Интерактивная трассировка лучей на графическом процессоре

Денис Боголепов, Виталий Трушанин, Вадим Турлапов
Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики, Нижний Новгород, Россия
bogdencmc@inbox.ru, vitaly.trushanin@rambler.ru, vadim.turlapov@cs.vmk.unn.ru

Abstract
High-performance GPU-implementation of the backward ray tracing algorithm is proposed. High performance is provided by new ray-primitive intersection algorithms without dynamic branches, uniform grid accelerating structure with “proximity cloud” information; by some optimizing techniques for ray tracing algorithm on GPU architecture. The implementation was done as cross platform on GLSL and provides real-time rendering for scenes with 300K and more triangles.

Keywords: *Ray Tracing, Interactive Graphics, GPGPU, GPU Ray Tracing, GLSL*

1. ПРЕДШЕСТВУЮЩИЕ РАБОТЫ

Задача реализации алгоритма обратной трассировки лучей в реальном времени имеет достаточно богатую историю [1]-[5], новый импульс которой придало появление графических процессоров, способных выполнять вычисления общего назначения. Расширенные возможности программируемости графических процессоров и появление технологий NVIDIA CUDA и AMD Stream привели к пересмотру ряда принципов, заложенных в реализацию трассировки лучей для графического процессора. Интересным примером служит работа [6], в которой авторы одни из первых реализовали полноценный алгоритм трассировки лучей в *одном* фрагментном шейдере. Среди последних работ, использующих технологию NVIDIA CUDA, следует отметить проект [7], в которой трассировка лучей реального времени реализована с учетом теней, отражений и преломлений для достаточно сложных сцен (сотни тысяч треугольников).

Цель данной работы – реализация алгоритма обратной трассировки лучей на графическом процессоре с поддержкой всех основных возможностей метода: визуализация теней, отражений, преломлений, поддержка прозрачных объектов и текстурирования. В качестве платформы выбран графический интерфейс OpenGL и связанный с ним язык шейдеров OpenGL Shading Language (GLSL).

2. ТРАССИРОВКА ЛУЧЕЙ НА GPU

Каждая реализация трассировки лучей характеризуется набором *базовых геометрических примитивов*, поддержкой *статических* или *динамических* компьютерных сцен, а также применяемой *ускоряющей структурой*.

В данной работе предполагается, что все объекты сцены представлены набором *треугольников*, что увеличивает эффективность [8]. Кроме того, рассматриваемая реализация предназначена для визуализации сцен со *статической* геометрией, однако источники света и наблюдатель могут произвольно менять свое положение в пространстве во время визуализации.

Наряду с традиционной *регулярной сеткой* в данной работе использовалась регулярная сетка с информацией о *близости* (*proximity cloud*), позволяющая повысить эффективность поиска точек соударения [4]. Для этого всем пустым вокселям сетки приписывается расстояние до *ближайшего* вокселя, содержащего объекты сцены.

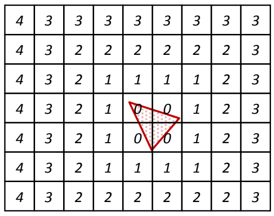

Рисунок 1. Пример регулярной сетки с информацией о близости *непустых* вокселей

Данные значения могут быть использованы во время прохода сетки для быстрой обработки больших пустых областей.

2.1 Алгоритмы ускоряющих структур

2.1.1 Пересечение треугольника и вокселя

В работе [9] предложен следующий эффективный алгоритм, основу которого составляет теорема “*О разделяющих осях*” двух *выпуклых* многогранников A и B. В качестве многогранника A выступает параллелепипед P, стороны которого параллельны осям системы координат, однозначно определяемый положением центра $\vec{c}$ и вектором половинных размеров $\vec{h}$. В качестве B - треугольник T, заданный своими вершинами $\vec{u}_0$, $\vec{u}_1$ и $\vec{u}_2$. Для упрощения теста вершины треугольника преобразуются в *локальную* систему координат параллелепипеда: $\vec{v}_i = \vec{u}_i - \vec{c}$, $i \in \{0, 1, 2\}$. Согласно теореме, существует 13 осей, вдоль которых параллелепипед P и треугольник T могут быть разделены:

- [*3 теста*] $\vec{e_0} = (1, 0, 0)$, $\vec{e_1} = (0, 1, 0)$, $\vec{e_2} = (0, 0, 1)$ – нормали к граням параллелепипеда (тест на пересечение параллелепипеда P с *минимальным* ограничивающим параллелепипедом треугольника T).
- [*1 тест*] $\vec{n}$ – нормаль к треугольнику T (тест на пересечение плоскости T с параллелепипедом P).
- [*9 тестов*] $\vec{a}_{ij} = \vec{e_i} \times \vec{f_j}$, $i, j \in \{0, 1, 2\}$, где $\vec{f_0} = \vec{v}_1 - \vec{v}_0$, $\vec{f_1} = \vec{v}_2 - \vec{v}_1$, $\vec{f_2} = \vec{v}_0 - \vec{v}_2$ – ребра преобразованного треугольника T (ниже рассмотрен случай $i = j = 0$).

Если все перечисленные выше тесты пройдены, то нет ни одной разделяющей оси, и P *пересекается* с T.

Для теста из второго пункта достаточно рассмотреть *две* вершины, лежащие на концах диагонали, направление которой наиболее близко к нормали $\vec{n}$:

```
vec3 n = cross ( f0, f1 );
float d = -dot ( n, v0 );
vec3 p0 = -sign ( n ) · h;
if ( dot ( n, p0 ) < d ) {
    vec3 p1 = sign ( n ) · h;
    if ( dot ( n, p1 ) > d ) return “пересечение есть”;
}
return “пересечения нет”;
```

Рассмотрим один из девяти тестов из третьего пункта при $i = j = 0$. В этом случае будем иметь $\vec{a}_{00} = \vec{e_0} \times \vec{f_0} = (0, -\vec{f}_{0z}, \vec{f}_{0y})$. Проекцией треугольника T на потенциальную разделяющую ось (для краткости обозначим ее символом $\vec{a}$) является отрезок с концами в точках $min(p_0, p_1, p_2)$ и $max(p_0, p_1, p_2)$:

$p_0 = \vec{a} \cdot \vec{v}_0 = v_{0z} \cdot v_{1y} - v_{oy} \cdot v_{1z}$,

$p_1 = \vec{a} \cdot \vec{v}_1 = v_{0z} \cdot v_{1y} - v_{0y} \cdot v_{1z} = p_0$,

$p_2 = \vec{a} \cdot \vec{v}_2 = (v_{1y} - v_{0y}) \cdot v_{2z} - (v_{1z} - v_{0z}) \cdot v_{2y}$.

Однако в рассматриваемом случае $p_0 = p_1$, что позволяет определять только $min(p_0, p_2)$ и $max(p_0, p_2)$, используя аппаратные функции *min* и *max*. Затем вычислим “радиус” r проекции параллелепипеда P на ось $\vec{a}$, учитывая, что $a_x = 0$:

$$r = h_x \cdot |a_x| + h_y \cdot |a_y| + h_z \cdot |a_z| = h_y \cdot |a_y| + h_z \cdot |a_z|.$$

Если все необходимые величины вычислены, тест сводится к проверке следующих двух условий:

```
if ( min ( p0, p1 ) > r or max ( p0, p1 ) < -r ) return “пересечения нет”;
```

2.1.2 Формирование регулярной сетки

Построение регулярной сетки предполагает задание ограничивающего параллелепипеда компьютерной сцены, который определяется своей *минимальной* и *максимальной* точкой B_{min} и B_{max} соответственно. Для построения равномерной сетки с числом разбиений m, n, k вдоль осей x, y и z соответственно можно использовать следующий эффективный алгоритм:

```
foreach ( Triangle T from Triangles ) {
    vec3 Tmin = vec3 ( min ( T.u0.x, T.u1.x, T.u2.x ),
                       min ( T.u0.y, T.u1.y, T.u2.y ),
                       min ( T.u0.z, T.u1.z, T.u2.z ) );
    vec3 Tmax = vec3 ( max ( T.u0.x, T.u1.x, T.u2.x ),
                       max ( T.u0.y, T.u1.y, T.u2.y ),
                       max ( T.u0.z, T.u1.z, T.u2.z ) );
    vec3 start = floor ( ( Tmin - Bmin ) / VoxelSize );
    vec3 final = ceil ( ( Tmax - Bmin ) / VoxelSize );
    start = clamp ( start, vec3 (1, 1, 1), vec3 (m, n, k) );
    final = clamp ( final, vec3 (1, 1, 1), vec3 (m, n, k) );
    for i, j, k ∈ [start, final]
        if ( T ∩ Voxel[i, j, k] ≠ ∅ )
            Voxel [i, j, k].Add ( T );
}
```

Для каждого треугольника определяется его *минимальный* ограничивающий параллелепипед с концевыми вершинами T_{min} и T_{max}. Данный параллелепипед используется для определения номера начального (*start*) и конечного (*final*) вокселя из окрестности треугольника, включающей только те воксели сетки, с которыми указанный треугольник может пересекаться. Далее треугольник тестируется на пересечение с каждым вокселем данной окрестности и, если пересечение имеет место, добавляется в список объектов вокселя.

2.1.3 Обход регулярной сетки

В работе [10] дается эффективный алгоритм обхода регулярной сетки, который включает себя два этапа: *инициализация* (*initialization*) и *пошаговый обход* (*incremental traversal*).

Инициализация. Номер начального вокселя на пути луча заносится в целочисленный вектор *voxel*:

```
ivec3 voxel = floor ( ( ray.Origin - Bmin ) / VoxelSize );
```

Вектор *step* содержит знак приращения номера вокселя при прохождении луча через его границу вдоль осей x, y и z соответственно:

```
ivec3 step = sign ( ray.Direction );
```

Время прохождения лучом одного вокселя вдоль осей x, y и z записывается в вектор *delta*:

```
vec3 delta = VoxelSize / abs ( ray.Direction );
```

Наконец, вычисляется время соударения луча с ближайшей гранью вокселя вдоль осей x, y и z:

```
vec3 out = delta * max ( step, 0 ) -
         mod ( ray.Origin - Bmin, VoxelSize ) / ray.Direction;
```

Наименьшее из трех найденных значений времени t определяет расстояние, которое следует пройти вдоль луча до соударения с границей вокселя.

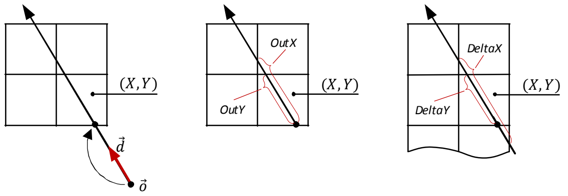

Рисунок 2. Инициализация регулярной сетки

Пошаговый проход. Этап пошагового прохода регулярной сетки выполняется весьма просто: в качестве следующего вокселя всегда выбирается *ближайший*:

```
vec3 next;
float min;
do {
    next = (0, 0, 1);
    min = out.z;
    if ( out.x < out.y )
        if ( out.x < out.z ) {
            next = (1, 0, 0);
            min = out.x;
        }
    else
        if ( out.y < out.z ) {
            next = (0, 1, 0);
            min = out.y;
        }
    Проверка всех треугольников текущего вокселя
    out += delta * next;
    voxel += step * next;
} while ( min < final );
```

Следует обратить внимание на важную особенность алгоритма: ближайшая точка соударения с треугольниками рассматриваемого вокселя может находиться *вне* данного вокселя. Предложенная программная реализация алгоритмов в значительной мере векторизована, что может дать существенный выигрыш в производительности как для графических, так и для центральных процессоров.

2.1.4 Формирование регулярной сетки близости

Построение регулярной сетки с информацией о близости отличается от формирования обычной регулярной сетки одним дополнительным шагом – построением *карты расстояний*. Для решения данной задачи был использован эффективный алгоритм [13].

2.1.5 Проход регулярной сетки близости

Отличие состоит в том, что в качестве следующего выбирается воксель, который находится на расстоянии $max(D, 1)$ от текущего вдоль направления распространения луча, где D – значение соответствующего элемента карты расстояний:

```
vec3 next;
float min;
do {
    ...
    Проверка всех треугольников текущего вокселя
    int proximity = Proximity ( voxel );
    for ( int i = 0; i < proximity - 1; i++ ) {
```

```
        if ( out.x < out.y )
            if ( out.x < out.z )
                next += (1, 0, 0);
            else
                next += (0, 0, 1);
        else
            if ( out.y < out.z )
                next += (0, 1, 0);
            else
                next += (0, 0, 1);
        out = delta * next;
    }
    voxel += step * next;
} while ( min < final );
```

2.2 Алгоритмы пересечения

2.2.1 Пересечение луча с параллелепипедом

Алгоритм используется для вычисления *начального* и *конечного* времени соударения луча с ограничивающим параллелепипедом сцены (`start` и `final` соответственно). В работе [11] дается эффективный алгоритм на основе "*полос*" ("*slabs*"), где под полосой подразумевается область пространства, заключенная между двумя плоскостями. Если начальное время соударения с наиболее удаленной полосой окажется больше конечного времени соударения с ближайшей полосой (`start` > `final`), то луч *не пересекается* с параллелепипедом:

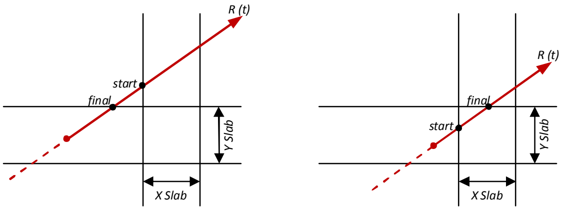

Рисунок 3. Варианты расположения луча и параллелепипеда

Программная реализация алгоритма была оптимизирована за счет использования свойств стандарта IEEE 754 и векторных возможностей современных центральных и графических процессоров:

```
vec3 omin = ( B_min - ray.Origin ) / ray.Direction;
vec3 omax = ( B_max - ray.Origin ) / ray.Direction;
vec3 tmax = max ( omax, omin );
vec3 tmin = min ( omax, omin );
final = min ( tmax.x, min ( tmax.y, tmax.z ) );
start = max ( tmin.x, max ( tmin.y, tmin.z ) );
return start < final and final > 0;
```

Приведенный выше псевдокод выполняет *12* элементарных арифметических операций и *12* операций сравнения и не содержит *явных* ветвлений, что повышает эффективность при исполнении на современной графической аппаратуре.

2.2.2 Пересечение луча с треугольником

В работе [12] предложен наиболее эффективный алгоритм, основанный на использовании *барицентрических координат* (α, β), в которых любую точку M треугольника $T = \{\vec{u}_0 \vec{u}_1 \vec{u}_2\}$ можно представить в следующем виде:

$$M(\alpha, \beta) = (1 - \alpha - \beta) \cdot \vec{u}_0 + \alpha \cdot \vec{u}_1 + \beta \cdot \vec{u}_2,$$

$\alpha \geq 0$, $\beta \geq 0$ и $\alpha + \beta \leq 1$. Данные координаты активно используются на дальнейших стадиях визуализации для интерполяции нормалей, текстурных координат и любых других вершинных атрибутов.

Вычисление точки соударения луча $R(t) = \vec{o} + \vec{d} \cdot t$ с треугольником T эквивалентно решению уравнения $R(t) = M(\alpha, \beta)$, которое можно записать в виде:

$$\vec{o} + \vec{d} \cdot t = (1 - \alpha - \beta) \cdot \vec{u}_0 + \alpha \cdot \vec{u}_1 + \beta \cdot \vec{u}_2.$$

Опуская промежуточные выкладки, приведем псевдокод варианта алгоритма, оптимизированного предварительным вычислением нормали к треугольнику $\vec{n}$:

```
vec3 e1 = u0 - u1;
vec3 e2 = u2 - u0;
vec3 r = u0 - o;
vec3 s = cross ( r, d );
res = vec3 ( dot ( r, n ), dot ( e2, s ), dot ( e1, s ) ) / dot ( d, n );
return res.x > 0 and res.y ≥ 0 and res.z ≥ 0 and res.y + res.z ≤ 1;
```

Предварительный расчет нормалей позволяет исключить одну операцию векторного произведения, и трудоемкость модифицированного алгоритма составляет *42* элементарных арифметических операции и *4* операции сравнения.

2.3 Формат данных и схема вычислений

Для расчета освещенности принята модель Уиттеда. Глубина трассировки является параметром, но по умолчанию для повышения производительности установлена трассировка *одного* отражения и *двух* преломлений луча. Кроме того, для повышения реалистичности изображения визуализируются тени посредством трассировки для каждого источника света одного *теневого* луча.

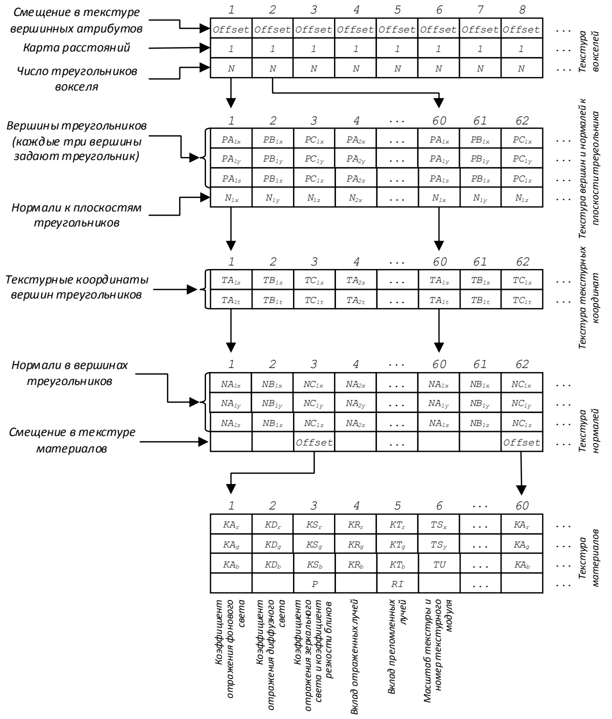

Рисунок 4. Текстуры, используемые для передачи данных

В соответствии с принципами вычислений на графическом процессоре все необходимые для расчетов данные должны быть переданы посредством текстур и глобальных `uniform`-переменных. В задаче трассировки лучей текстуры используются для передачи в шейдер модели компьютерной сцены, представленной в виде ускоряющей структуры. Первичной является *трехмерная текстура вокселей* равномерной сетки. В каждом текселе текстуры вокселей хранится смещение в *текстуре вершин*, *число треугольников*, перекрывающихся с данным вокселем, *радиус* окрестности пустых вокселей и т.д. Глобальные `uniform`-переменные используются для передачи положения и ориентации камеры и источников света.

Для инициирования вычислений следует установить параллельную проекцию и нарисовать прямоугольник, заполняющий всю область видимости. На этапе растеризации данный прямоугольник будет разбит на отдельные фрагменты, соответствующие пикселям в буфере кадра, а каждый фрагмент передан для обработки фрагментному шейдеру.

3. ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ

Для оценки производительности программы трассировки лучей использовалась тестовая сцена “Lexus”.

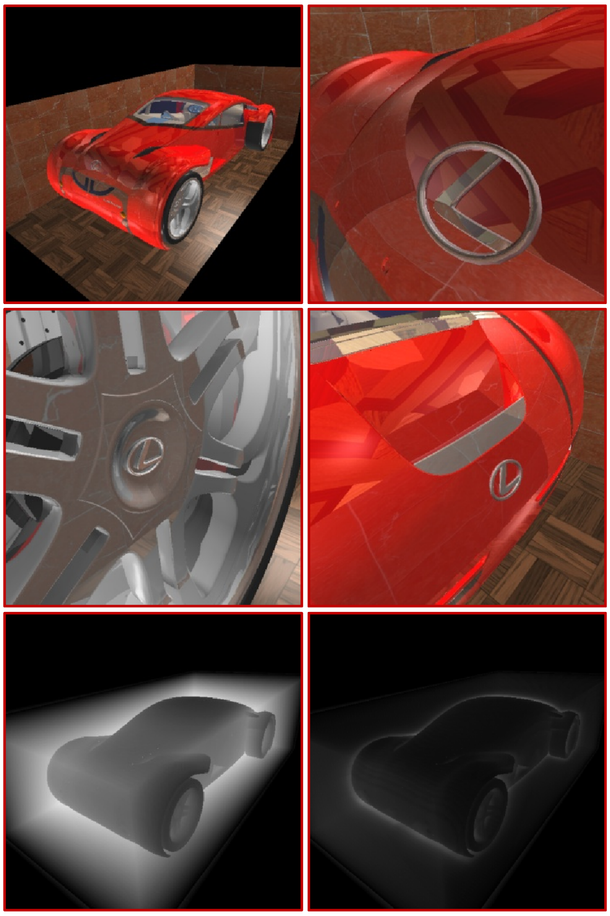

Рисунок 5. Тестовая сцена для визуализации

Модель состоит из ~325 000 треугольников, большинство материалов являются прозрачными и отражающими, установлено *два* точечных источника света, размер окна визуализации составляет 512 × 512 точек. Ниже в таблице даны замеры производительности (*кадр/сек*), соответствующие положению камеры для первого кадра (один из тяжелых случаев).

	Uniform Grid		*Proximity cloud*	
Размерность сетки	*AMD Radeon 4850*	*NVIDIA Quadro 5600*	*AMD Radeon 4850*	*NVIDIA Quadro 5600*
32 × 32 × 32	*12 / 7*	*21 / 11*	*11 / 7*	*20 / 11*
64 × 64 × 64	*22 / 12*	*37 / 20*	*29 / 16*	*51 / 26*
128 × 128 × 128	*25 / 14*	*42 / 22*	*34 / 20*	*68 / 35*

Первое значение соответствует расчету только *прямого* освещения, второе – расчету *прямого* и *вторичного* освещения. Два нижних кадра показывают оттенками серого цвета число обработанных вокселей сетки при отсутствии и наличии информации о близости.

4. ЗАКЛЮЧЕНИЕ

В данной работе построена эффективная реализация метода обратной трассировки лучей для исполнения на графическом процессоре в реальном времени. Данная реализация аккумулирует и развивает существенную часть опыта разработки трассировок реального времени (Real Time Ray Tracing или RTRT). При рядовых аппаратных возможностях реальное время обеспечено для достаточно сложных сцен (из 300K и более треугольников). Эксперимент позволяет сделать вывод о высокой эффективности: ускоряющей структуры на основе регулярной сетки с “облаком близости”; улучшенных алгоритмов пересечения “луч-примитив” без динамических ветвлений; других использованных и оптимизированных в данной работе техник. Подход на основе OpenGL и GLSL позволил задействовать все возможности современных графических процессоров и, одновременно, обеспечил межплатформенность.

5. ЛИТЕРАТУРА

[1] I. Wald, T. Purcell, J. Schmittler, C. Benthin, P. Slusallek. “*Real-time Ray Tracing and its use for Interactive Global Illumination*”/ Eurographics State of the Art Reports (2003).

[2] S. Woop, J. Schmittler, P. Slusallek. “*RPU: A Programmable Ray Processing Unit for Real-time Ray Tracing*”/ Proc. of SIGGRAPH (2005).

[3] T. Purcell. *“Ray Tracing on a Stream Processor*” (2004). (http://graphics.stanford.edu/papers/tpurcell_thesis/tpurcell_thesis.pdf)

[4] F. Karlsson, C. Ljungstedt. “*Ray tracing fully implemented on programmable graphics hardware*” (2004). (http://www.ce.chalmers.se/~uffe/xjobb/GPURT.pdf)

[5] M. Christen. “*Ray Tracing on GPU*” (2005). (http://gpurt.sourceforge.net/DA07_0405_Ray_Tracing_on_GPU-1.0.5.pdf)

[6] A. Adinetz, S. Berezin. “*Implementing Classical Ray Tracing on GPU - a Case Study of GPU Programming*”/ Proceedings of Graphicon (2006).

[7] В. Фролов. Проект “*CUDA Engine of Ray Force*”. (http://ray-tracing.ru/articles163.html)

[8] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. “*Interactive Rendering with Coherent Ray Tracing*”// Computer Graphics Forum, 20(3), 153-164, 2001.

[9] T. Akenine-Moller. “*Fast 3D triangle-box overlap testing*”// Journal of Graphics Tools, 6(1), 29-33, 2001.

[10] J. Amantides, A. Woo. “*A Fast Voxel Traversal Algorithm for Ray Tracing*”. In proceedings of Eurographics’87, 3-10, New York, 1987.

[11] T. Kay, J. Kajiya. “*Ray tracing complex scenes*”. Computer Graphics, 20(4), 269–278, 1986.

[12] T. Akenine-Moller, B. Trumbore. “*Fast, Minimum Storage Ray/Triangle Intersection*”. Journal of Graphics Tools, 2(1), 21-28, 1997.

[13] T. Saito, J. Toriwaki, “New algorithms for n-dimensional Euclidean distance transformation”, Pattern Recognition, 27-11 (1994), 1551-1565.