Ray-Triangle Intersection Test

In this project I was able to implement a Ray-Triangle Intersection through two possible algorithms from the three presented: “naive approach” by Peter Shirley, shown in Chapter 10 of his book, and “Fast, Minimum Storage Ray/Triangle Intersection” by Tomas Möller.

Both of them brought the same results, but (as the name suggests) Möller’s one was faster and had a significant memory saving, thanks to his way of doing the calculations, in which it doesn’t need to compute the plane equation and also doesn’t need to use as many variables as Peter’s one, getting rid of a lot of operations through it.

Bellow we can see the Triangle’s header file:

*triangle.h triangle.cpp

```cpp
#ifndef TRIANGLE_H_
#define TRIANGLE_H_

#include <glm/glm.hpp>
#include "primitive.h"
#include "intersection_record.h"
#include "ray.h"

class Triangle : public Primitive
{

    glm::vec3 vertex[3];

public:

    Triangle( void );

    Triangle( const glm::vec3 &v1, const glm::vec3 &v2, const glm::vec3 &v3 );

    bool intersect( const Ray &ray,
            IntersectionRecord &intersection_record ) const;

};


#endif /* TRIANGLE_H_ */
```

triangle.h

And next, the Triangle's implementation with both functions (put together just to be compared, but only one of them should be kept for the sake of making the renderer work):

triangle.h triangle.cpp

```
#include "triangle.h"

Triangle::Triangle(const glm::vec3 &v1, const glm::vec3 &v2,
        const glm::vec3 &v3) {
    vertex[0] = v1;
    vertex[1] = v2;
    vertex[2] = v3;
}

// A "naive approach" by Peter Shirley.
bool Triangle::intersect(const Ray &ray,
        IntersectionRecord &intersection_record) const {

    float a = vertex[0].x - vertex[1].x;
    float b = vertex[0].y - vertex[1].y;
    float c = vertex[0].z - vertex[1].z;

    float d = vertex[0].x - vertex[2].x;
    float e = vertex[0].y - vertex[2].y;
    float f = vertex[0].z - vertex[2].z;

    float g = ray.direction_.x;
    float h = ray.direction_.y;
    float i = ray.direction_.z;

    float j = vertex[0].x - ray.origin_.x;
    float k = vertex[0].y - ray.origin_.y;
    float l = vertex[0].z - ray.origin_.z;

    float ei_hf = (e * i) - (h * f);
    float gf_di = (g * f) - (d * i);
    float dh_eg = (d * h) - (e * g);

    float ak_jb = (a * k) - (j * b);
    float jc_al = (j * c) - (a * l);
    float bl_kc = (b * l) - (k * c);

    float M = (a * ei_hf) + (b * gf_di) + (c * dh_eg);

    float t = -((f * ak_jb) + (e * jc_al) + (d * bl_kc)) / M;
    if (t < 0.0f)
        return false;

    float gama = ((i * ak_jb) + (h * jc_al) + (g * bl_kc)) / M;
    if (gama < 0 || gama > 1)
        return false;

    float beta = ((j * ei_hf) + (k * gf_di) + (l * dh_eg)) / M;
    if (beta < 0 || beta > 1 - gama)
        return false;

    intersection_record.t_ = t;
    intersection_record.position_ = ray.origin_
            + intersection_record.t_ * ray.direction_;
    intersection_record.normal_ = glm::normalize(glm::cross(vertex[1] - vertex[0],
                vertex[2] - vertex[0]));

    return true;

}
```

```cpp
// “Fast, Minimum Storage Ray/Triangle Intersection” by Tomas Möller.
bool Triangle::intersect(const Ray &ray,
            IntersectionRecord &intersection_record) const {

    float epsilon = 0.000001;

    glm::vec3 edge1, edge2, tvec, pvec, qvec;
    double det, inv_det;
    double t, u, v;

    edge1 = vertex[1] - vertex[0];
    edge2 = vertex[2] - vertex[0];

    pvec = glm::cross(ray.direction_, edge2);
    det = glm::dot(edge1, pvec);

    if (det > -epsilon && det < epsilon)
        return false;

    inv_det = 1.0f / det;

    tvec = ray.origin_ - vertex[0];
    u = glm::dot(tvec, pvec) * inv_det;
    if (u < 0.0f || u > 1.0f)
        return false;

    qvec = glm::cross(tvec, edge1);
    v = glm::dot(ray.direction_, qvec) * inv_det;
    if (v < 0.0f || u + v > 1.0f)
        return false;

    t = glm::dot(edge2, qvec) * inv_det;

    intersection_record.t_ = t;
    intersection_record.position_ = ray.origin_ + intersection_record.t_ * ray.direction_;
    intersection_record.normal_ = glm::normalize(glm::cross(vertex[1] - vertex[0],
               vertex[2] - vertex[0]));

    return true;
}
```

triangle.cpp

To compare both implementations renderer speed, I made a test rendering 10000 random triangles through the use of a for-loop and it became clear how much Möller’s algorithm is faster. The results are as follows:

```
  progress .........................:  99.61%
  progress .........................:  99.80%
  progress .........................: 100.00%
Buffer saving started... finished!
44162.3 ms
```

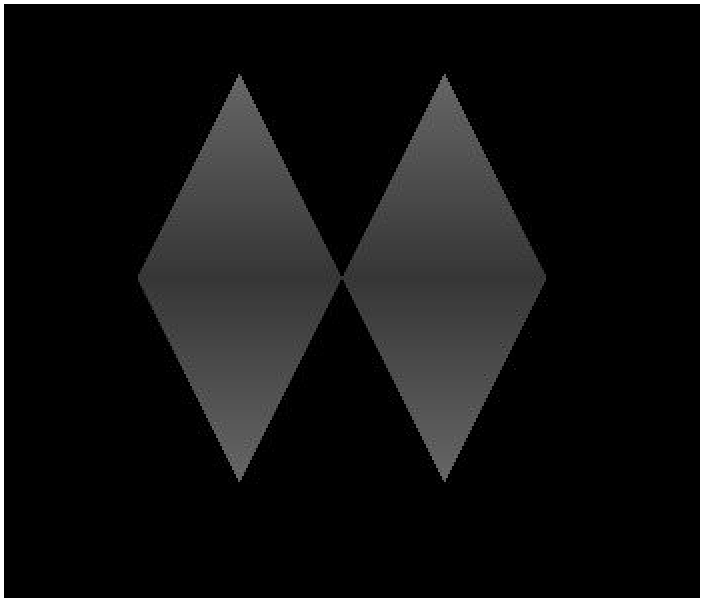
“Naive Approach” rendering time

```
  progress .........................:  99.61%
  progress .........................:  99.80%
  progress .........................: 100.00%
Buffer saving started... finished!
40213 ms
```

“Fast, Minimum Storage Ray/Triangle Intersection” rendering time

As it can be seen, Möller’s algorithm was 4 seconds faster on a scene composed just by simple triangles. We can imagine that the difference would be a lot more significant on a real scene.

To conclude this report, here is a simple image created hardcoded to demonstrate what can be done through both of this methods:



Simple image