Parallelization of shadow projection in synthetic three–dimensional scenes

Javier Hernández–Aceituno
Departamento de Ingeniería Informática, Universidad de La Laguna.
Avda. Astrofísico Fco. Sánchez s/n, 38204 Canary Islands, Spain
E-mail: javier@isaatc.ull.es

Abstract. The generation of projected shadows in synthetic three–dimensional scenes is a complex procedure which, because of its computational cost, has yet to be perfected. In this work, we present an algorithm that simplifies this process using the basic three–dimensional geometry of three elements: a point cloud which defines the object that casts the shadow, the position of the light source, and a projection plane. Shadows are generated as irregular polygons with as many vertexes as the size of their corresponding generative point cloud. The parallelization potential of the resulting algorithm is then studied for real–time applications.

1. Introduction
Light effects, such as reflection, refraction or shadow projection, are very difficult to simulate when computing synthetic three–dimensional scenes, because of their computational cost. The accurate reproduction of light behavior for each image pixel – a procedure known as ray–tracing [1, 2] – often requires recursion, as each beam may be reflected or refracted on several objects along its path. Therefore, this technique requires a long processing time for each image, and can not be used in real time applications, for which developers favor inaccurate heuristic methods [3] or simplified approximations [4, 5].

In this work, we describe an algorithm that produces the projected shadows of three–dimensional objects, which are defined as point clouds, given the positions of the light sources that may affect them and the geometric planes that are affected by the projection. This procedure is explained in section 2, the computational cost of its application is studied on section 3 and the results of its parallelization are analyzed on section 4.

2. Algorithm definition
According to the basic principles of optical physics, given the three–dimensional positions of a point light source \( L \) and a hypothetical point object \( P \), the shadow that the latter casts over any surface must be aligned with the direction of the light beam that produces it, that is, \( P − L \). If we name the projection of the object as \( X \), we can deduce that

\[
X = P + \alpha \times (P − L),
\]

where \( \alpha > 0 \) depends on the desired projection plane; if \( \alpha < 0 \), the projection plane is located at a relative position to \( P \) where a shadow cannot be casted, such as between \( P \) and \( L \), and \( X \) would not exist.
A projection plane $Z$ will be defined by its normal vector $N$ and any point $Q$ which belongs to it, such that $N \cdot (Y - Q) = 0$ for every point $Y \in Z$ [6]. Since the projection point $X$ must necessarily belong to the projection plane, using the properties of the dot product [7], the value of $\alpha$ can be obtained as follows:

$$N \cdot (X - Q) = N \cdot (P + \alpha \times (P - L) - Q) = 0 \Rightarrow \alpha = \frac{N \cdot (Q - P)}{N \cdot (P - L)}$$  \hspace{1cm} (2)

Notice how, if the light beam $P - L$ is parallel to $Z$, $N \cdot (P - L) = 0$ and $P$ does not have a valid projection. This calculation must be executed once for each object point, light source, and projection plane. Since any image is bound to contain several of these, we must optimize (2) to reduce its computational cost as much as possible. Calculating $\alpha$ as

$$\alpha = \frac{N \cdot Q - N \cdot P}{N \cdot P - N \cdot L},$$  \hspace{1cm} (3)

allows us to compute $N \cdot Q$ only once per projection plane, $N \cdot L$ once per light source, and $N \cdot P$ once per object point.

Some restrictions of this technique must be considered. Our algorithm considers hypothetically boundless flat surfaces as projection planes, and the projection of points of different polygons of an object may overlap, which would cause undesired effects in case of semitransparent shadows. Users of this method must assess the impact of these limitations on their image generation procedures.

3. Experiments

The point cloud which defines a complex three–dimensional object may contain a very large number of points. For the sake of simplicity, we will study the behavior of our algorithm depending on the amount of points to project, regardless of whether they belong to different objects.

Our method was coded in C/C++ language [8, 9], using the OpenGL three–dimensional graphic library [10] and the OpenMP parallelization interface [11], as shown in Fig. 1. We defined a three–dimensional Point structure, with all the required mathematical operators (vector addition and subtraction, dot product and scalar multiplication) and a Plane structure, which includes two Point elements: a normal vector $n$ and a plane point $q$. Function projection receives three parameters: a vector of light sources $l$, a vector of projection planes $z$, and a vector of object points $p$; it returns a matrix of shadow projections $x$.

The generation time for a single image depends on its target media, but a minimum frame rate of 30 Hz is required, or else the animation would not look right to human viewers [12]. This sets a maximum restriction of 33.3 milliseconds per frame if our algorithm is to be used in real time.

In order to observe the efficiency of our method on slow computers, all experiments were done in a virtual machine. We observed unacceptable generation times for scenes with more than 10000 object points; however, since the shadows of static objects can be calculated offline and complex objects do not usually require more than 10000 points, our algorithm works correctly in real time.

4. Parallelization

Since our function contains three nested loops, it can be easily parallelized. However, although the number of object points in a scene may be very high, there are normally few light sources and projection planes. This may cause the communication between cores and memory accesses to increase the computational cost of the procedure and undo the parallelization enhancement. [13]
std::vector<Point> projection(const std::vector<Point> l,  
const std::vector<Plane> z,  
const std::vector<Point> p) {
    std::vector<Point> x;
    x.reserve(p.size() * l.size() * z.size());
    int i, j, k;
    float nq, np, nl, alpha;
    for (i = 0; i < z.size(); i++) { // For each projection plane
        nq = z[i].n * z[i].q;
        for (j = 0; j < l.size(); j++) { // For each light source
            nl = z[i].n * l[j];
            for (k = 0; k < p.size(); k++) { // For each object point
                np = z[i].n * p[k];
                if (np == nl) continue; // p[k] does not cast a shadow
                alpha = (nq - np) / (np - nl);
                if (alpha < 0) continue; // p[k] does not cast a shadow
                x.push_back(p[k] + alpha * (p[k] - l[j]));
            }
        }
    }
    return x;
}

Figure 1. Sequential C++/OpenGL code for shadow projection.

To prove this point, we tested different configurations: parallelizing all loop levels (projection planes, light sources, and object points), parallelizing only the innermost loop (object points), and, for the sake of comparison, parallelizing only the two outer loops (projection planes and light sources). The time results for a four core virtual machine are shown in Fig. 2.

A full parallelization was found to improve the image generation time for over two projection planes, but to increase it otherwise. As expected, an innermost parallelization solved this problem and produced the best results for most cases, especially for large amounts of object points. An outer parallelization was found less efficient than the rest of options, including sequential executions, for most configurations.

5. Conclusion
The proper parallelization of our algorithm depends on the complexity of the scene to be generated. Parallelization will allow for greater amounts of visible objects, but applying it to all loops will only be useful when more than one light source and one projection plane appear in the image.

A greater number of light sources and projection planes will however reduce the allowable amount of object points, even when parallelizing. We observed that in this situation, full parallelization and innermost parallelization behave similarly. Therefore, an innermost parallelization is advisable for scenes which include unknown amounts of object points.

References
[1] Appel A 1968 AFIPS Spring Joint Comp. Conf. (ACM) pp 37–45
[2] Reif J H, Tygar J D and Yoshida A 1994 Discrete Comput. Geom. 11 265–88
[3] Laine S, Aila T, Assarsson U, Lehtinen J and Akenine-Möller T 2005 ACM Trans. Graph. 24 1156–65
[4] Crow F C 1977 SIGGRAPH Comput. Graph. 11 242–48
[5] Porquet D, Dischler J and Ghazanfarpour D 2005 SIGGRAPH Comput. Graph. (ACM) pp 213–20
[6] Anton H 2010 Elementary linear algebra (John Wiley & Sons)
[7] Istrîtescu V I 1987 Inner product structures: theory and applications vol 25 (Springer)
Figure 2. Generation times for different scene configurations, for sequential executions (seq), full parallelization (full), innermost parallelization (inner) and outer parallelization (outer).

[8] Ritchie D M 1993 ACM SIGPLAN Notices 28 201–208
[9] Stroustrup B 2000 The C++ Programming Language (Addison-Wesley Longman Publishing Co., Inc.)
[10] Woo M, Neider J, Davis T and Shreiner D 1999 OpenGL Programming Guide: The Official Guide to Learning OpenGL (Addison-Wesley Longman Publishing Co., Inc.)
[11] Chapman B, Jost G and Pas R 2007 Using OpenMP (The MIT Press)
[12] Boff K R, Kaufman L and Thomas J P Cognitive processes and performance Tech. rep.
[13] Rünger G and Rauber T 2013 Parallel Programming for Multicore and Cluster Systems 2nd ed (Springer)