МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Ярославский государственный университет им. П.Г. Демидова»

Кафедра математического моделирования

"Допустить к защите"
Зав. кафедрой,
доктор ф.-м.н., профессор
_____ С. А. Кащенко
« ___ » _____ 2015 г.

Выпускная квалификационная работа бакалавра

Параллельные вычисления в задаче оптимизации структуры 3D моделей

Научный руководитель
кандидат физ.-мат. наук, доцент
_____ Д. С. Глызин
« ___ » _____ 2015 г.

_____ А. Ф. Самсонов
« ___ » _____2015 г.

Ярославль 2015 г.

Реферат

Объем 25 с., 8 рис., 3 табл., 9 источников, 4 прил.
Параллельные вычисления, самозатенение, ambient occlusion, GPGPU, OpenCL, оптимизация 3D моделей.

Объектом исследования являются алгоритмы расчета самозатенения 3D моделей с использованием параллельных вычислений.

Цель работы – разработка программного комплекса для расчета самозатенения и удаления внутренней геомтерии.

В процессе работы были найдены способы расчета коэффициента самозатенения, а так же оптимизировано время его расчета.

В результате была созданна программа, способная расчитать самозатенение для моделей любого размера (вплоть до нескольких миллионов полигонов) за приемлимое время.

На данном этапе программа может применятся для подготовки моделей к 3D печати.

Содержание

Введение

3D-принтер — это периферийное устройство, использующее метод послойного создания физического объекта по цифровой 3D-модели. В настоящее время это очень быстроразвивающаяся область, имеющая массу применений и позволяющая изготавливать объекты, которые невозможно создать другими способами. Существует несколько технологий, с помошью которых происходит создание итогового объекта

- Лазерная стереолитография
- Лазерное сплавлени
- Ламинирование
- Застывание материала при охлаждении
- Полимеризация фотополимерного пластика под действием ультрафиолетовой лампы
- Склеивание или спекание порошкообразного материала

Все варианты печати предусматривают использование некоторой трехменой модели для создания программы печати, например на кафедре математического моделирования ЯРГУ, некоторые модели, печатающиеся там, представляют из себя визаулизацию некоторых математических функций. При задании модели функцией получается модель теоретически неограниченной точности, но зачастую в такой модели образуется большое число полигонов, которые находятся внутри модели и не имеют смысла как для печати, так и для рендеринга. В настоящей работе целью была разработка программы, предназначенной для оптимизации таких моделей т.е. отсечения геометрии, которая находится внутри модели, будем называть такую геометрию внутренней.

1. Входные данные

1.1. STL

STL (от англ. stereolithography) – открытый формат файла используемый для хранения 3D моделей и является основным для большинства 3D принтеров, за исключением внутренних форматов, которые рассматривать не будем, т.к. число этих форматов постоянно увеличивается и, фактически, большинство разработчиков аппаратуры 3D печати создают собственные форматы под свои нужды. Информация в STL может хранится как в текстовом, так и в двоичном виде. Вся геометрия задается в виде наборов точек, составляющих треугольник и соответствующую им нормаль, которая не используется при печати. Этот формат был выбран основным форматом для программы по причине его простоты, достаточности для целей не цветной печати и большой распространенности. Формат используется как для загрузки моделей, так и для сохранения.

1.2. Obj

Более продвинутый формат, чем STL, разработанный в Wavefront Technologies. Формат так-же является открытым и очень распространен, поддерживается большинством современных 3D редакторов, но в отличии от STL имеет более расширенные возможности, такие как хранение текстурных координат, нетреугольных полигонов, а так же содержат дополнительные файлы, описывающие материалы, используемые для текстурирования. Используется в проекте для загрузки моделей, выбран по причине простоты и большой распространенности.

2. Расчет досягаемости геометрии

Основным приемом отсечения геометрии является отсечение ее по коэффициенту самозатенения, который вычисляется по следующему алгоритму.

Первым шагом алгоритма будет упрощение данных о геометрии. Упрощение сводит каждый полигон к окружности, имеющей координаты центра, нормаль и площадь, оно позволит сильно упростить расчеты того, насколько одна из поверхностей затеняет другую. Центром окружности является усредненное значение координат всех точек полигона.

$$\sum_{i=0}^{n} x_i/n \tag{1}$$

Нормаль вычисляется по трем любым точкам (a, b, c) по следующей формуле:

$$\begin{aligned} w &= (c-a) \times (b-a) \\ n &= w \cdot 1/\sqrt{\sum_{i=0}^{n} w_i^2} \end{aligned} \tag{2}$$

Площадь треугольника вычислим по формуле Герона.

$$\begin{aligned} p &= \frac{a+b+c}{2} \\ S &= \sqrt{p(p-a)(p-b)(p-c)} \end{aligned} \tag{3}$$

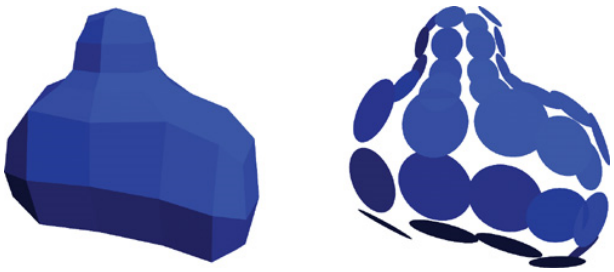

Рис. 1. Перевод полигонов в диски той же площади

Ambient occlusion – полезная техника для затенения объектов, которая используется в современной графике. Благодаря ambient occlusion получаются мягкие тени за счет затемнения поверхностей, которые частично видны с некоторой внешней точки. Эта техника также включает в себя расчет коэффициента досягаемости полигона к окружению, или,

другими словами, число обратное числу пересечений его с другими полигонами в полусфере перед первым.

Будем называть элемент, на который падает тень ресивером, а элемент бросающий тень – эммитером. Для расчетов количества тени, которая бросается эммитером на ресивер используется формула, основанная на телесном угле для ориентированного диска

$$\frac{r \cos \Theta_E \max(1, 4 \cos \Theta_R)}{\sqrt{\frac{A}{\pi} + r^2}} \tag{4}$$

Где – площадь эммитера
E – эммитер
R – ресивер
RE – отрезок, соединяющий центры дисков
r – длина отрезка RE, расстояние между дисками
Θ_R – угол между нормалью ресивера и отрезком ER
Θ_E – угол между нормалью эммитера и отрезком ER

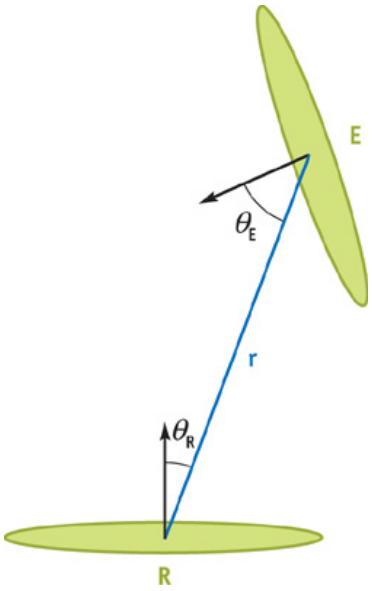

Рис. 2. визуализация расположения эммитера и ресивера и эелементов уравнения 4

Если вычислить значение затенения каждого полигона с каждым другим, то суммарная величина и будет являться коэффициентом самозатенения этого полигона. Полигоны, имеющие относительно низкие значения этого коэффициента более вероятно являются оболочкой, нежели находятся внутри и наоборот. Для полигонов, находящихся внутри модели коэффициент будет близок в единице. Полигоны, имеющие некоторые

средние значения коэффициента могут как находится внутри модели, так и являться оболочкой. Для корректного удаления такой геометрии при отсечении используется вводимая пользователем граница отсечения. В случае слишком агрессивного, или наоборот, отсечения можно отменить сделанные изменения и повторить процедуру с другим коэффициентом. Пересчета самозатенения для этого не требуется, поэтому процедура выполняется в реальном времени.

3. General-purpose graphics processing units

Большинство задач, связанных с графикой и триангулированной геометрией требуют вычислений с большим количеством операций. Современные графические адаптеры устроены таким образом, чтобы выполнять огромное число достаточно простых, не связанных друг с другом, операций одновременно.

Расчеты, в создаваемом, в рамках настоящей работы, приложении, было решено выполнять с помошью GPU.

Существует большое количество технологий для выполнения таких расчетов, рассмотрим некоторые из них.

3.1. Compute shader

Шейдер – программа написанная на особом языке, предназначенная для выполнения процессором видеокарты. Шейдеры бывают нескольких видов:

- Fragment shader
- Vertex shader
- Compute shader
- Tesselation control shade
- Tesselation evaluation shader

Использование геометрических, а в некоторых случаях и фрагментных, шейдеров открыло возможность для расчетов общего назначения с появлением программируемого графического конвеера, взамен статическому. Сами фрагментные шейдеры используются для вычисления цветов пикселей, применения затенения и текстурирования к ним, во время растеризации полигонов видеокартой. В современных GPU может находиться до нескольких тысяч ядер процессора, способных выполнять такие шейдеры. Задача, решаемая в рамках ВКР хорошо распараллеливается, что позволяет получить все преимущества от параллельных вычислений на GPU. Недостатком использования вычислительных шейдеров являются сложности с использованием памяти. Фактически единственным способом как передачи больших массивов данных в программу, так и получения результатов ее работы, является передача их через текстуру, что не всегда является удобным.

3.2. OpenCL

OpenCL – это фреймворк для написания компьютерных программ, основанных на параллельных вычислениях. OpenCL задумывался для работы в гетерогенной вычислительной среде и абстрагирует платформу, на которой производятся вычисления (GPU, CPU) и производителя устройства, предоставляя единый интерфейс.

OpenCL первоначально был разработан в компании Apple Inc. Apple внесла предложения по разработке спецификации в комитет Khronos. 16 июня 2008 года была сформирована рабочая группа Khronos Compute для разработки спецификаций OpenCL. 9 декабря 2008 года организация Khronos Group представила первую версию стандарта.

OpenCL предоставляет гораздо более удобный интерфейс для работы с массивами входных и выходных данных, было принято решение использовать его в качестве основного инструмента вычислений.

4. Реализация расчета самозатенения

4.1. Первичная реализация

Первичная реализация была выполнена с помошью встроеных средств языка программирования C#. Число потоков выполнения определяется числом ядер центрального процессора, затем обрабатываемая модель разделяется на соответствующее число частей и вызываются потоки расчетов, кажому из который достается свой фрагмент модели. Листинг программы расчетов представлен в приложении В. Средняя скорость расчета для модели в 200000 полигонов составила 00:01:40.2732. Учитывая то, что модели аттракторов, печатаемые на кафедре на 3D принтеры имеют до нескольких миллионо полигонов было принято решение оптимизировать скорость вычислений с помошью более продвинутых вычислительных технологий.

4.2. Реализация с помощью compute shader

Следующей реализацией алгоритма была реализация на языке GLSL. Производительность была заметно увеличена за счет числа одновременно выполняемых расчетов и составила всреднем 00:00:30.0788. Производительность была увеличена примерно в 3 раза, однако это все еще было решено недостаточным. Так же очевидными недостатками подхода можно выделить ограничения на обрабатываемые модели, которое обусловлено ограничением видеокарты на одновременно обрабатываемую геометрию (у современных видеокарт порядка 2 миллионов вершин) и зависание работы GUI и ОС в следствие высокой нагрузки на GPU, что переодически приводило ОС к решению о зависании видеодрайвера и его перезагрузке, что прерывало расчеты. Разделение расчетов на небольшие части полностью убирало выигрыш по производительности от использования вычислительных шейдеров.

4.3. OpenCL реализация

Финальной реализацией стала реализация с помошью фреймворка OpenCL. Среднее время выполнения расчетов для модели, состоящей из 200000 полигонов, всреднем, составило 00:00:02.5196, что составило прирост в 40 раз по сравнению с изначальной реализацией. Листинг программы на диалекте языка C99 представлен в приложении Б. Вычисления состоят из двух этапов:

– Предрасчет площадей полигонов (floatSquareDivPi)

– Расчет самозатенения (floatAO)

Предрасчет площадей, непосредственно перед расчетом самозатенения, позволяет значительно увеличить производительность за счет использования дополнительной памяти, но благодаря этому расчет площади производится только число раз, равное числу полигонов, а не квадрату этого числа, как в случае расчета непосредственно во время расчета самозатенения. Примери работы приложения и интерфейс предсавлен в приложениях.

5. Пример работы и интерфейс приложения

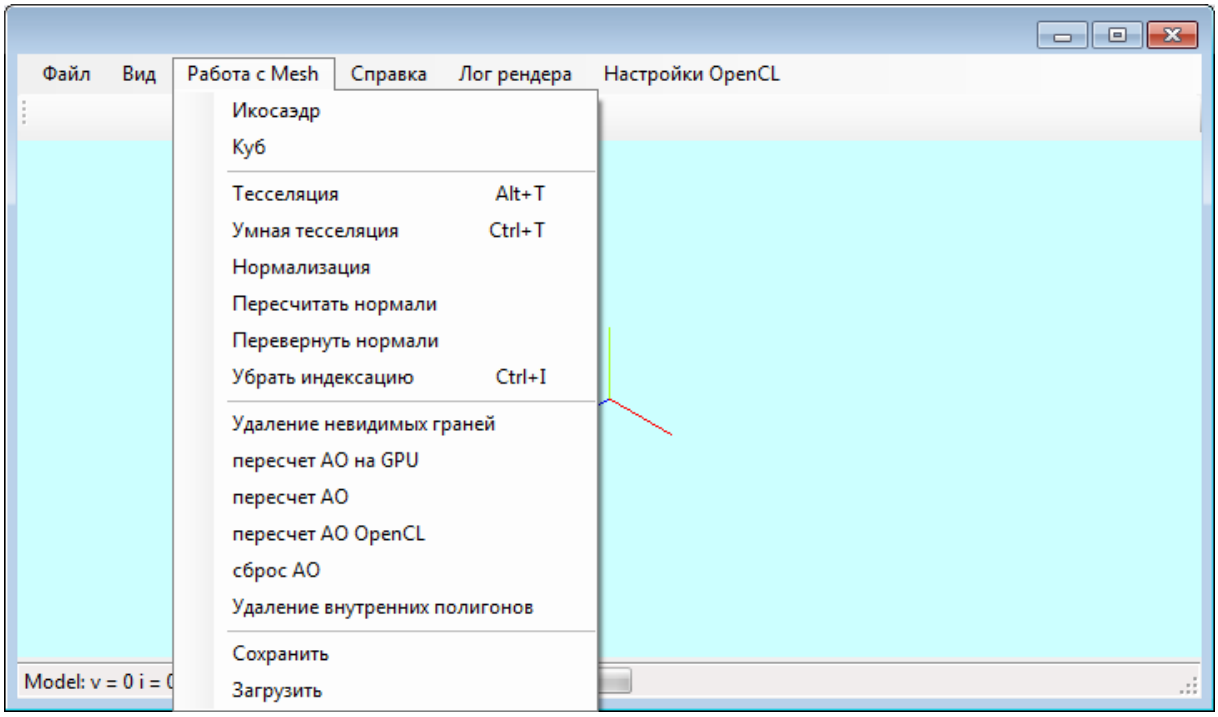

Рис. 3. интерфейс приложения

На рисунке 3 представлен интерфейс приложения. Основной частью которого является интерактивный экран, голубого цвета, на котором отображается модель, обработка которой осуществляется в текущий момент. Рендер модели может производиться в режиме заливки и wireframe. Возможно выбрать способ заливки – затенение и самозатенение. В режим затенением является комбинацией нормального затенения и самозатенения. Режим самозатенения используется для удобного выбора коэффициента отсечения геометрии т.к. позволяет визуально определить что отсекается.

Так же в интерфейсе присутствует меню, состоящее из нескольких пунктов:

– Файл. Позволяет вызвать диалоги сохранения и загрузки файлов.

– Вид. Выбор настроек одображения модели.

– Работа с Mesh. Основные возможности программы. Позволяет тесселировать геометрию несколькими способами для более точного отсечения, убрать индексацию для индексированных моделей, а

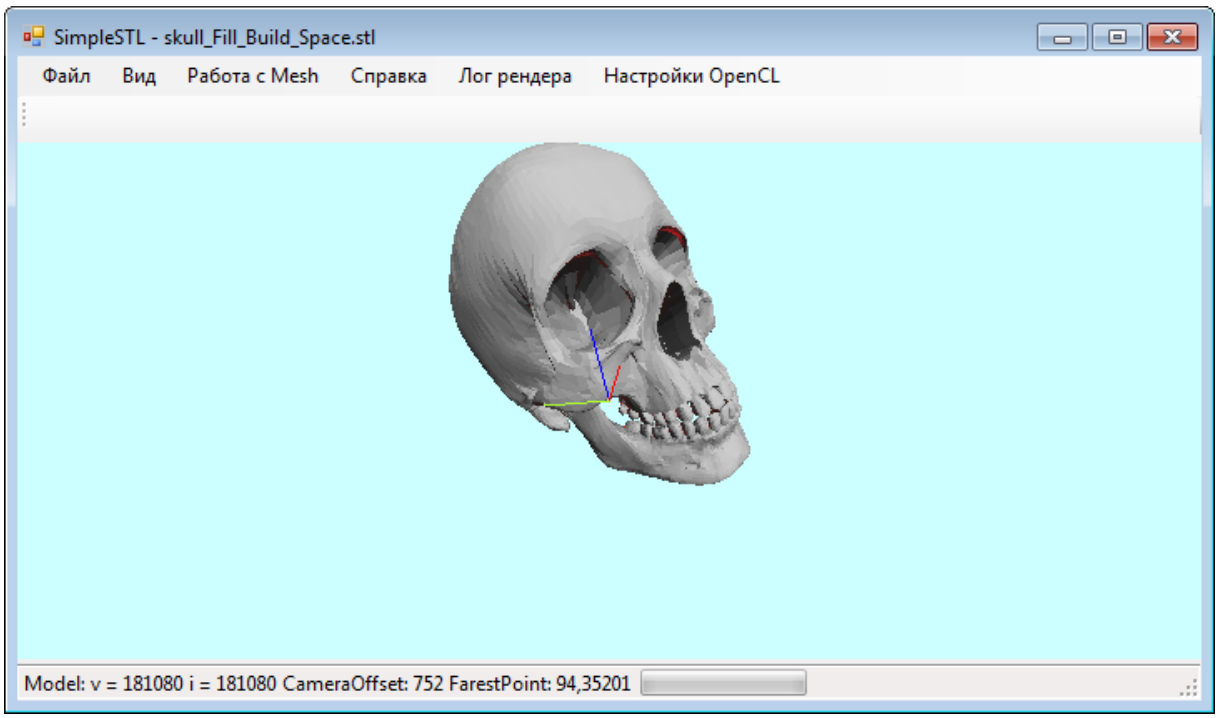

Рис. 4. Модель до расчета самозатенения

так же расчитать коэффициент самозатенения и вызвать диалог отсечения геометрии.

– Справка. Описание работы программы.

– Лог рендера. Выводит информацию о ошибках шейдеров, которые используютя для вывода модели, а так же вычислительных шейдеров и ошибки OpenCL

– Настройки OpenCL. Позволяет выбрать платформу, на которой будет производиться вычисления фреймворком OpenCL.

В нижней строке состояния отображается информация о модели, такая как число вершин и индексов модели, а так же строка прогресса расчета самозатения.

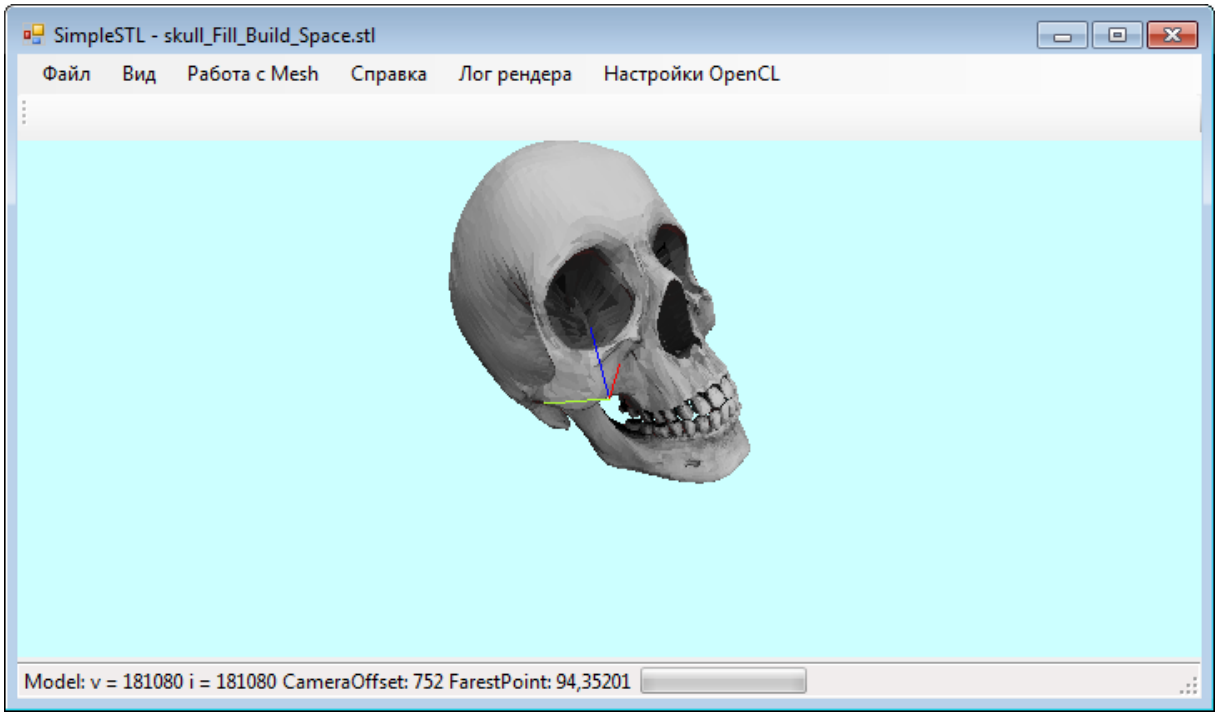

Рис. 5. Модель с расчитанным самозатенением

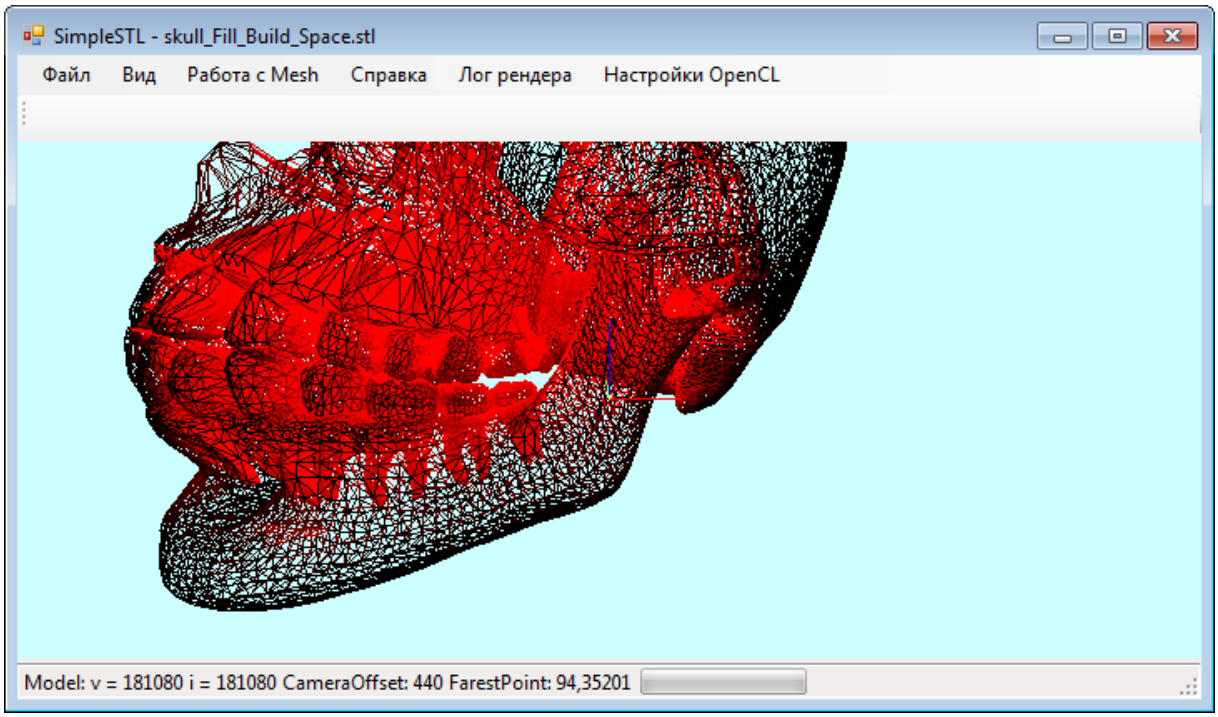

Рис. 6. Самозатенения для геометрии, находящейся внутри модели

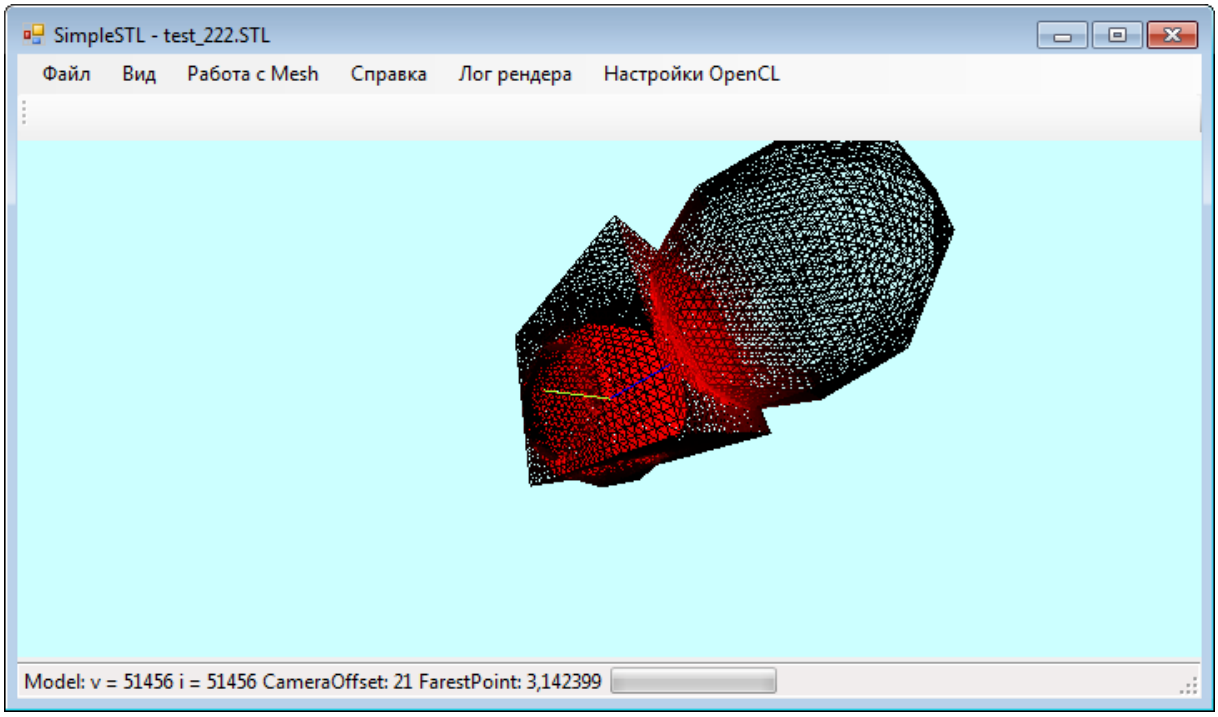

Рис. 7. Модель сфер, пересекающихся с кубом до отсечения геометрии

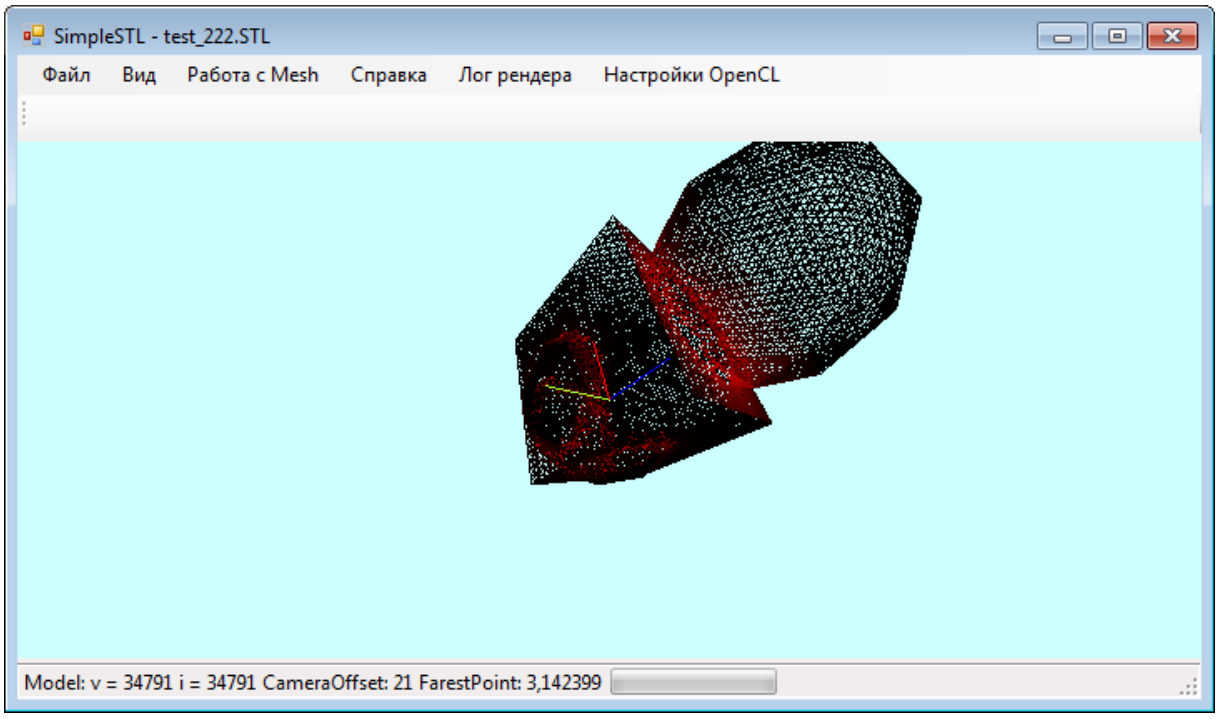

Рис. 8. Модель сфер, пересекающихся с кубом после отсечения геометрии

Заключение

В результате было создано приложение, которое может загружать, сохранять и отображать любые модели в формате STL и Obj, а так же выполнять расчет самозатенения и отсечения геометрии, тем самым упрощая структуру модели непосредственно перед ее печатью или последующим рендером.Одной из целей настоящей работы являлось сравнение выгоды от применения параллельных вычислений, в.т.ч. с применением GPU. Применение GPU позволило ускорить скорость работы алгоритма примерно в 40 раз. Производительность оказалась вполне достаточной для разового расчета, для больших моделей, порядка четырех миллионов полигонов, производится в течении 8 минут на видеокарте бюджетного уровня. Для небольших моделей (несколько сотен тысяч полигонов) расчет производится практически мгновенно.

Список литературы

[1] GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation. Matt Pharr, Randima Fernando. Pearson Addison Wesley Prof, 2005 -Всего страниц: 814

[2] Christensen, P. H. 2002. Note #35: Ambient Occlusion, Image-Based Illumination, and Global Illumination.

[3] Chunhui, M., Jiaoying, S., and Fuli, W. 2004. Rendering with Spherical Radiance Transport Maps. Computer Graphics.

[4] Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. 1988. The Triangle Processor and Normal Vector. Shader: A VLSI System for High Performance Graphics.

[5] Iones, A., Krupkin, A., Sbert, M., and Zhukov, S. 2003. Fast, Realistic Lighting for Video Games.

[6] Kautz, J., Lehtinen, J., and Aila, T. 2004. Hemispherical Rasterization for Self-Shadowing of Dynamic Objects.

[7] Sattler, M., Sarlette, R., Zachmann, G., and Klein, R., 2004. Hardware-accelerated ambient occlusion computation. To appear in the proceedings of International Fall Workshop on Vision, Modeling, and Visualization 2004.

[8] `http://www.cadcamcae.lv/hot/STL_n23_p64.pdf`

[9] `http://www.fabbers.com/tech/STL_Format`

Приложение А

Листинг программы на C# для расчета нормалей

```
public void RecalcNormals()
{
    for (int i = 0; i < Verteces.Count; i+=3) {
        var a = Verteces[i].Position;
        var b = Verteces[i + 1].Position;
        var c = Verteces[i + 2].Position;
        var normal = Vector3.Normalize(
              Vector3.Cross(c - a, b - a));
        var i0 = Verteces[i];
        var i1 = Verteces[i + 1];
        var i2 = Verteces[i + 2];

        i0.Normal = i1.Normal = i2.Normal =
            normal;
        Verteces[i] = i0;
        Verteces[i + 1] = i1;
        Verteces[i + 2] = i2;
    }
    FlipNormals();
}
public void FlipNormals() {
    for (int i = 0; i < Verteces.Count; i++) {
        var vertex = Verteces[i];
        vertex.Normal *= -1;
        Verteces[i] = vertex;
    }}
```

```
public static void AmbientOcclusionRecalc(
          Mesh mesh) {
    var squares = new float[
          mesh.Verteces.Count];
        for (int i = 0; i <
        mesh.Verteces.Count;
        i += 3) {
            var v0 = mesh.Verteces[i];
            var v1 = mesh.Verteces[i + 1];
            var v2 = mesh.Verteces[i + 2];

            var sq = SstlHelper.TriangleSquare(
                            v0, v1, v2);
            squares[i] = squares[i + 1] =
            squares[i + 2] = sq;

        }

    AorResultTotal = mesh.Verteces.Count;
    AorResult = 0;
    int processorCount =
         Environment.ProcessorCount;
    IAsyncResult[] results =
    new IAsyncResult[processorCount];
    var t = SstlHelper.NearestMod(
          (int) (mesh.Verteces.Count/
          (float) processorCount), 3);
    for (int i = 0; i < processorCount; i++) {
        Action<Mesh, float[], int, int>
                        aorT = AorThread;
        results[i] = aorT.BeginInvoke(
          mesh, squares, t*i, t, null, null);
    }

    while (true) {
        bool all = true;
        for (int i = 0; i < results.Length;
               i++) {
            if (!results[i].IsCompleted) {
            all = false;
        }
```

```
        } if (all) {
            break;
        }
        Thread.Sleep(300);
    }}
```

Приложение Б

Листинг программы на C99 для расчета самозатенения

```
__kernel void floatSquareDivPi(
    __global float * vx,
    __global float * vy,
    __global float * vz,
    __global float * nx,
    __global float * ny,
    __global float * nz,
    __global float * a,
    __global float * s)
{
    int i = get_global_id(0)*3;
    float3 vec1 = (float3)(vx[i],
              vy[i], vz[i]);
    float3 vec2 = (float3)(vx[i+1],
               vy[i+1], vz[i+1]);
    float3 vec3 = (float3)(vx[i+2],
              vy[i+2], vz[i+2]);
    float A = distance(vec1, vec2);
    float B = distance(vec2, vec3);
    float C = distance(vec1, vec3);
    float S = (A + B + C) / 2.0f;
    float sq = sqrt(S * (S - A) *
            (S - B) * (S - C)) / M_PI;
    s[i] = s[i+1] = s[i+2] = sq;
}

float ElementShadow(float3 v,
        float rSquared,
        float3 receiverNormal,
        float3 emitterNormal,
        float emitterArea) {
    return (1.0f - rsqrt(emitterArea/
               rSquared + 1.0f)) *
    clamp(dot(emitterNormal, v),
              0.0f, 1.0f) *
    clamp(4.0f * dot(receiverNormal,
              v), 0.0f, 1.0f);
```

```
}

__kernel void floatAO(
                __global float * vx,
                __global float * vy,
                __global float * vz,
            __global float * nx,
                __global float * ny,
                __global float * nz,
            __global float * a,
                __global float * s,
                long from)
{
    int i = from + get_global_id(0)*3;
    float res = 0.0f;
    float3 v = (float3)(0, 0, 0);
    float d2 = 0;
    float value = 0;
    for(int j = 0; j<count;j+=3){
        v = (float3)(vx[j], vy[j], vz[j]) -
                (float3)(vx[i], vy[i], vz[i]);
        d2 = dot(v, v) + 1e-16;
        v *= rsqrt(d2);
        value = ElementShadow(v, d2,
        (float3)(nx[i], ny[i], nz[i]),
        (float3)(nx[j], ny[j], nz[j]),
        s[j]);
        res += value;
    }
 a[i] = a[i+1] = a[i+2] = 1.0f - res;
}
```

Приложение B

Листинг программы на C# для расчета самозатенения

```
public static void
AmbientOcclusionRecalc(Mesh mesh) {
    var squares = new
          float[mesh.Verteces.Count];
    for (int i = 0; i < mesh.Verteces.Count;
           i += 3) {
        var v0 = mesh.Verteces[i];
        var v1 = mesh.Verteces[i + 1];
        var v2 = mesh.Verteces[i + 2];
        var sq = SstlHelper.TriangleSquare(v0,
              v1, v2);
        squares[i] = squares[i + 1] =
        squares[i + 2] = sq;
    }

    AorResultTotal = mesh.Verteces.Count;
    AorResult = 0;
    int processorCount =
         Environment.ProcessorCount;
    IAsyncResult[] results =
         new IAsyncResult[processorCount];
    var t = SstlHelper.NearestMod((int)
          (mesh.Verteces.Count/
          (float) processorCount), 3);
    for (int i = 0; i < processorCount; i++) {
        Action<Mesh, float[], int, int> aorT =
                    AorThread;
        results[i] = aorT.BeginInvoke(mesh,
           squares, t*i, t, null, null);
    }

    while (true) {
        bool all = true;
        for (int i = 0; i < results.Length;
               i++) {
            if (!results[i].IsCompleted) {
                all = false;
```

```
                }
            }
            if (all) {
                break;
            }
            Thread.Sleep(300);
        }
    }
```

Приложение Г

Данные для сравнения скорости работы программы

Таблица 1. Базовая реализация

10000	200000	2000000
00:00:00.591	00:01:43.419	02:00:00.000
00:00:00.576	00:01:44.241	
00:00:00.602	00:01:32.125	
00:00:00.580	00:01:44.158	
00:00:00.611	00:01:37.423	
00:00:00.592	00:01:40.2732	02:00:00.000

Таблица 2. Compute shader

10000	200000	2000000
00:00:01.205	00:00:30.205	-
00:00:01.312	00:00:29.944	-
00:00:01.193	00:00:30.072	-
00:00:01.288	00:00:29.832	-
00:00:01.326	00:00:30.341	-
00:00:01.2548	00:00:30.0788	-

Таблица 3. OpenCL

10000	200000	2000000
00:00:00.362	00:00:02.527	00:02:37.310
00:00:00.334	00:00:02.502	00:02:38.175
00:00:00.388	00:00:02.509	
00:00:00.352	00:00:02.549	
00:00:00.329	00:00:02.511	
00:00:00.353	00:00:02.5196	00:02:37.7525