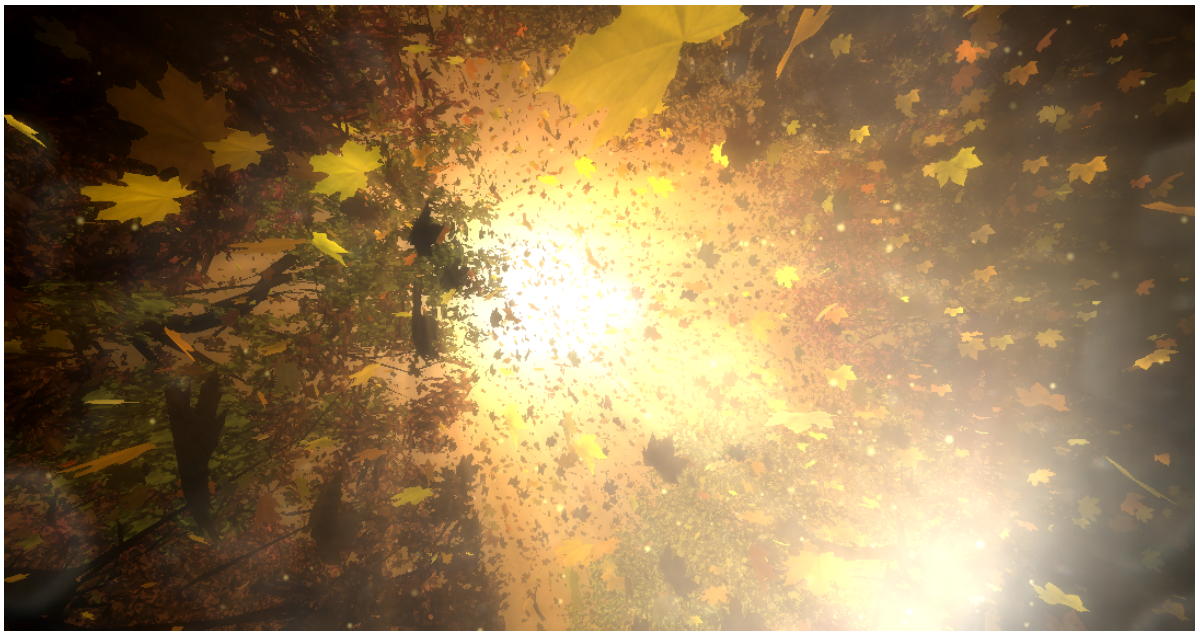
CSCI 580, Prof. Saty Raghavachary

Crepescular Rays using Ray Tracing

Team Uncanny Valley

Sanskriti Lastname, Saurabh Mistry, Sreepada Rao Singeetham, Tushar Tiwari

November 28, 2014



1 Objective

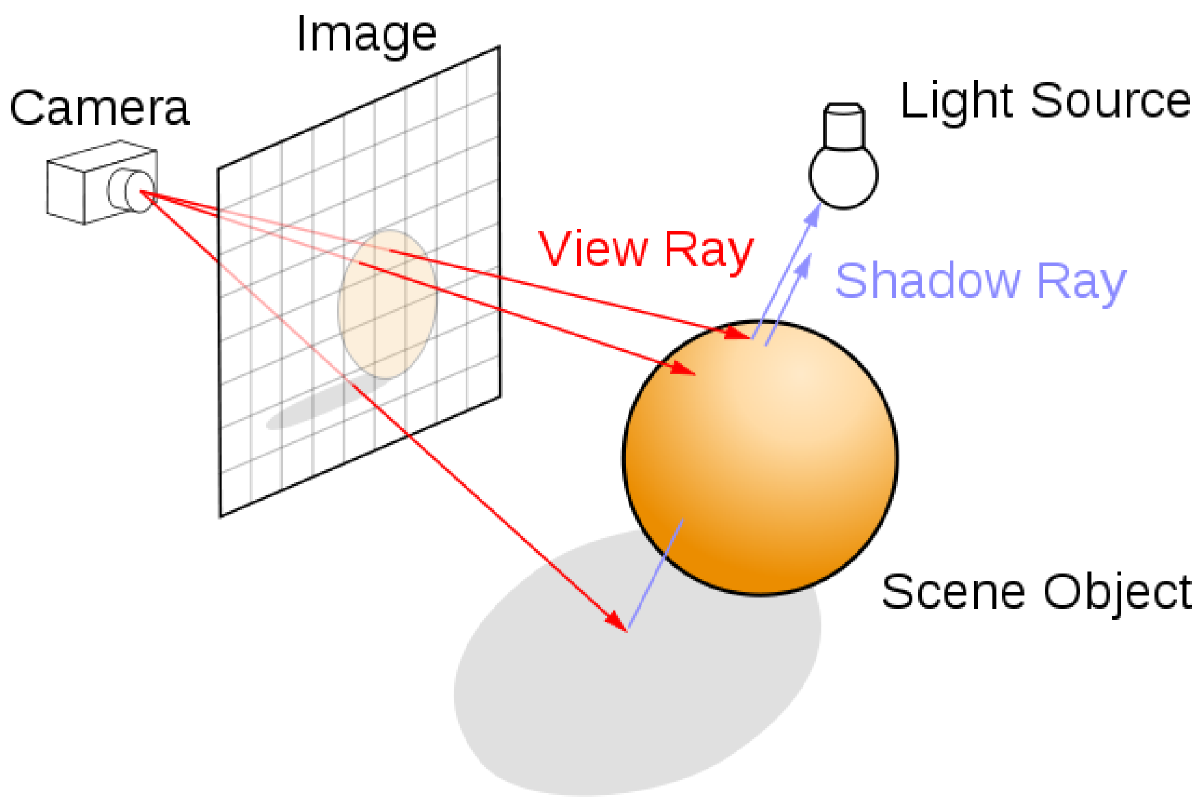
To achieve the above render, we make use of two well-known techniques namely Ray Tracing and Volumteric Lighting. Ray Tracing is a technique for generating an image by tracing the path of light. Volumetric Lighting is a technique that allows the viewer to see beams of light called crepescular rays.

2 INTRODUCTION

The Ray Tracing algorithm very much mimics the procedure of observing the color of an object in nature. A light source emits a ray of light which travels, eventually, to a surface that interrupts its progress. One can think of this "ray" as a stream of photons traveling along the same path. There are two basic ideas for ray tracing.

Forward ray tracing and backward ray-tracing. Forward ray-tracing is how the color of an object is observed by the eye in real life. That is, a ray of light is reflected of an object and travels into the eye producing an image. However, this algorithm is impractical because there will be many rays that will miss the camera and will not be useful to us. Thus, there will be a huge computational overhead.

For CG purposes, we make use of the backward ray-tracing algorithm. The basic idea is shoot a ray from the camera through each pixel of the image plane. If this ray intersects with an object, we send out another ray in the direction of our light source to determine if the object is under a shadow of another object. We also send out a reflected ray to obtain the reflection of another object if the object is capable of reflections.



3 Implementation

3.1 Scene Creation and Obj Format

Since most objs out there use the v, vt, f and quadrilateral format, our program had to be modified to read data in that format and modify it into triangles.

Moreover, a single leaf obj wouldn't suffice for the purposes of this project. So we took the same leaf obj, randomly translated it, rotated it and scaled it to create a scene that represented falling leaves. All these combinations of leaves in different orientations were merged into one singe obj file that could now depict the whole screen.

3.2 Backward Ray Tracing

3.2.1 Compute the Camera Rays

We represent a ray by obtaining the ray's origin and its direction. The first ray we look at is the viewing or camera ray. This ray is computed by making use of the camera position and the current pixel which has to be coloured. We first transform the pixel coordinates to $x, y \in [-1, 1]$.

$$i = \frac{x * 2}{imagePlaneWidth} - 1$$

$$j = \frac{y * 2}{imagePlaneHeight} - 1$$

The ray line equation can be written as:

$$Somestuff.$$

3.2.2 Compute the intersection of the camera ray with the plane of the triangle

Since there is no way knowing which triangle the ray will intersect with, we have to try out all triangles. To check if the ray intersects with the plane of the triangle is easy. By finding out the normal to the triangle and taking the dot product of the ray and this normal we can verify if the triangle is parallel to the ray or starts behind the triangle or neither of those.

To begin with we need to make sure that we get the normal outwards rather than inwards. This will ensure that we get the correct normals. We first arrange all vertices in a counter-clockwise manner. If $v_0, v_1 and v_2$ are the vertices in counter-clockwise order, then the normal is obtained by,

$$n = \overrightarrow{(v_1 - v_0)} \times \overrightarrow{(v_2 - v_0)}$$

$$\triangle_{normal} = \hat{n} = \frac{n}{|n|}$$

Then the dot product of the $\triangle_{normal}$ and the ray can provide us with information whether the $\triangle$ is parallel to the ray or not.

$$\triangle_{normal} \cdot \overrightarrow{r_c} = 0 \implies \triangle \parallel \overrightarrow{r_c}$$

$$\triangle_{normal} \cdot \overrightarrow{r_c} < 0 \implies \triangle is behind \overrightarrow{r_c}$$
$$\triangle_{normal} \cdot \overrightarrow{r_c} > 0 \implies \triangle \nparallel \overrightarrow{r_c}$$

We are interested in the last condition because this signifies that the camera ray intersects the plane of the △.

3.2.3 Check if intersection point is inside the Triangle

To obtain the point of intersection we must first compute the triangle distance:

$$d(\triangle) = \triangle_{normal} \cdot v_0$$

Then, the following equation gives us the distance to the plane:

$$d(plane) = -1 \times \frac{\triangle_{normal} \cdot (\overrightarrow{r_c} + (\triangle_{normal} \times -d(\triangle)))}{\triangle_{normal} \cdot \overrightarrow{r_c}}$$

Next we need to find the point that intersects with the plane. This point, say q, is found by:

$$q = d(plane) \cdot \overrightarrow{r_c} + r_c$$

Now, that we have the point, we need to verify if this point lies inside the △.

$$\frac{Area(q, v_1, v_2) + Area(q, v_0, v_2) + Area(q, v_0, v_1)}{Area(v_0, v_1, v_2)} \approx 1$$

If the above condition holds then our point is inside the △.

3.3 Volumetric Shadows

3.4 Lens Flare

We made use of an image downloaded from the internet. This image was drawn over the canvas to obtain the lens-flare effect.

4 Challenges

- One of the biggest challenges in the project is the slow computation time of ray tracing. Since every pixel is rendered against every triangle in the scene. The computation time is very huge.
- We were stuck at one point when finding whether the point on the ray-plane intersection is inside or outside the triangles. We had to try various ways to get the ray tracing working.

5 Conclusion

While researching and implementing Ray-Tracing we were exposed to many methods that showed us how to implement photorelaistic images. We were also made aware of the enormous computations done in the ray-tracing algorithm. We also investigated several ways to check if the point exists in the triangle or not.

6 References

-