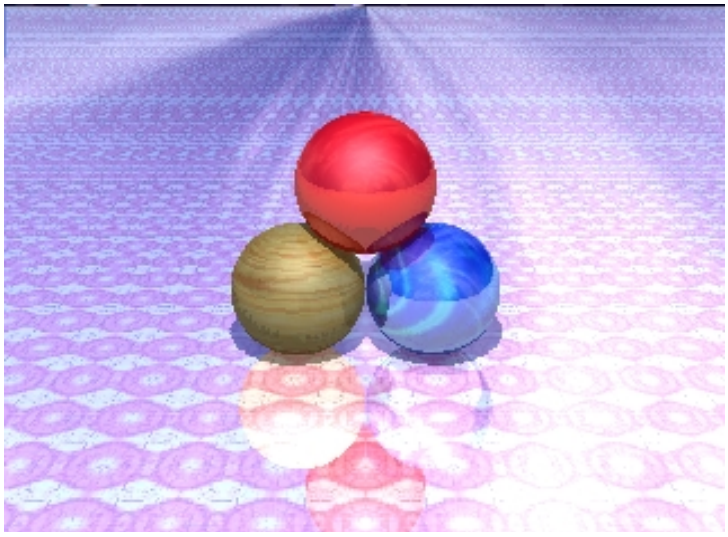

**Figure 11:** Output of REALITY for a simple scene with all features enabled. Note the additional aliasing of shadows due to the use of a shadow-buffer. Figure 18 shows this effect in more detail.

## *C. Scene 3*

The configuration of REALITY for this run is as follows –

| Feature | Status |
|---|---|
| Shadows | ON |
| Inter-Object Reflections | ON |
| Textures | ON |
| t-buffer | ON |
| Shadow-buffer | ON |

Figure 11 shows the output of REALITY for this configuration.

1. **Filling the t-buffer**

   - This step is carried out in a manner identical to the previous illustration

2. **Filling the shadow-buffer**

   i. Initialize all the entries of the shadow-buffer to the tuple (*null*, *infinity*) so that the object pointer is set to null and the intersection distance is set to infinity
   ii. Consider all the objects at a time. In this particular scene, there are only four objects - the ground and three spheres. Suppose the ground is considered first
   iii. Shoot all the shadow-rays which lead to an intersection with the ground and fill the corresponding shadow-buffer entries with the intersection information, namely, the object pointer *o* and the intersection distance *t*
   iv. Process the spheres in a similar fashion. Since the light source is present above the ground for this particular scene, the intersection distance for any point on the spheres is smaller than the value already present in the shadow-buffer at the corresponding location. This has the effect that all shadow-buffer entries which correspond to the ground as well as any of the spheres which lies lower and is obstructed from the view

of the light source, are filled with the sphere's information instead, overwriting the earlier entries corresponding to the ground present at those locations. This eventually leads to the shadow of the spheres on the ground

v. Now the shadow-buffer contains view-independent and complete shadowing information of the scene for the current light source

3. **Rendering**

i. The initial steps are the same as steps 1 through 4 of the algorithm described in the first illustration.

ii. **Shadow Computation:** Join the intersection point of the primary ray and the object hit, $\vec{I}$ to the light source and find the shadow-buffer entry corresponding to $\vec{I}$, as illustrated in equation (7). Suppose the entry points to the object $o'$. Further, suppose the object on which the intersection point $\vec{I}$ lies is o[8]

a. If $o \neq o'$, $\vec{I}$ is in the shadow of the light source and the light source's direct contribution to the radiance is 0. The reason is that, from the definition of the shadow-buffer the object $o'$ is nearer to the light source than $o$ for the current shadow-ray under consideration

b. If $o = o'$, knowing the position of the light source and the surface normal $\hat{n}$, find the specular and diffuse components of the color contribution from the light source - $C_s$ and $C_d$ respectively.

iii. Calculate the inter-object reflection contribution as follows –

a. Find the reflection of the primary ray $\vec{r}$ about the surface normal $\hat{n}$

b. Call the classical ray-tracing algorithm with $\vec{I}$ as the initial point and $\vec{r}$ as the direction vector of the ray. Suppose the color contribution returned is $C_r$

iv. Calculate the ambient contribution to the color $C_a$ (assuming some constant amount of ambient light)

v. Return the color for pixel P as $C_{total} = C_a + C_d + C_s + C_r$

# V. Results

The major objective of the T-BUFFER algorithm is to reduce the computational complexity of ray-tracing by reducing the number of ray-object intersection tests. For testing whether our algorithm did give such gain and to get numerical estimates of the gain involved, we implemented our own ray-tracing engine, named REALITY, which implements both the traditional ray-tracing engine and the T-BUFFER ray-tracing engine. Care was taken so that both the engines share the routines for intersection tests and other common features, so that the running times are truly representative of the number of intersection tests performed, and all other factors affecting the running time are distilled out.

[8] The object pointer *o* was found in the initial steps, by doing a t-buffer lookup as illustrated in step 2 of the rendering part of the algorithm described in section 4A

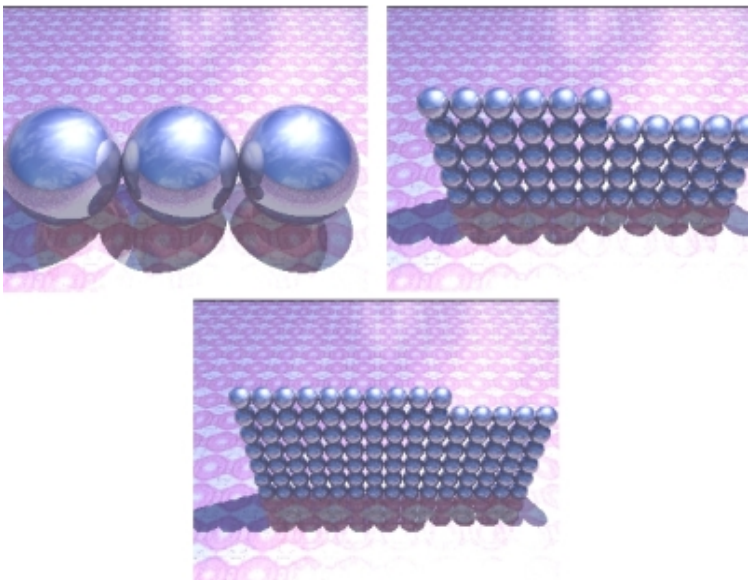

**Figure 12:** The procedural scene used to test the effect of the number of objects on the running times. The scene complexities for the three cases shown are 3, 50 and 100 respectively.

## *A. The test scene*

The main parameter for testing the efficacy of our approach is to test the variation in running times with respect to the number of objects in the scene. It was clear that no scene file could give us this flexibility. So we used a procedurally specified scene, which would be dynamically generated as per the specifications of the program. The input consisted in the number of objects desired, and the kind of objects that is, sphere, triangle etc. The scene that is generated consists of these objects stacked on top of each other so that they fill the viewport as much as possible. On increasing the scene complexity, the objects grow smaller so as to accommodate the increased number of objects. Figure 12 illustrates the scene generated for three different scene complexities with sphere as the object type.

## *B. Performance enhancements with the t-buffer algorithm*

The major factors which affect the performance of the T-BUFFER algorithm are the number of objects in the scene (Scene Complexity) and the resolution of the shadow-buffer. We analyzed the impact of varying both these parameters on the performance of the T-BUFFER algorithm. Since we wanted to analyze the variation in the computational efficiency for primary rays and shadows only, we kept other features such as reflections and textures disabled for these tests. We present these results and their detailed analysis next.

## *C. Performance enhancements due to t-buffer alone*

Figure 13 compares the running times of the traditional and the T-BUFFER algorithm, for varying number of objects in the scene (10-6000). Note that the shadow-buffer is turned off in this case. The object type is set to sphere. Figure 14 is a plot of the ratio of the running times for the classical and t-buffer algorithms for the scene versus the scene complexity. The graph shows a very complex pattern of growth in running times but the general trend points to a growth in the performance gain as the scene complexity increases, though the gain saturates somewhere close to 2.2 when the scene complexity approaches 5000.

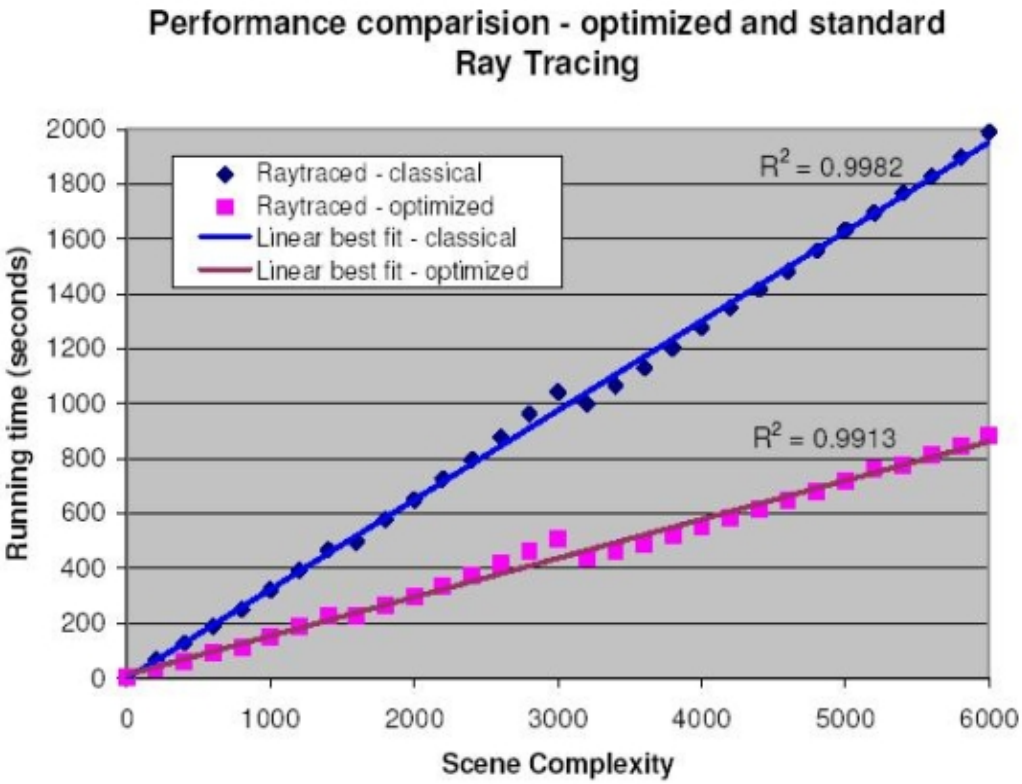


**Figure 13:** The running times of the classical algorithm and the T-BUFFER algorithm for varying number of objects in the scene, with the shadow-buffer disabled. The object type is set to sphere. The T-BUFFER algorithm consistently performs better.

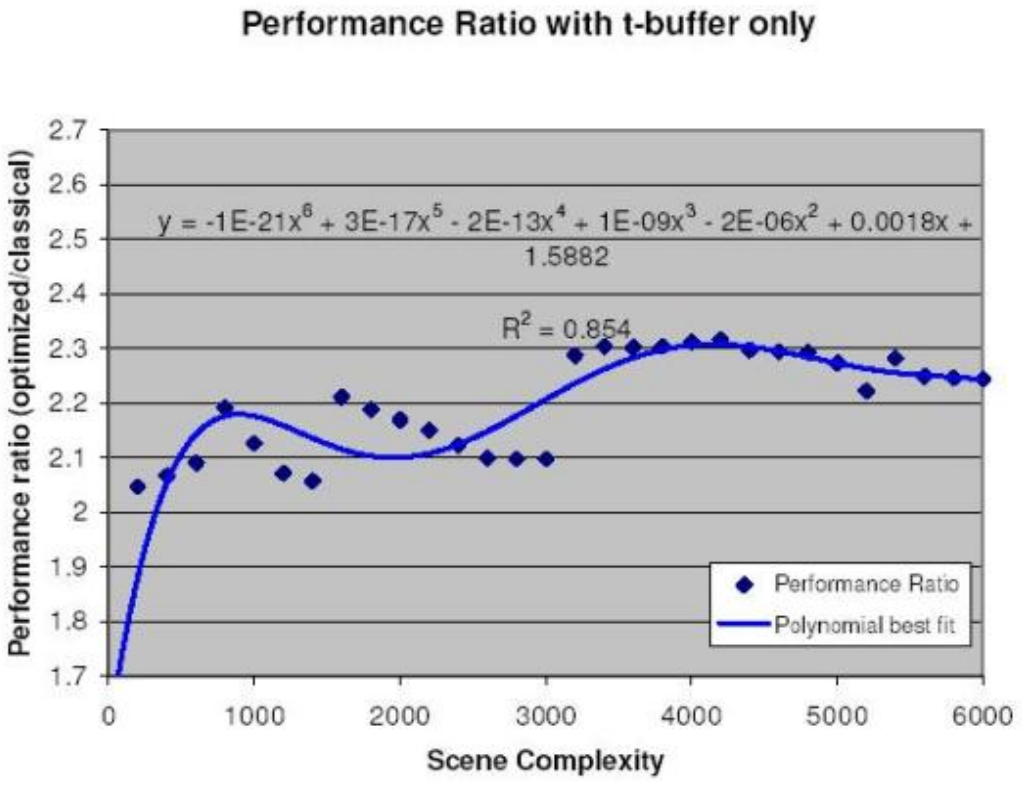


**Figure 14:** The growth in performance ratio as the complexity is increased, with shadow-buffer disabled.

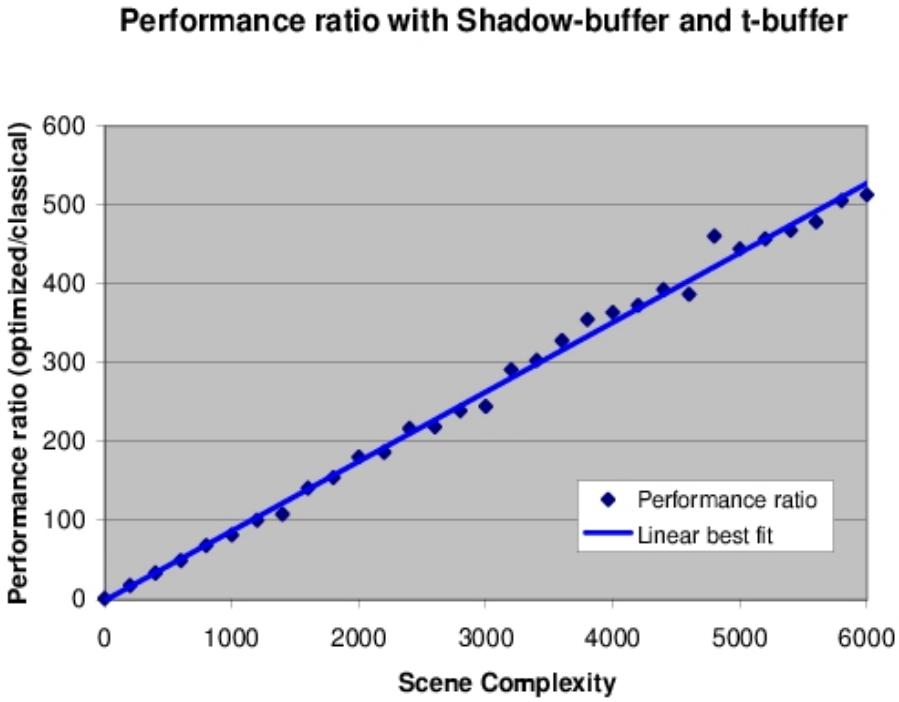


**Figure 15:** The increase in performance enhancement due to the T-BUFFER algorithm as the scene complexity is increased. The large gain may be attributed mainly to the shadow-buffer.

## *D. Performance enhancements due to t-buffer and shadow-buffer*

This is the main result of our research. Figure 15 shows a plot of the computational **gain** (inverse ratio of the running times) of the T-BUFFER algorithm with respect to the classical algorithm for scenes of varying complexity. The shadow-buffer resolution has been kept fixed at $400 \times 800$. It is clear that the computational gain is approximately linear in the scene complexity and the computational effort for the shadow-buffer algorithm is almost independent of the number of objects. The classical ray-tracing algorithm's running time is demonstrated to be linear in scene complexity, which is expected, as each ray involves an intersection test with all the objects in the scene, with a constant number of rays being shot. The important result is that the running time for the T-BUFFER algorithm shows very little variation with scene complexity, in comparison to the classical algorithm. Therefore, as more and more complex scenes are considered, the computational gain due to the T-BUFFER algorithm becomes more evident. This constancy in running time may be attributed to the fact that usually shadows due to a single light source do not occupy a majority of the area of the final rendering. With the shadow-buffer approach, calculations are performed only for the places in the scene where shadows may actually form, while the classical algorithm does calculations for all the points in the scene, even where there are no shadows. This leads to the large computational gain.

## *E. Effect of shadow-buffer resolution on performance*

The shadow-buffer resolution is a variable which is completely in the control of the user and has an important impact on the performance of the algorithm and the visual accuracy of the shadows in the rendering. Figure 16 shows a plot of the computational gain versus the scene complexity for a shadow buffer resolution of $1000 \times 2000$ and compares it with the plot obtained in the previous section. It is clear that the computational gain has fallen with the increased resolution.

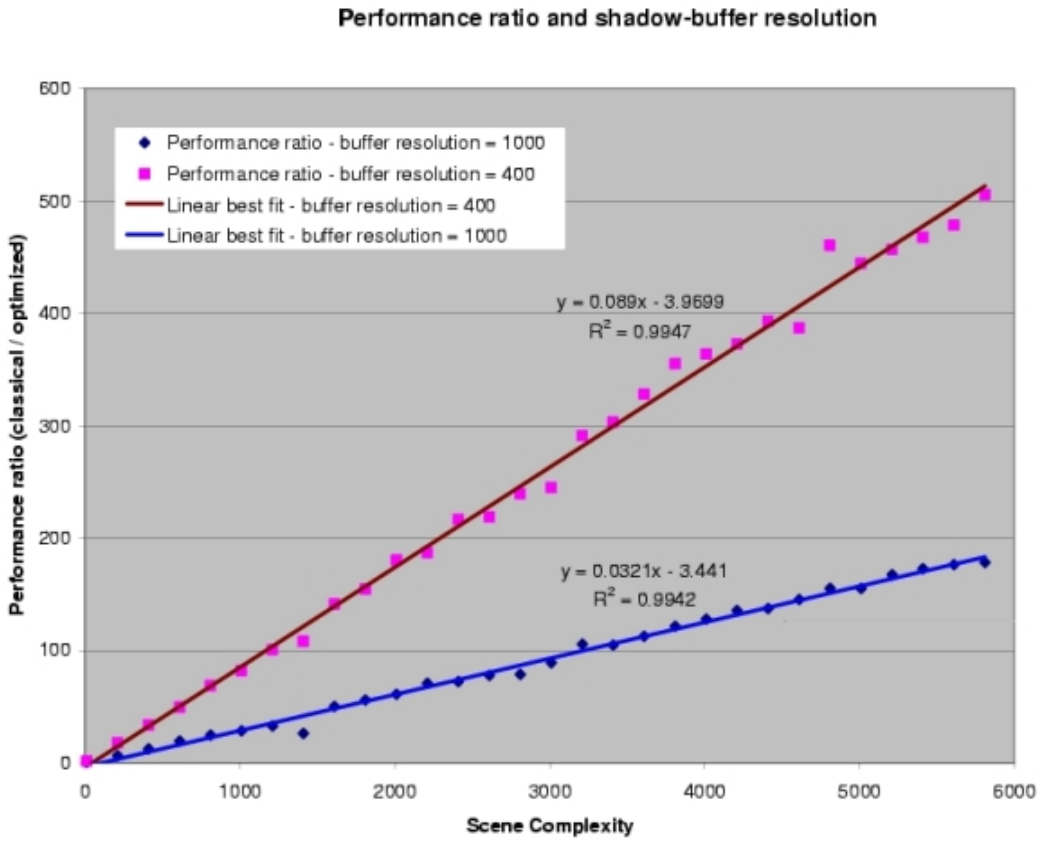


**Figure 16:** The effect of increase in the shadow-buffer resolution on the performance of the T-BUFFER algorithm. The gain still increases with an increase in the number of objects but not at the same rate as the case when the shadow-buffer resolution is kept at a lower value.

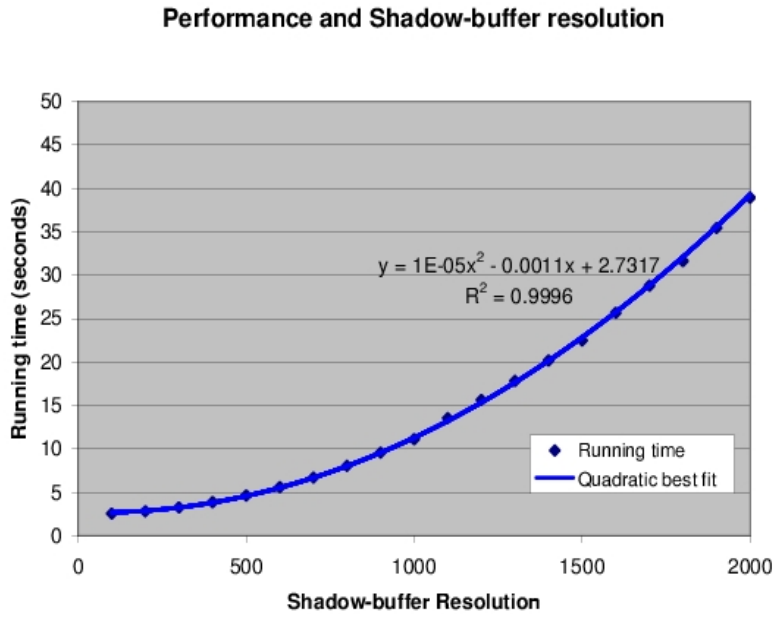


**Figure 17:** Variation in the running time due to change in shadow-buffer resolution. It is clearly seen that the performance drop is quadratic in the resolution of the shadow-buffer.

To study this effect more closely, the shadow-buffer resolution was varied, keeping the scene complexity fixed at 500. Figure 17 shows the variation in running time as the shadow buffer

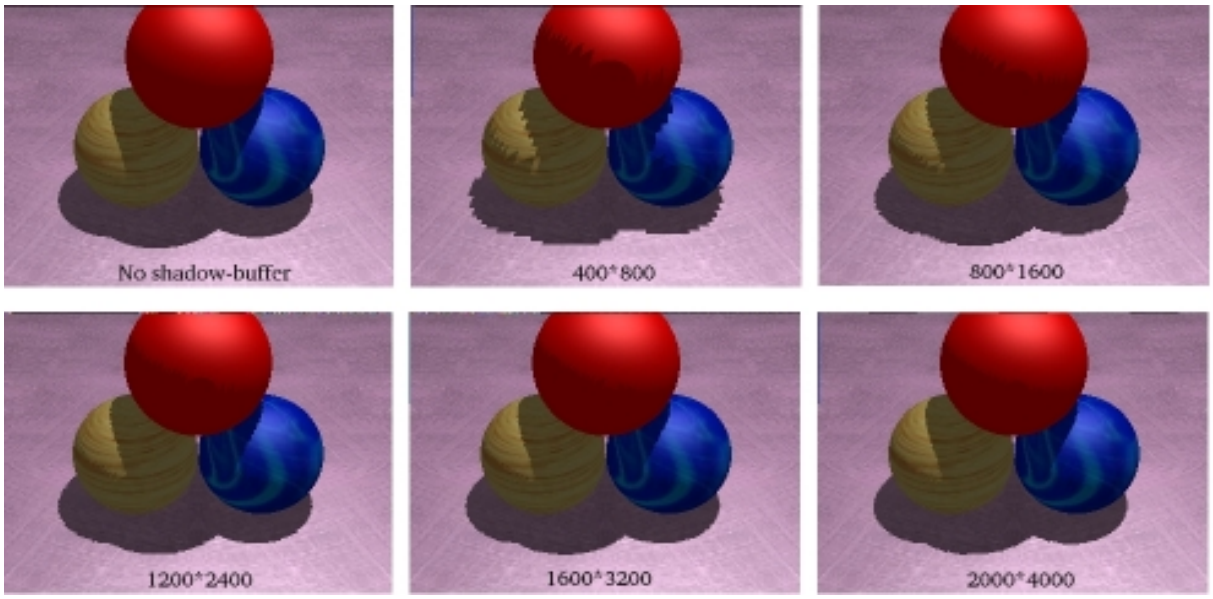


**Figure 18:** A demonstration of the effect of shadow-buffer resolution on the quality of output. The first rendering is the output of pure ray-tracing without employing any shadow-buffer. The other renderings have been done at successive shadow-buffer resolutions of $400 \times 800$, $800 \times 1600$, $1200 \times 2400$, $1600 \times 3200$ and $2000 \times 4000$ respectively. Note that the improvement in visual quality saturates at a resolution of $1200 \times 2400$ for the scene under consideration.

resolution is increased from $200 \times 400$ to $2000 \times 4000$. It is clearly seen that the curve is quadratic in resolution, as pointed to by the high degree of correlation (.9996) between the quadratic curve and the actual running times. This is in accord with theoretical predictions, since the number of entries in the shadow buffer grows quadratically in resolution, and each entry of the shadow-buffer needs to be filled as part of the t-buffer algorithm. This result shows that the shadow-buffer resolution must be chosen very judicially keeping another important factor in mind, the quality of the shadows generated - this is discussed next.

## *F. Effect of shadow-buffer resolution on the quality of output*

The shadow-buffer resolution has very noticeable effects on the quality of the shadows that are produced. Since the shadow-buffer quantizes the possible angles at which rays are shot from the light source, it introduces some aliasing artifacts in the shadows produced. These aliasing artifacts can be reduced by increasing the shadow-buffer resolution. Figure 18 demonstrates renderings of the same scene with the shadow-buffer resolution set to 400, 800, 1200, 1600 and 2000 respectively. The quality gains should be obvious from the images. However, increasing the resolution results in a quadratic drop in performance as was discussed in the previous section. Moreover, the improvement in appearance due to an increase in resolution is not uniform - It should be evident that the quality of shadows shows a large improvement as the resolution is increased from 400 to 1200, but the same kind of gain is not observed as the resolution is increased from 1200 to 2000. Therefore, the shadow-buffer resolution must be chosen very judiciously, keeping this speed-quality tradeoff in mind.

Recent developments in anti-aliasing techniques for rendering 3D scenes offer a very attractive solution to the above problem and may give reasonably anti-aliased shadows while keeping the shadow-buffer resolution at a relatively low value. While we have not implemented these techniques in this work, they may be easily incorporated into our approach. Some techniques for generating anti-aliased ray-traced renderings and shadows are discussed in [8,7].

# VI. Conclusion

The T-BUFFER algorithm shows a lot of promise as a technique for accelerating 3D ray-traced image synthesis, mainly in accelerating the generation of shadows. Moreover, it provides a speed-quality tradeoff for shadow generation which can be a very useful feature in applications where time is at a premium. One more advantage of the shadow-buffer is its viewpoint-independence. Therefore, in applications where the same scene is to be rendered from different viewpoints, the calculations for the shadow-buffer need not be repeated, which will make the process much more efficient.[9]

This algorithm shows the way to a very basic idea which is explicitly exploited in the algorithm to accelerate visibility-calculations as well as shadow-generation. This concept may be extended in the future to accelerate the processing of secondary rays in the recursive ray-tracing algorithm, to efficiently generate reflections as well. The T-BUFFER algorithm has the specific advantage that it always maintains a locality of reference in the scene, which is ensured by the way each object is processed separately to fill the t-buffer and the shadow-buffer. Hence, it can be easily combined with techniques which strive to accelerate rendering algorithms by using the memory occupied by the scene data "intelligently" as described by Pharr[5]. Moreover, the above property opens the way for methods which exploit object coherence to fill the buffers more efficiently.

# REFERENCES


[1] Appel, A., "Some Techniques for Shading Machine Renderings of Solids," SJCC, 1968, pp. 37-45.

[2] Whitted, T., "An Improved Illumination model for Shaded Display," CACM, 23(6), 1980, pp. 343-349.

[3] Kay, D.S., "Transparency, Refraction and Ray Tracing for Computer Synthesized Images," M.S. Thesis, Program of Computer Graphics, Cornell University, 1979.

[4] Bouknight, W.J., K.C.Kelly, "An algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources," SJCC, AFIPS Press, 1970, pp. 1-10.

[5] Pharr, Matt et. al., "Rendering Complex Scenes with Memory-Coherent Ray Tracing," Computer Graphics, 31(Annual Conference Series), August 1997, pp. 101-108.

[6] Kim, Sehyun, Sung-ye Kim, Kyung-hyun Yoon, "Study on the Ray-Tracing Acceleration Technique Based on the ZF-Buffer Algorithm," International Conference on Information Visualisation (IV2000), July 2000, pp. 393.

[7] Reeves, William T., David H. Salesin, Robert L. Cook, "Rendering antialiased shadows with depth maps," Computer Graphics (SIGGRAPH '87 Proceedings), July 1987, pp. 283-291.

[8] Painter, James, Kenneth Sloan, "Antialiased Ray Tracing by Adaptive Progressive Refinement," Proceedings of SIGGRAPH'89. In Computer Graphics 23, 3, July 1989, pp. 281-285.

[9] Kajiya,J T, "The Rendering Equation," SIGGRAPH'86, Computer Graphics, Vol. 20, No 4, 1986, pp. 133-142.


[9] One such application is the use of web-based "virtual 3D worlds", which users may interactively explore. For this interactive application, usual ray-tracing techniques would be too slow. However, if a pre-processed shadow-buffer is kept, it may be possible to implement a small ray-tracing engine based on the techniques discussed in this paper, which would do interactive renderings much more quickly and give the user the ability to control the speed of the engine depending on the capabilities of his/her computer.

[10] Bui-Tuong, Phong, “Illumination for Computer Generated Pictures,” CACM, 18(6), June 1975, pp. 311-317.

**Nikunj Raghuvanshi** obtained his B.Tech in Computer Science and Engineering from the Indian Institute of Technology Kanpur, India in May 2003. He is currently working as a Project Associate in the Media Lab Asia Chapter at the institute. His interests lie mainly in Rendering and Animation for Computer Graphics. He will be joining the University of North Carolina at Chapel Hill for a PhD in Computer Science in August 2004.

**Prof. Sanjay G. Dhande** obtained his PhD in Mechanical Engineering from the Indian Institute of Technology Kanpur, India in 1974. He joined the institute as an Assistant Professor in 1979, after a brief stint abroad. At present he is the Director of the institute and Professor in Computer Science and Mechanical Engineering. He has taught at Virginia Tech, University of Florida at Gainesville and other universities in the USA. He has authored/co-authored many books in the area of CAD/CAM with reputed publishers like Wiley Eastern Ltd. and has many publications in international conferences in USA and Europe. He has an extensive set of research publications in the areas of CAD/CAM and in other areas of Computer Science and Engineering.