Софийски университет „Св. Климент Охридски“

Факултет по математика и информатика

Курсов проект

Статично и динамично балансиране при паралелно трасиране на лъчи

Съдържание

1. Увод

Трасирането на лъчи или ray tracing е метод в компютърната графика за генерирането на реални изображения чрез симулиране на физичните закони за движение на светлината в пространството. Изчислява как светлината се движи в сцената и си взаимодейства си с различни повърхности. Най-основните ефекти, които предоставя един рей трейсър са: реалистично осветление, отражения, пречупвания, сенки и глобално осветление.

Такава програма е идеален кандидат за анализ върху потенциалното ускорение чрез разпределена обработка на лъчите. В основата си трасирането на лъчи е тежко изчислително начинание, изискващо значителна изчислителна мощност. Причината за да бъде избрано такова приложение е независимостта на всеки лъч, т.д. всеки лъч се изчислява сам, без да е нужна каквато и да е информация от останалите лъчи. Което улеснява дистрибутирането на системата.

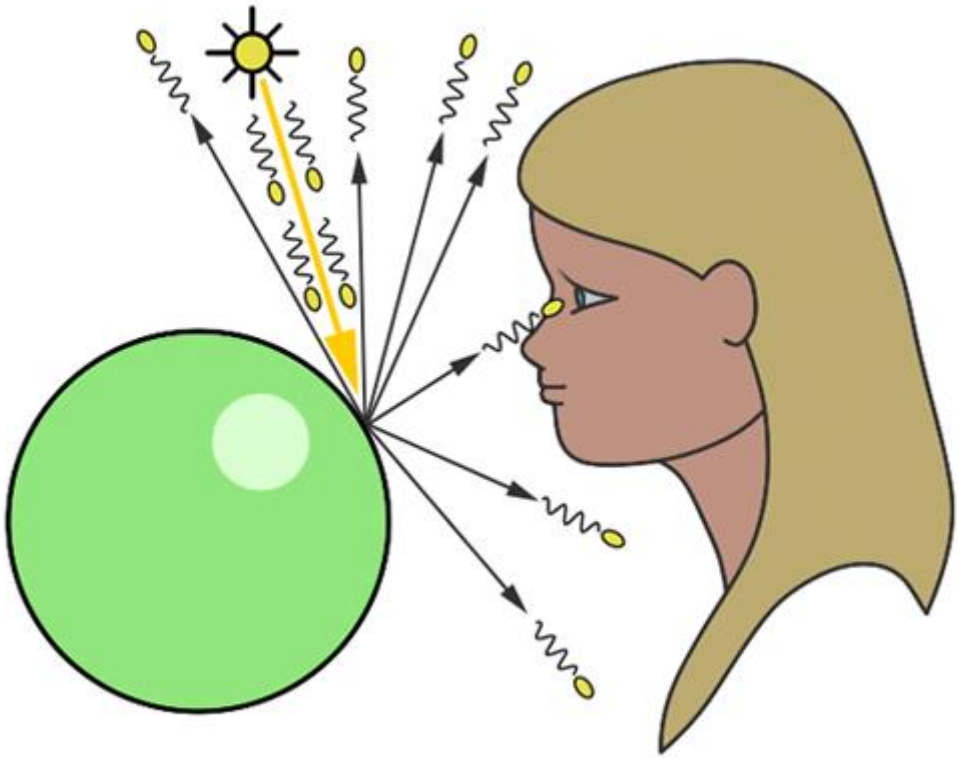
2. Кратко въведение в трасирането на лъчи и симулацията на различните материали

В тази точка се има за цел да запознае читателя накратко как един алгоритъм за трасиране на лъчи работи, без да се навлиза в особени детайли. Ще се фокусира върху основните елементи, както и потенциалните случаи, които могат да забавят приложението и следователно биха могли да се забързат с помощта на дистрибутираното изпълнение.

Трябва да имаме две основни наблюдения: едно - без светлина, не виждаме нищо и две – без да има обекти, с които светлината да си взаимодейства, тя остава невидима за нас. За второто наблюдение, най-лесно би било да си представим космосът. Въпреки множеството от фотони, които пътуват из пространството, ние виждаме единствено и само тъмнина.

Друго важно нещо, на което трябва да обърнем внимание е, че в реалният свят, от всичките лъчи, които биват отразен от даден обект, само малка част ще бъдат прехванати от човешкото око. Ако си представим източник на светлина, който изстрелва точно един фотон и той се удря в обект, който напълно отразява светлината, то ние ще възприемем отражението само тогава, когато фотона се отрази в нашето око.

На **фиг. 1** е предоставено графично изображение за това как работи този процес.

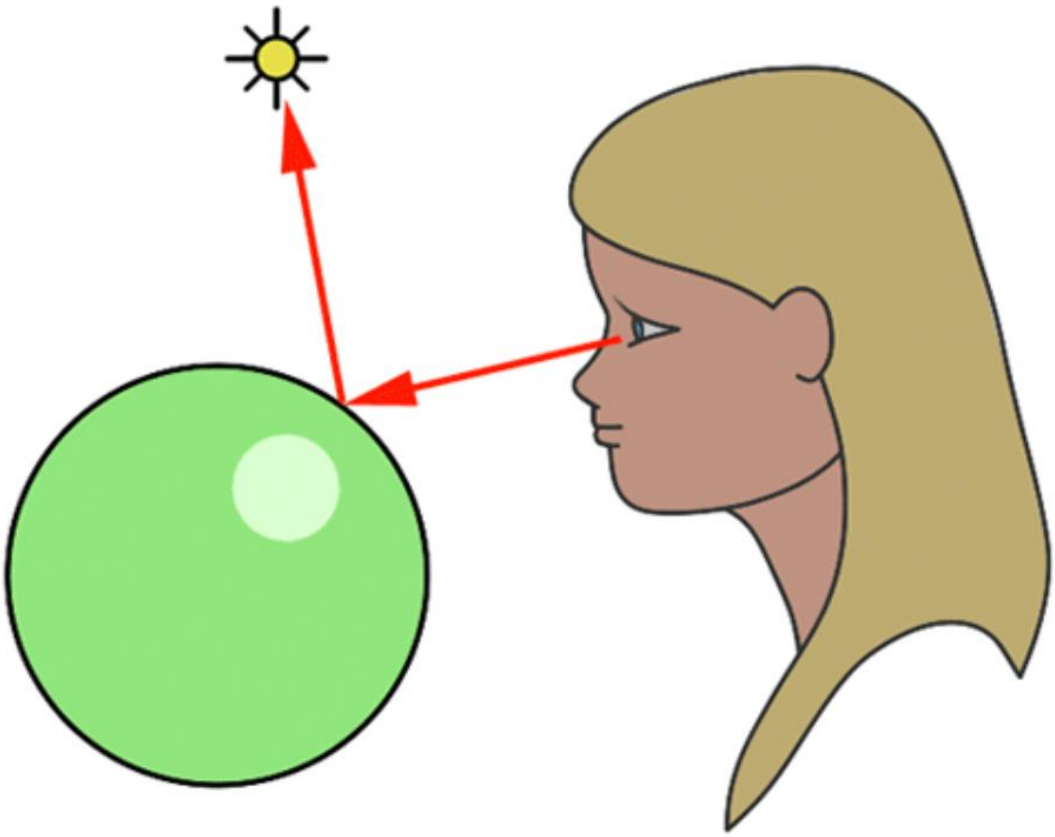


Фиг. 1: Безброй фотони произлизащи от източник на светлина, удрящи зелената сфера, но точно един се улавя от човешкото око.

Така преминаваме към компютърната симулация на този феномен. Нека имаме камера, която играе роля на човешкото око. Ако следваме стриктно физичните закони и от всеки източник на светлина изстрелваме теоретично безброй много лъчи, то много малко от тях реално ще се отразят в камерата и голям брой от изчисляванията ще бъдат напразно. Това се нарича **Forward Tracing**.

Точно затова при компютърното симулиране изстрелваме лъчи от самата камера и изчисляваме как тази лъчи си взаимодействат със самата сцена и елементите в нея. Тази техника се нарича **Backward Tracing**. По този начин генерираме само такива лъчи, които биха се отразили в камерата и няма нужда да правим ненужни сметки. След като се пресече даден елемент на сцената, създаваме нов лъч, който проверява дали елементът е в сянка. Лъча, който се изстрелва от камерата се нарича главен лъч или **Primary Ray**. На фиг. 2 е предоставена графика за това как работи тази техника.

Техниката на изстрелване на лъчи от източника на светлина или от камерата(окото) се нарича **Path Tracing**, като **Ray Tracing** се използва като синоним. На български език, тази техника се нарича трасиране на лъчи. Това е основната техника за генериране на реалистични изображения чрез симулация на оптични феномени като пречупване и разсейване на светлината, или глобално осветление, където светлината се влияе от останалите обекти в сцената.



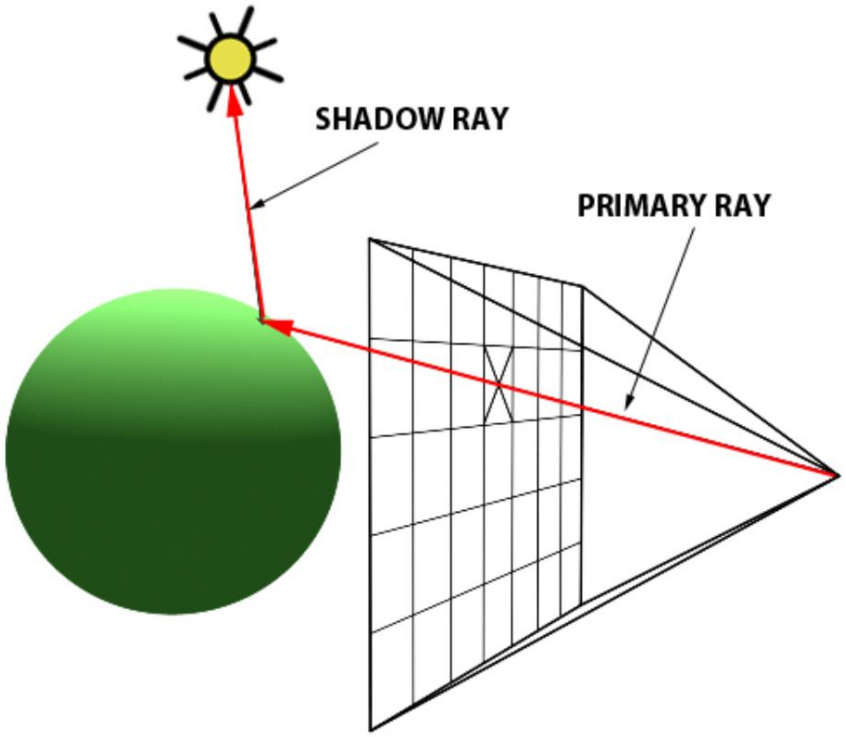
Фиг. 2: Backward Tracing. Трасираме лъч от окото към сцената и след това трасираме нов лъч от сферата към светлинния източник

Сега вече като знаем основите за това как работи светлината в реалният свят и в софтуера, е време да видим какво реално се случва в една програма за да генерира реалистични изображения.

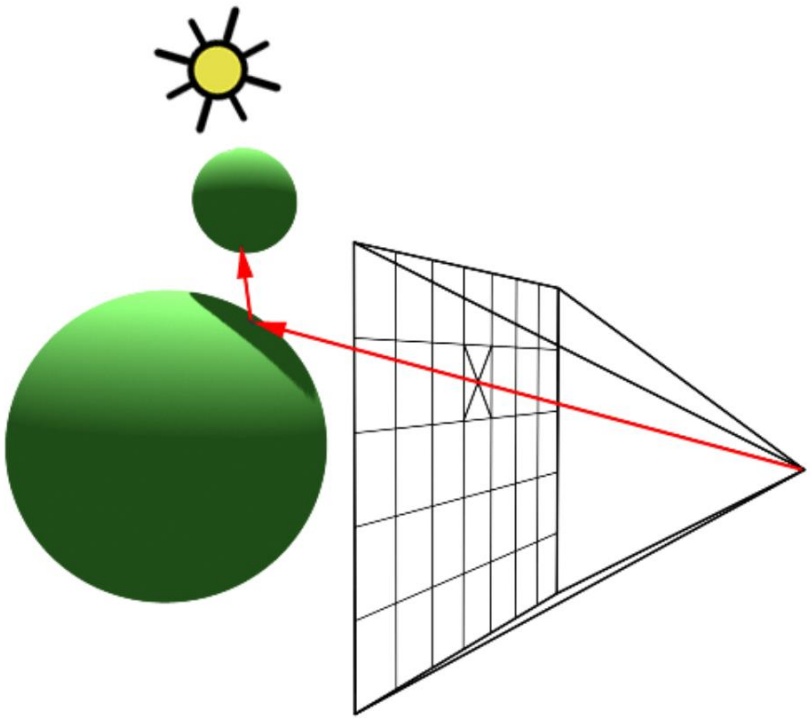
Основата на един алгоритъм за трасиране на лъчи е да генерира изображения пиксел по пиксел. Генерира главен лъч от окото към сцената, като посоката се определя чрез линия от окото към центъра на пиксела. Ако даден елемент бъде пресечен, то пътят на лъча продължава, докато не стигне определен източник на светлина, излезе извън сцената или достигне максимална дълбочина на проследяване. В случай в който се пресекат няколко елемента, се избира този, който е по-близък до камерата(окото). Накрая се генерира и още един лъч от мястото на удара на елемента, който проверява дали тази секция е в сянка като изстрелва лъч към всеки източник на светлина. (**фиг. 3**)

Ако допълнителният лъч, пресече някой друг обект, то тогава тази секция е в сянка и ще бъде оцветена в по тъмен цвят или в черно. (**фиг. 4**)

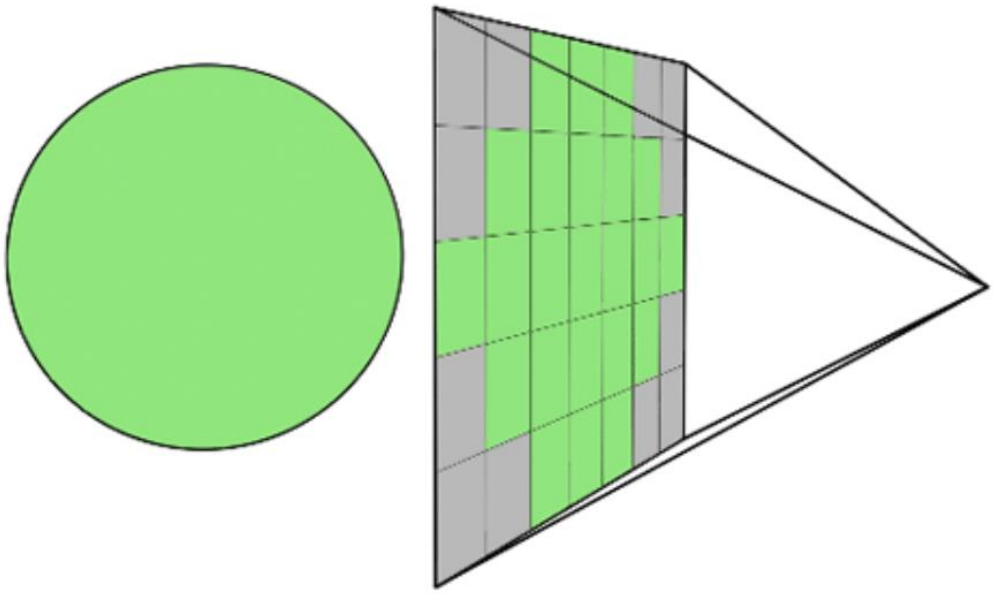
Тази процедура се повтаря толкова пъти, от колкото пиксели се състои нашето изображение. Колкото повече пиксели, толкова повече итерации ще направим, но и ще получим доста по-детайлно изображение. (**фиг. 5**)



Фиг. 3: Главен лъч се изстрелва към центъра на даден пиксел за да провери дали пресича някой обект в сцената. При пресичането на такъв, се генерира сенчест лъч, които определя осветеността на тази секция от обекта.



Фиг. 4: Пресеченият обект е в сянка, така като съществува друг обект, който пречи на източника на светлина да достигне до първия обект.



Фиг. 5: Генерирането на изображение се осъществява чрез пускане на множество от лъчи през всеки един пиксел.

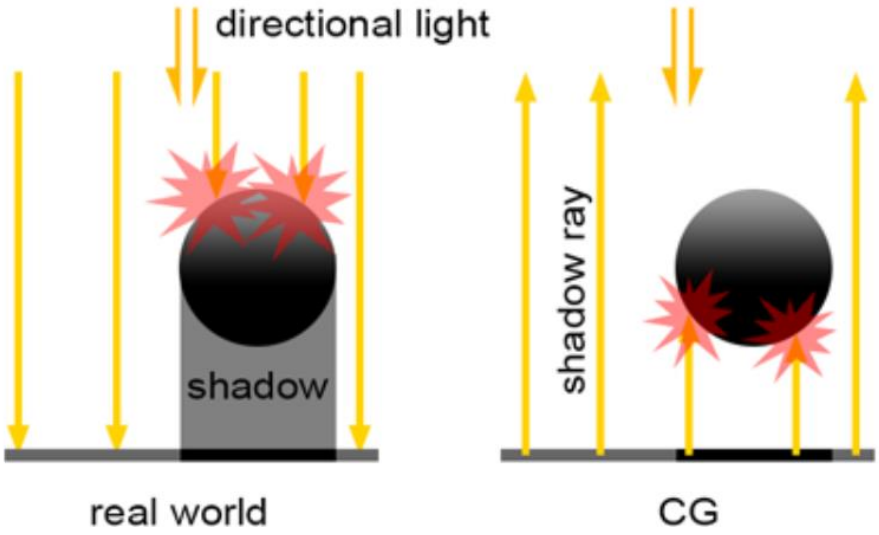
Алгоритъмът за тресиране на лъчи в основата си е с квадратична сложност (**O(N^2)**). Тоест за всеки лъч проверяваме дали пресича някой елемент на сцената, като следим информация за близостта на този обект. Разбира се в зависимост от материала на всеки пресечен обект, нещата се изменят много.

Ако материалът е дифузен, то тогава се изчислява както директното осветяване на елемента, така и неговото индиректно осветяване. Директното осветяване е сравнително просто за изчисляване – проверяваме до каква степен обекта се влияе от всяко осветително тяло на сцената. Индиректното осветяване от друга страна, генерира нови лъчи, които разпръсква из сцената. Това допълнително забавя процеса на генерирането на изображение, така като за всеки пресечен елемент, спрямо неговия материал се правят различни изчисления. (**фиг. 6** и **фиг. 7**).

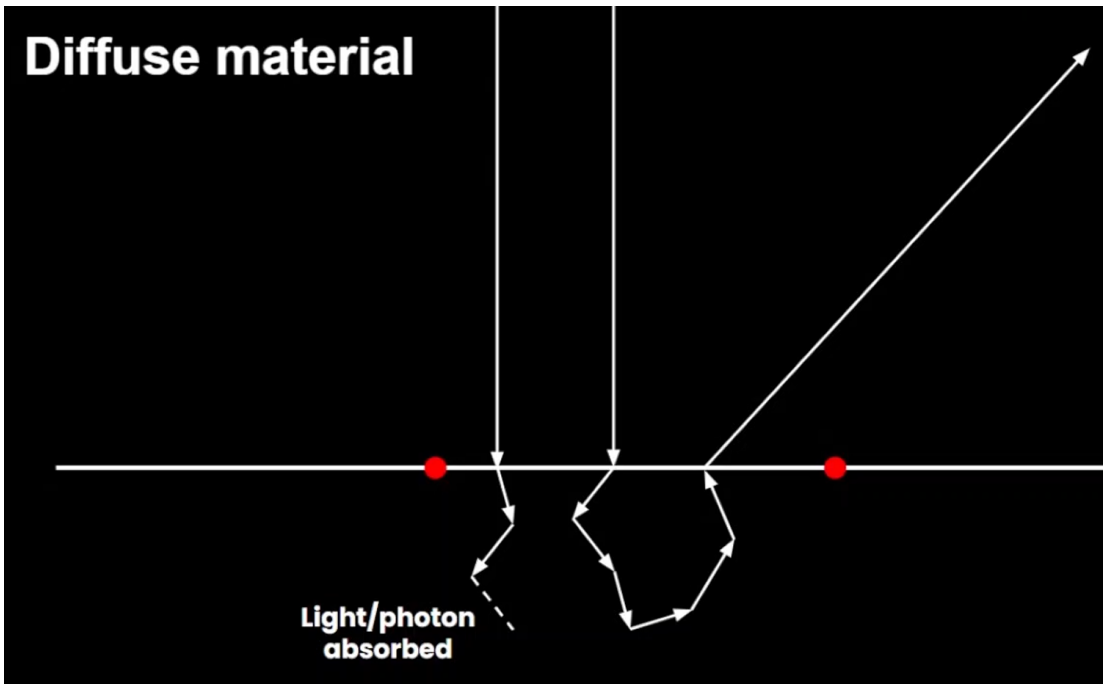
Ако материалът е рефлективен (примерно огледало), то тогава лъчът се отразява и продължава своя път в отразената посока. Лъча продължава своя път докато не достигне елемент с материал, който го терминира, източник на светлина или максимален брой на дълбочина на лъчите. (**фиг. 8** и **фиг. 9**) Разбира се, с всяко отразяване, лъча губи своята цветова интензивност.

Последният материал, който ще разгледаме е материал, който пречупва светлината. Когато светлината преминава от един прозрачен материал към друг, тя променя своята посока. Лъча се „огъва“, както прави светлината когато премине от въздух към вода и обратно. Новата посока се определя от индекса не

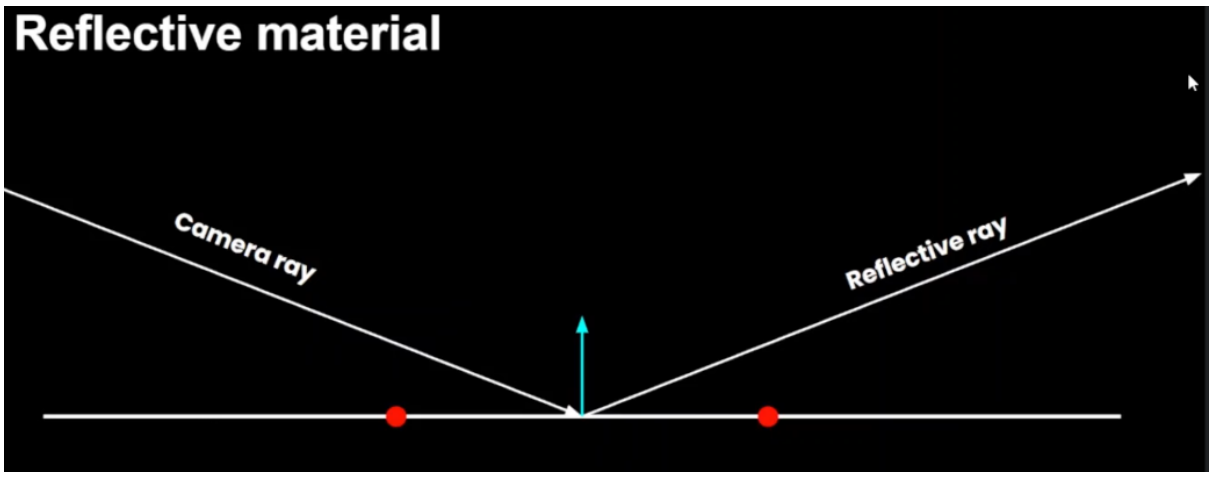
пречупване. (**фиг. 10** и **фиг. 11**) Когато светлината попадне в новия материал, тя се абсорбира и отразява. След абсорбирането, светлината продължава своя път в новия материал докато не излезе от него. Няма да навлизаме в детайли за това как работи този процес. За целите на този документ е нужно да се знае, че е много сложен процес и изисква генерирането на множество лъчи за да се генерира реалистично изображение. Съществува и така нареченото „тотално вътрешно отразяване“, където светлината(или лъча) се отразява в самия обект докато не излезе от него или не достигне определена дълбочина на изминатия път.



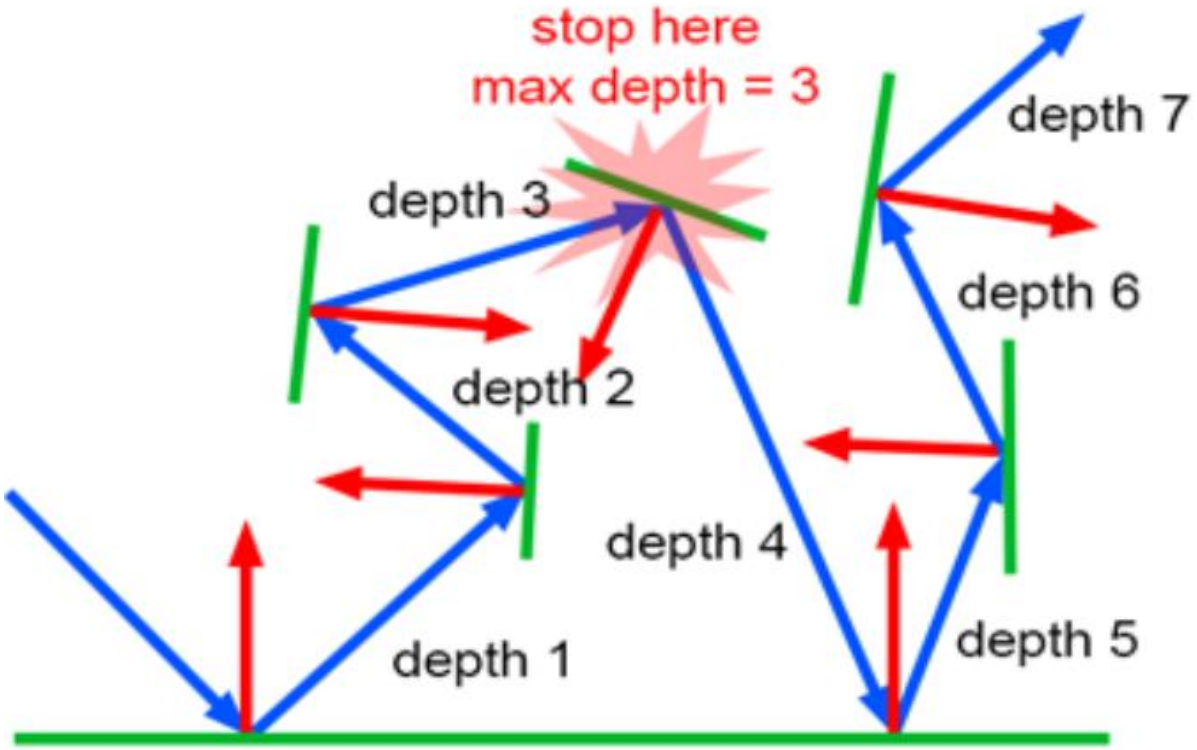
Фиг. 6: Сравнение между изчислението на директното осветление за един дифузен обект в реалният свят и в компютърната графика



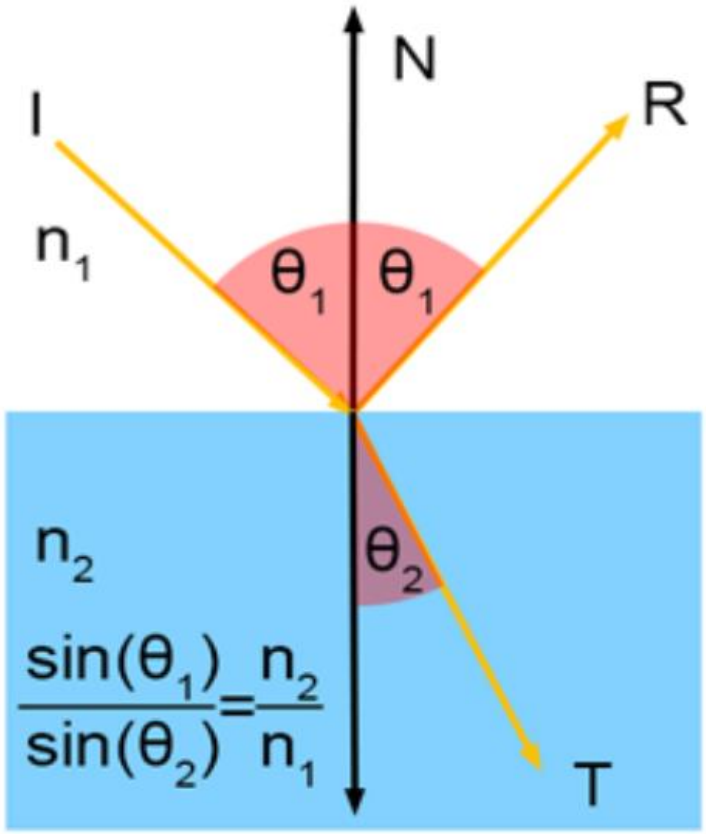
Фиг. 7: Показва как светлината се поглъща и отразява обратно в сцената при дифузните обекти. По този начин се изчислява индиректното осветяване.



Фиг. 8: Показва как един лъч се отразява в един обект с рефлективен материал



Фиг. 9: Отражението е рекурсивен процес



Фиг. 10: Графично представяне на това как един лъч се пречупва и отразява, когато преминава от един прозрачен материал, към друг.

Фиг. 11: Ефектът на пречупване на светлината

С тази информация вече имаме всичко нужно, като информация за това как работи един софтуер за генериране на изображения с помощта на **Ray Tracing** и защо един такъв софтуер е добър кандидат за анализ върху дистрибутираното му изпълнение. Важно е да се отбележи, че изчисленията за всеки пиксел са независими един от друг. Това означава, че всеки пиксел или група от пиксели биха могли да бъдат разделени един от друг и да бъдат изчислявани на независими едни от други ядра.

Това е причината комерсиалните **Ray Tracer**-и да работят основно върху графични ускорители с много на брой ядра, независими един от друг. За целите на документа, нашата имплементация ще работи само върху процесора и няма да се възползва от техники като **SIMD**(**Single Instruction/Multiple Data**). Обектите ще бъдат представени от множество от триъгълници, както и от различни материали, описани по-горе.

3. Цел на анализа

Целта на анализа е да се разгледа по какви начини бихме могли да забързаме една програма занимаваща се с трасирането на лъчи. Да се разгледа как различните начини на балансиране променят скоростта на генерирането на изображения, както и различният брой ядра на процесора.

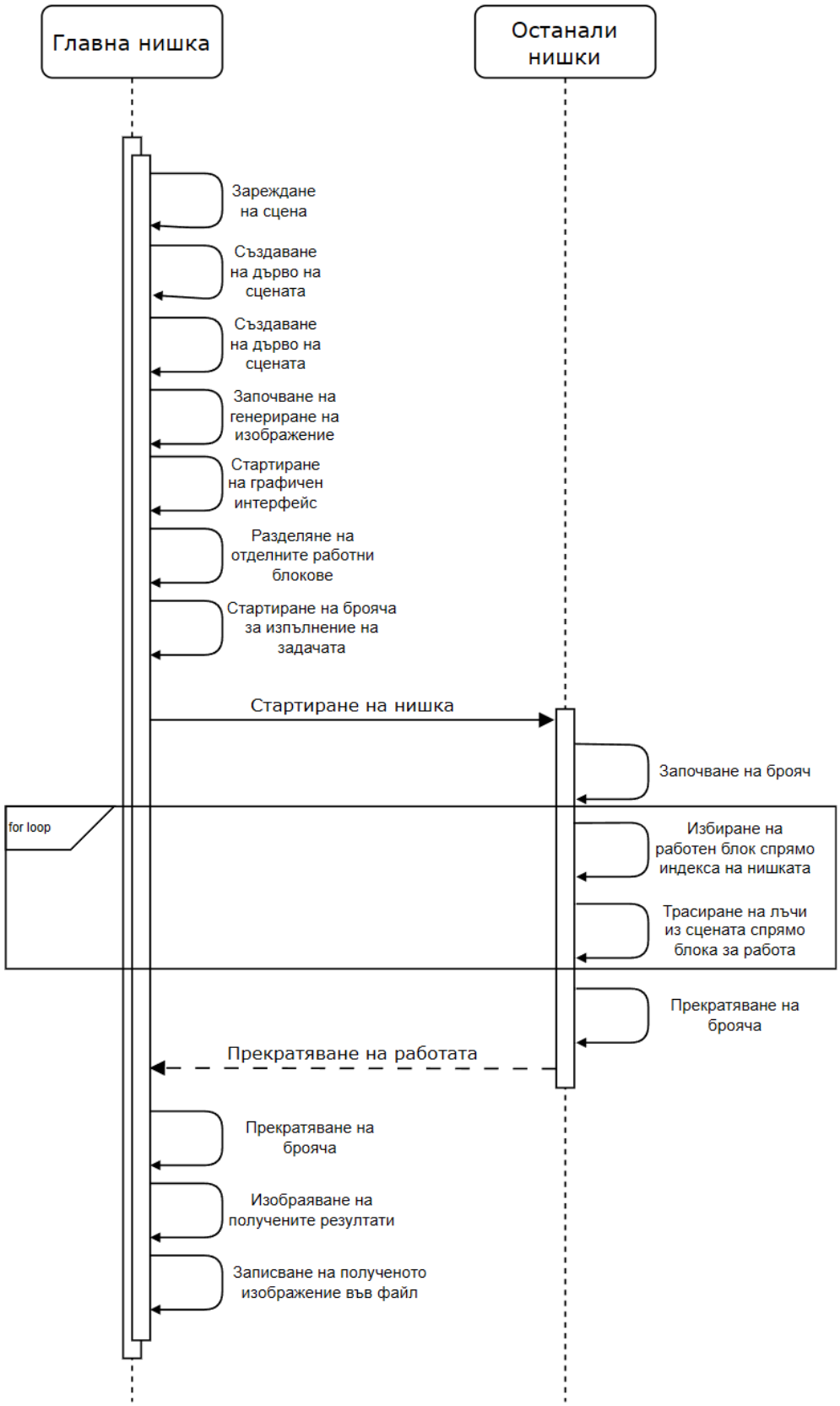
4. Проектиране

4.1. Функционално проектиране

В рамките на проекта са реализирани два вида балансиране: статично циклично и динамично централизирано, с оглед на това да се направи сравнение между двете. Паралелизмът се постига чрез декомпозиция по данни и асинхронен алгоритъм. Стремежът пада основно върху това да се построи архитектура от тип **Single Program Multiple Data**, където всеки процес ще изпълнява едни и същи изчисления за сцената, но върху различни данни(материали). Архитектурата ще напомня на **Master-Slave**, така като първата нишка ще отговаря за разделянето на секторите от сцената измежду различните нишки, както и тяхното стартиране. Основният фокус на този документ е да проверим как различните начини на балансиране променят скоростта на изпълнение на задачата.

4.1.1 Реализация със статично циклично балансиране

При статичното циклично балансиране, матрицата на предвиденото за генериране приложение се разделя на предварителен брой клъстери. Тези клъстери след това се възлагат на всяка налична нишка по предварително определен начин. Това позволява по-справедливо разпределение на работата. От друга страна този метод може да се окаже проблемен ако даден нишка получи прекалено тежки задачи при първоначалното разпределяне, а останалите нишки да имат лесни обекти и материали за трасиране.

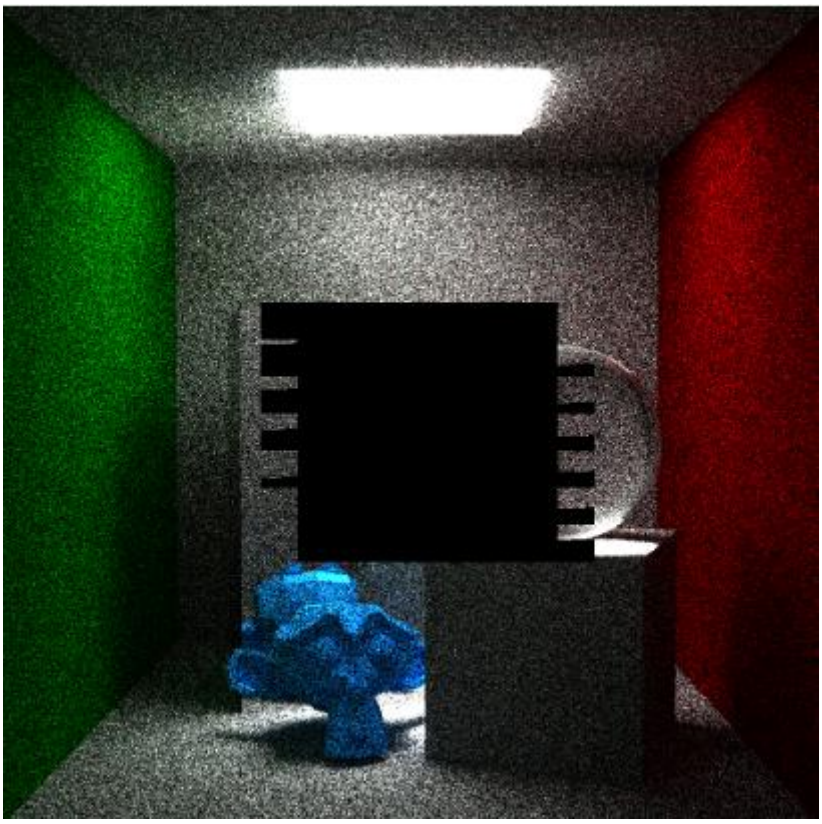


Фиг. 12: Диаграма на последователността при статичното циклично балансиране.

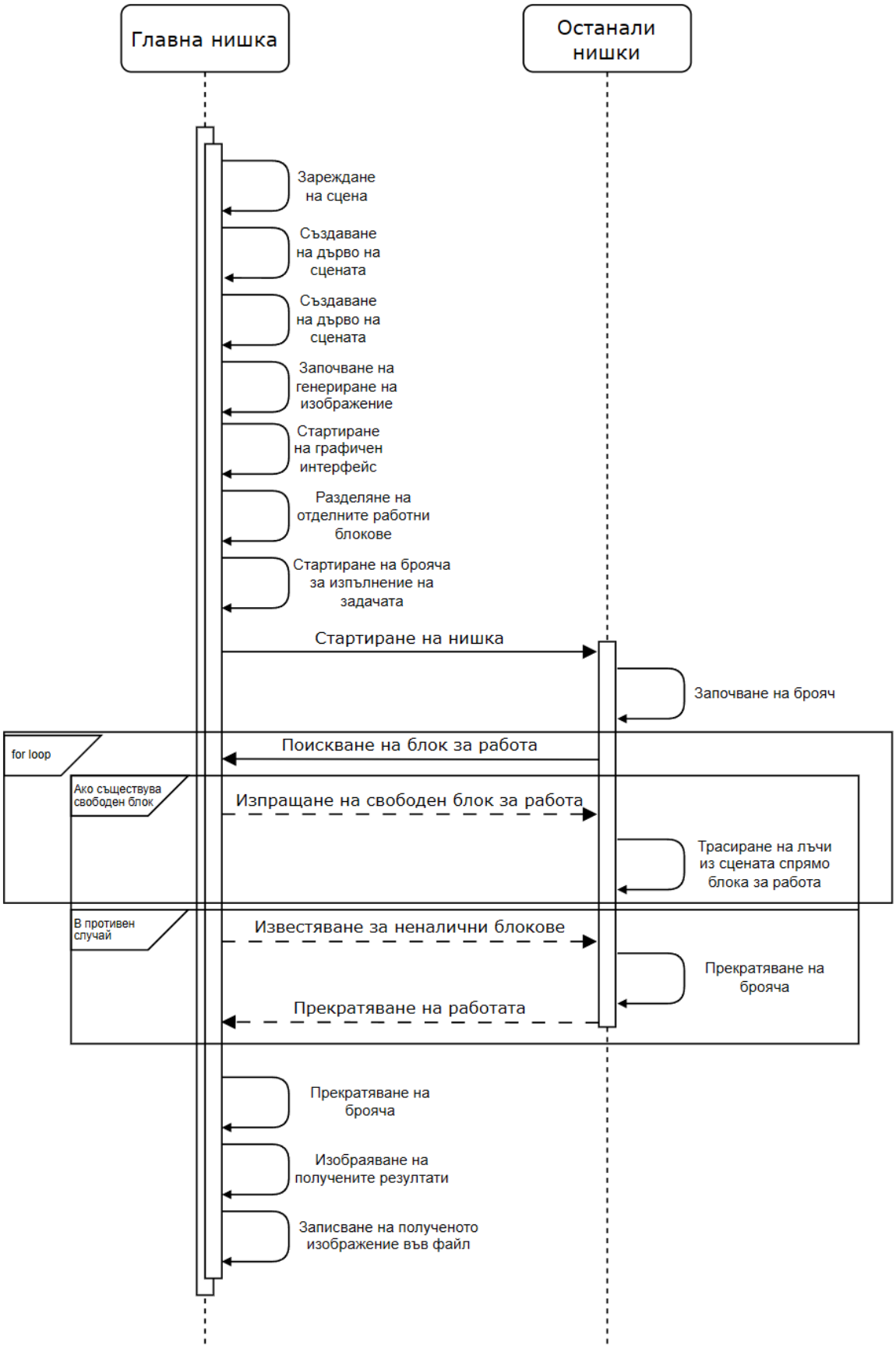
4.2. Реализация със динамично централизирано балансиране

При динамичното централизирано балансиране, матрицата отново се разделя на предварителен брой клъстери, но този път всяка нишка получава нов клъстер тогава и само тогава, когато е готова с вече изпълняваната от нея задача. С други думи се използва **First Come, First Serve** моделът за разпределяне на задачите измежду нишките. Този подход гарантира, че през повечето време, процесите са заети. От друга страна, това налага интензивна комуникация между отделните процеси. Така, като трасирането на лъчи е доста сложен процес, изискващ голяма изчислителна мощ, се очаква този подход да бъде по-добър от статичния в случай на големи и обемни сцени. И статичният да бъде по-добър при прости сцени без много елементи.

В текущата система, динамичното балансиране ще се осъществява в спираловидно движение. Това означава, че първата задача ще започне от горния ляв ъгъл и раздаването ще продължи надясно по ръбовете на изображението.



Фиг. 13: Спираловидно рендериране на сцена



Фиг. 14: Диаграма на последователността при динамично централизирано балансиране

4.2. Технологично проектиране

Системата е проектирана не езикът C++, като са използвани няколко външни библиотеки, които не са свързани с дистрибутивността ѝ. За да се постигне паралелизъм се използва вградения в езика **std::thread**, както и **std::mutex** за синхронизация измежду нишките при динамичното балансиране.

4.2.1. Архитектура

Програмата започва, като прочита сцена с формат .crtscene, от файловата директория на потребителя. Тази сцена се десериализира на отделни компоненти, като камера, светлинни източници, материали, триъгълници и така нататък. След това се създава **BVH** дърво за всеки обект на сцената. Може да се стартира и допълнителна нишка, която да изобразява **UI** с помощта на **OpenGL**.

След като се зареди сцената успешно, се пристъпва към генериране на изображението. То се извършва в член-функцията **void Renderer::render** (**фиг.15**). Първата стъпка е да се създаде изображение, върху което да се чертаят цветовете от сцената. След това създаваме нужните блокове по които ще работят отделните нишки. Всеки блок представлява квадратна матрица, която съответства на определен сектор от квадрати на изображението върху което се рендерира сцената. Накрая, тази функция създава нужните нишки и изчаква за тяхното изпълнение.

Всяка нишка изпълнява един и същ код. Една нишка се представя като отделен обект от тип **Worker**, който играе ролята на функтор.(**фиг. 16**) Първо се генерират се определен брой лъчи, който да се изстрелят из сцената. След това влизаме в цикъл, който се изпълнява, докато не се изчерпат нужните задачи предназначени за нишката(статично балансиране) или не се изчерпат свободните блокове за работа(динамично балансиране). След като дадена нишка получи работен блок по който да работи, се започва трасирането на всеки лъч на база данните в блока.

Раздаването на работен блок може да се случва по два начина: (1) чрез статично балансиране(**фиг. 17**) или (2) чрез динамично балансиране(**фиг. 18**).

При статичното балансиране, операциите са сравнително тривиални. Всяка нишка достъпва индекса на този блок, на който се намира тя. Ако разполагаме с 16 работещи нишки и текущата нишка е с индекс 3(ако първият индекс е 0), то тогава текущата нишка ще изпълнява всяка 4та задача през 16 блока. Тоест на индекс 3, 19, 35 и т.н.

При динамичното балансиране е нужно да използваме метод, чрез който да блокираме достъпа до процеса на раздаване на нишките. Блокирането го осъществяваме чрез помощта на **std::mutex**. Така се избягват проблеми с **Race Conditions** и ще се забрани на няколко нишки, да модифицират **s_Buckets** едновременно. **fetchNextBucket()** сама по себе си подпомага спираловидното раздаване на работните блокове.

4.2.2. Програмен код

```
void Renderer::render(const Scene& scene, unsigned sampleCount, unsigned threadCount)
{
    s_ShouldStop = false;
    const SceneSettings& settings = scene.settings();
    m_Image = Image(settings.width, settings.height);

    const unsigned bucketSize = scene.settings().bucketSize;
    Worker::createBuckets(bucketSize, scene);

    threadCount = threadCount == 0 ? std::thread::hardware_concurrency() : threadCount;

    std::vector<std::thread> threads;
    threads.reserve(threadCount);

    std::cout << "Rendering..." << std::endl;

    {
        Timer t;
        for (size_t i = 0; i < threadCount; i++)
            threads.emplace_back(Worker{}, std::ref(scene), std::ref(m_Image), sampleCount);

        for (std::thread& thread : threads)
            thread.join();

        std::cout << "Render completed!\n";
    }

    std::cout << "\n" << std::endl;
}
```

Фиг. 15: Програмният код, който започва генерирането на изображение

```
void Renderer::Worker::operator()(const Scene& scene, Image& image, unsigned sampleCount)
{
    const Camera& camera = scene.camera();

    std::vector<Ray> rays;
    rays.reserve(sampleCount);

    assert(s_Buckets.size() > 0);

    size_t iterations = 0;
    while (assignBucket())
    {
        Timer t(&m_Id, &iterations);

        if (s_ShouldStop)
            return;

        if (m_Id == 0)
            std::cout << "Progress: " << (int)((s_CompletedBuckets) / (float)s_BucketCount * 100) << "%\r";

        for (unsigned row = m_BlockY; row < (m_BlockY + s_BucketSize); row++)
        {
            for (unsigned col = m_BlockX; col < (m_BlockX + s_BucketSize); col++)
            {
                if (row >= s_ImageHeight || col >= s_ImageWidth)
                    break;

                Color color;
                camera.generateRays(col, row, rays, sampleCount);

                for (unsigned i = 0; i < sampleCount; i++)
                    color += traceRay(rays[i], scene);

                rays.clear();
                color *= 1.0f / sampleCount;
                color = clamp(color, 0.0f, 1.0f);
                image.setPixel(col, row, color);
            }
        }

        iterations++;
        s_CompletedBuckets++;
    }
}
```

Фиг. 16: Програмният код, който изпълнява една отделна нишка

4.2.2.1. Статично балансиране

```
bool Renderer::Worker::assignBucket(unsigned threadCount)
{
    unsigned currBlock = m_CompletedBlocks++;
    unsigned height = s_Buckets.size();
    unsigned width = s_Buckets[0].size();

    unsigned blockIndex = m_Id + threadCount * currBlock;

    if (blockIndex >= width * height)
        return false;

    unsigned y = blockIndex / width;
    unsigned x = blockIndex % width;

    if (!s_Buckets[y][x])
    {
        m_BlockX = x * s_BucketSize;
        m_BlockY = y * s_BucketSize;
        s_Buckets[y][x] = true;
    }

    return true;
}
```

Фиг. 17: Програмният код, който разпределя различните блокове статично на нишките

4.2.2.2. Динамично балансиране

```
bool Renderer::Worker::assignBucket()
{
    bool hasAssigned = false;

    std::lock_guard<std::mutex> lock(s_Mutex);
    {
        int x, y;
        while (!hasAssigned)
        {
            fecthNextBucket(x, y);
            if (y < 0 || y >= s_BucketHCap || x < 0 || x >= s_BucketWCap)
                break;

            if (!s_Buckets[y][x])
            {
                m_BlockX = x * s_BucketSize;
                m_BlockY = y * s_BucketSize;
                s_Buckets[y][x] = true;

                hasAssigned = true;
            }
        }

        return hasAssigned;
    }
}
```

Фиг. 18: Програмният код, който разпределя различните блокове динамично на нишките

5. Представяне на сцените, които ще анализираме

Избрани са общо 3 сцени, които да бъдат използвани за анализът. Сцените варират от сравнително прости до много сложни. В общия случай сцените, които биват подавани на един **Ray Tracer** са небалансирани и някои сектори от тях се изчисляват за повече време от други. В края на документа за изобразени сцените.

Избраните сцени варират откъм големина, сложност и материали. Тяхната цел е да представят възможно най-точни резултати на фона на техния брой. Естествено не е възможно да преминем и анализираме всяка една сцена.

Първата сцена е сравнително проста. Тя разполага със стая в която се намира една стъклена сфера. По този начин анализираме малки сцени, които имат в себе си сложни материали.

Втората сцена съдържа матов дракон, седящ върху огледално покритие. Така можем да анализираме средни сцени с прости материали.

Последната сцена е най-сложна от всички. Тя има най-много триъгълници, съдържа обекти от всеки наличен за приложението материал и изчислява глобалното осветление на всички елементи. Тя е най-сложна и най-интересна за анализ.

6. Сравнителна таблица на предвидените тестове

n – Номер на сцената
t – Броят триъгълници в сцената
s – Броят лъчи изстреляни на пиксел (sample count)
d – Максималната дълбочина, която могат да достигнат лъчите
D – Максималната дълбочина, която могат да достигнат лъчите при индиректно изчисляване на светлината
wh – Широчина и височина на изображението
g – Дали има глобално осветяване (global Illumination)
m – Материали
b – Кратко описание на сцената

n	*t*	*s*	*d*	*D*	*wh*	*g*	*m*	*b*
1	970	50	10	5	1080x1080	не	дифузен, пречупващ	Стъклена сфера
2	4013	50	5	5	1920x1080	не	дифузен, отразяващ	Матов дракон
3	9825	100	10	5	512x512	да	дифузен, пречупващ, отразяващ	Маймуна и геом. обекти

7. Тестови среди

Изследванията за проведени на две тестови среди:

Личен лаптоп:
Model: Lenovo Yoga Pro 7 14APH8
CPU: AMD Ryzen 7 7840HS
CPU(s): 8 / 16 логически
Base Clock: 3.8 GHz
Max. Boost Clock: 5.1 GHz
L1 Cache: 64 KB (на ядро)
L2 Cache: 1 MB (на ядро)
L3 Cache: 16 MB (споделена)
Instruction Set: x86-64
RAM: 16GB

Личен лаптоп
Model: Lenovo ThinkPad P1 G5
CPU: Intel Core i9-12900H
CPU(s): 6 performance / 8 efficiency (20 логичекси)
Base Clock: 2.5 GHz / 1.8 GHz
Max. Boost Clock: 5.0 GHz / 3.8 GHz
L1 Cache: 80 KB (на ядро)
L2 Cache: 1.25 MB (на ядро)
L3 Cache: 24 MB (споделена)
E-Core L1: 96 KB (на ядро)
E-Core L2: 2MB (на модул)
Instruction Set: x86-64
RAM: 64GB

8. Резултати

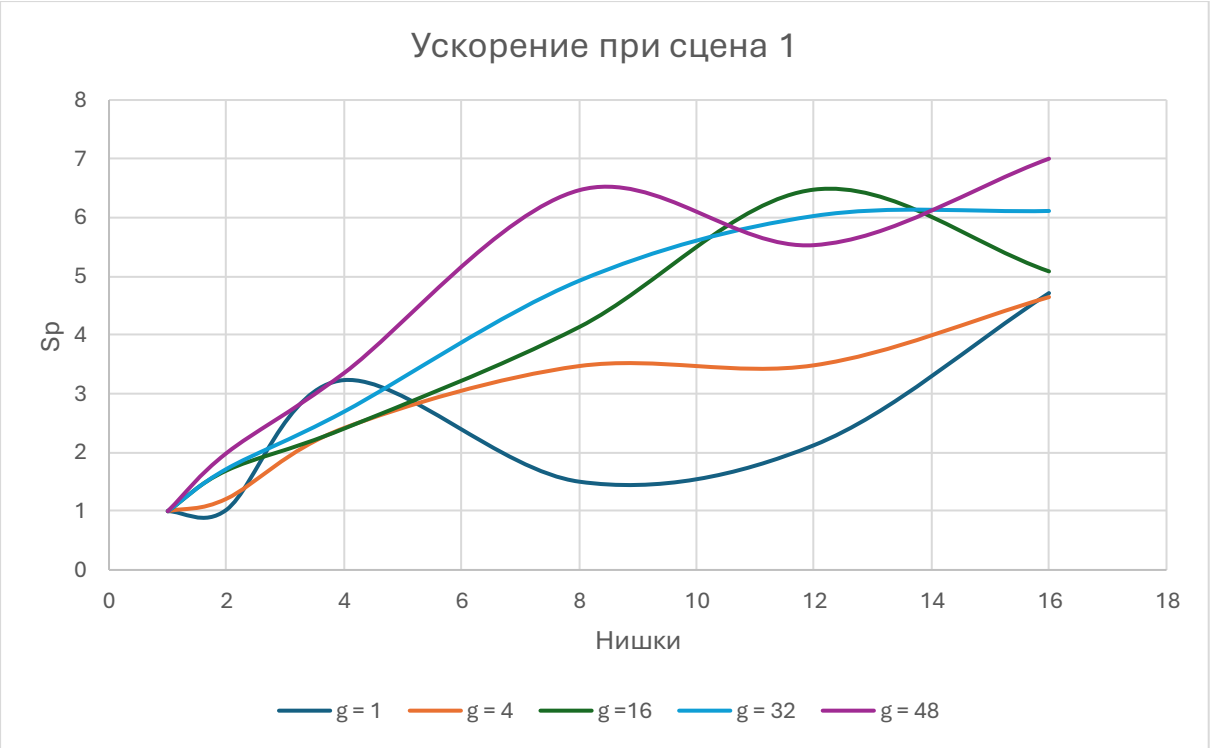
Легенда на резултатите:

#	Идентификатор на теста
p	Брой стартирани нишки
g	Грануларност
Tp(i)	Времето за изпълнение на програмата в секунди при i-тото стартиране
Tp = min()	Минималното време за изпълнение на програмата
Sp	Постигнато ускорение
Ep	Едективност

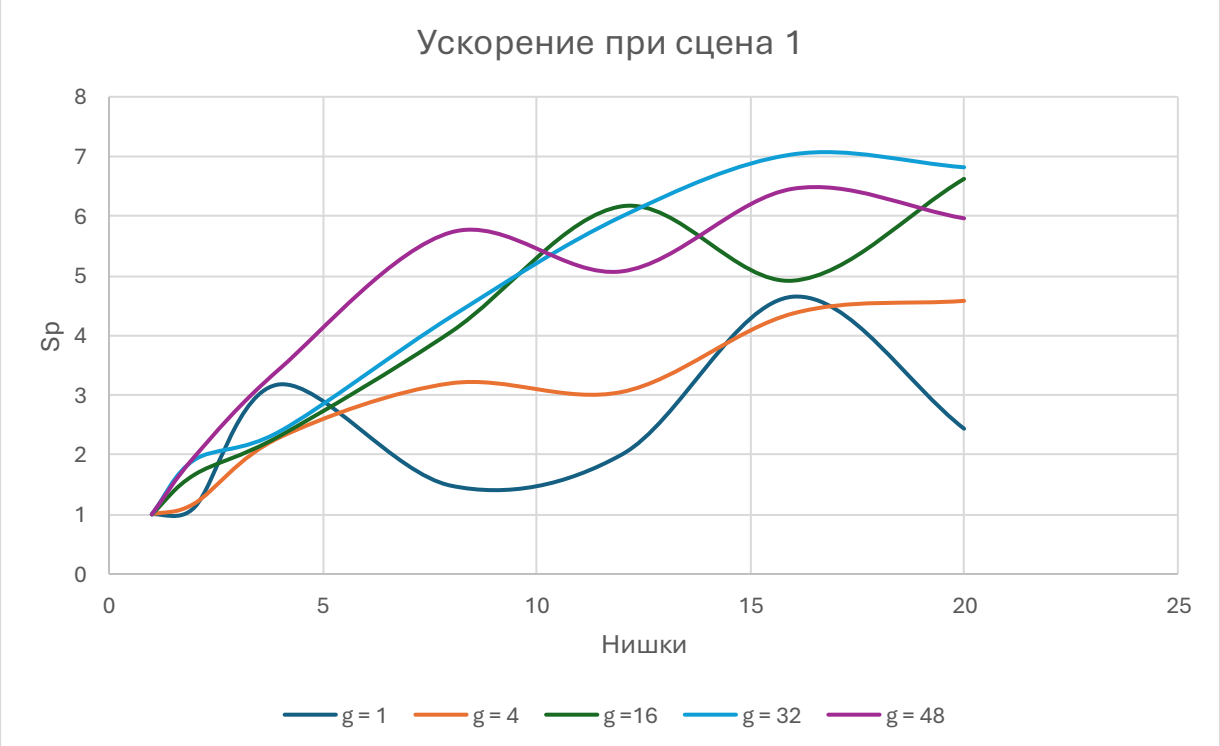
8.1. Първа сцена

8.1.1. Статично балансиране

Тестове извършени с Lenovo Yoga Pro 7 14APH8:

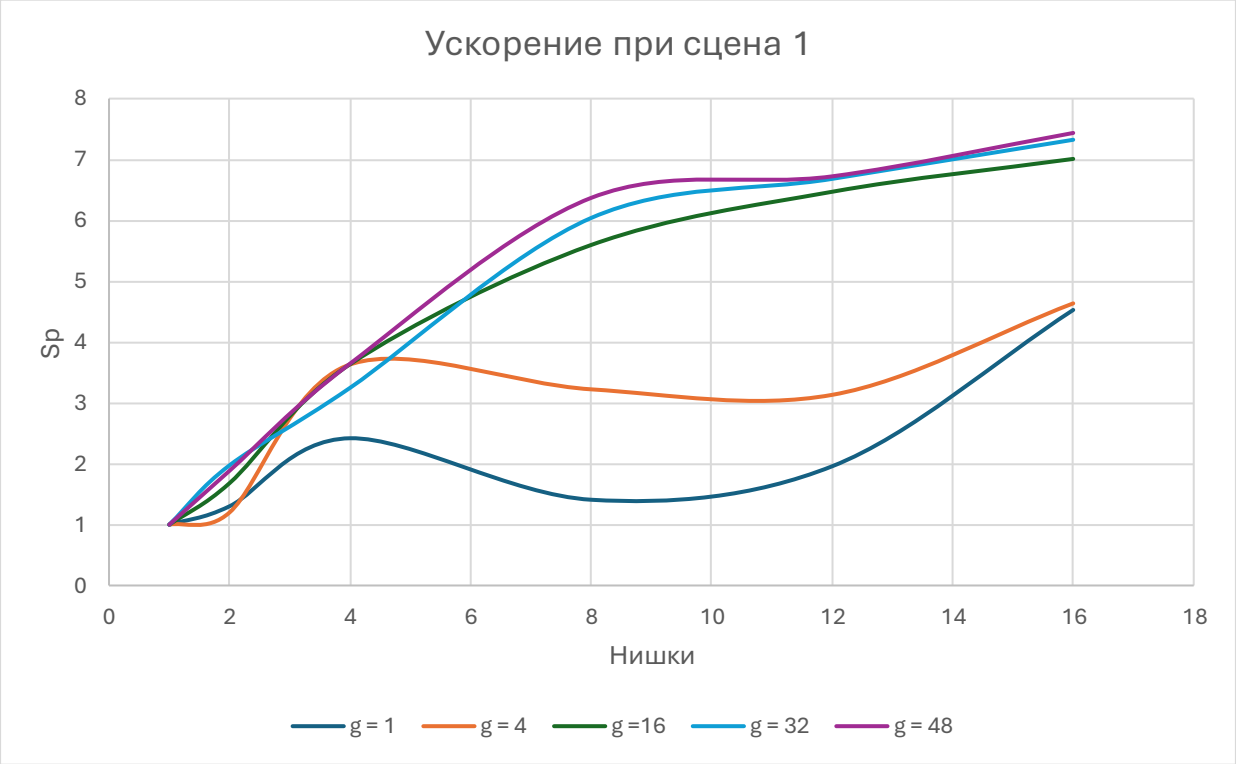


Тестове извършени с Lenovo ThinkPad P1 G5:

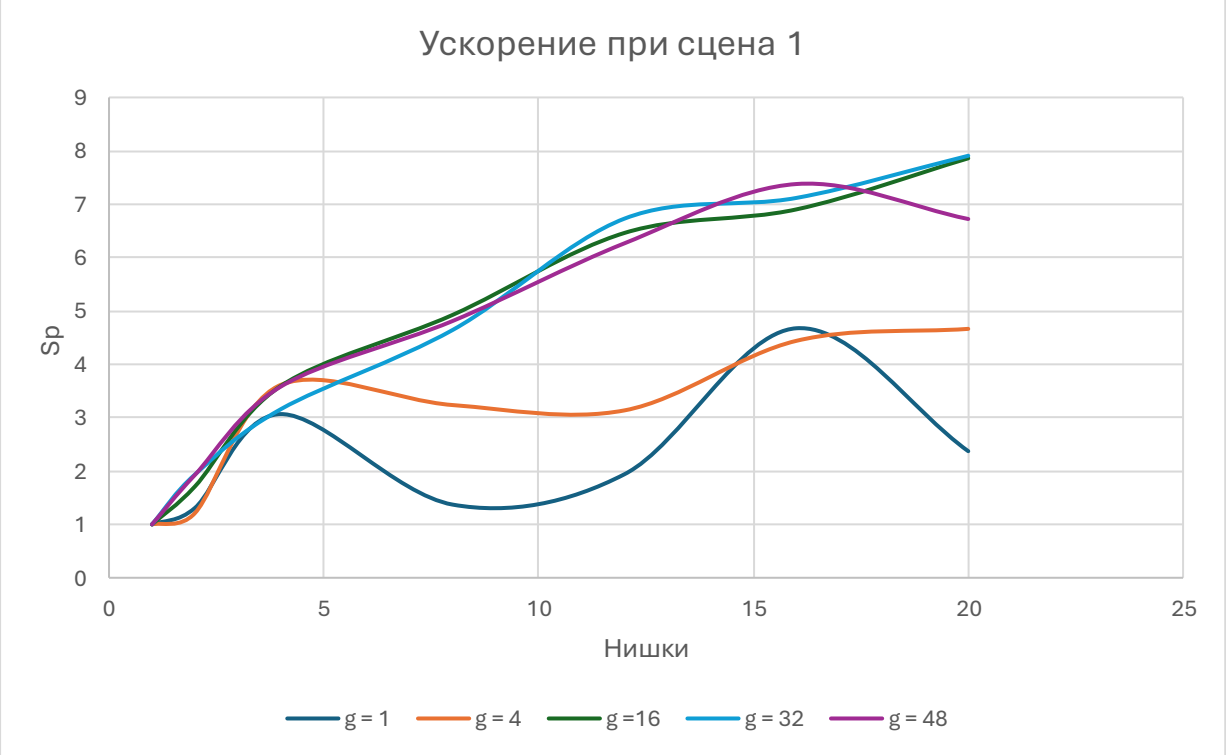


8.1.2. Динамично балансиране

Тестове извършени с Lenovo Yoga Pro 7 14APH8:



Тестове извършени с Lenovo ThinkPad P1 G5:



8.1.3. Анализ на резултатите

При статичното и динамичното балансиране, резултатите между двете машини са почти едни и същи, въпреки разделението на ядрата и техния брой. Лесно се забелязва разлика между статичното и динамичното балансиране. Динамичното балансиране е доста по предсказуемо спрямо зададената грануларност.

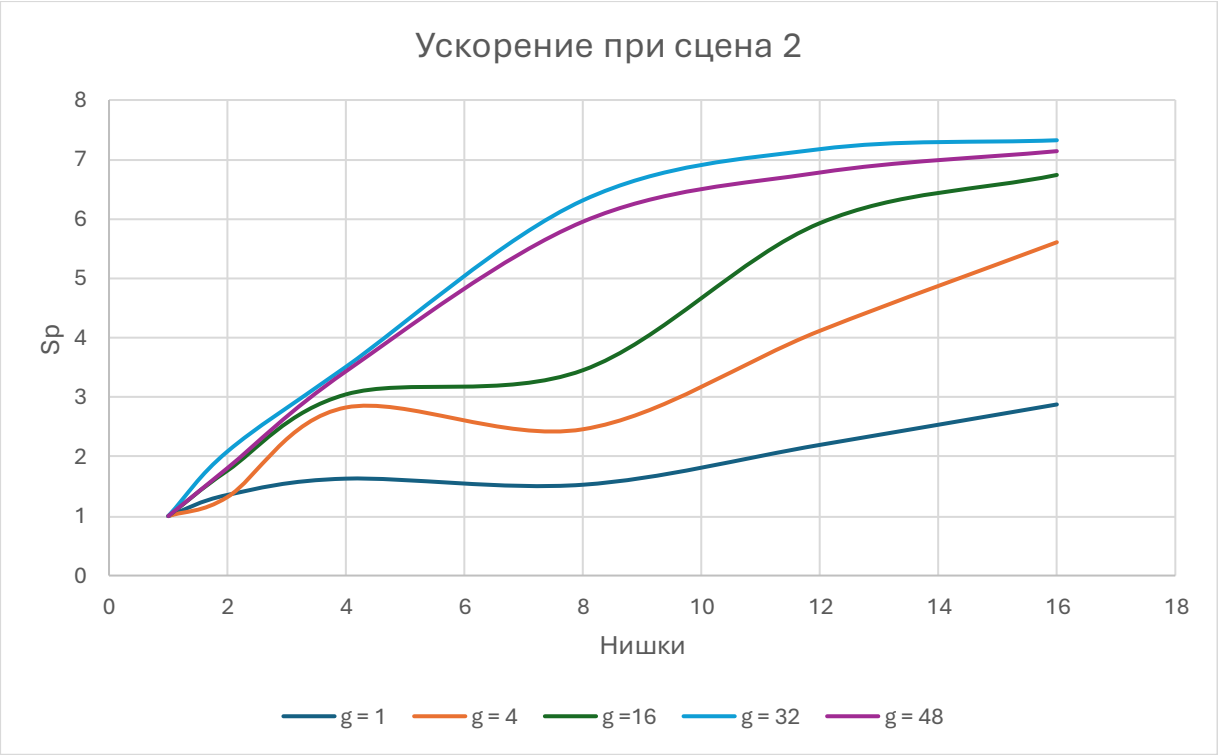
Същото обаче не може да се твърди при статичното балансиране. Забелязваме как фината грануларност увеличава ускорението, докато не се достигне броят на физическите ядра. След това тя става по-малко ефикасна в сравнение със средните грануларности 16 и 32. Като цяло диаграмата е доста по-хаотична в сравнение с динамичното балансиране.

Така като динамичното балансиране има по детерминистично поведение, при него няма нужда да нагаждаме грануларността спрямо броя на нишките, които искаме да пуснем. От средната(16 и 32) до фината грануларност(48) няма особени разлики спрямо полученото ускорение.

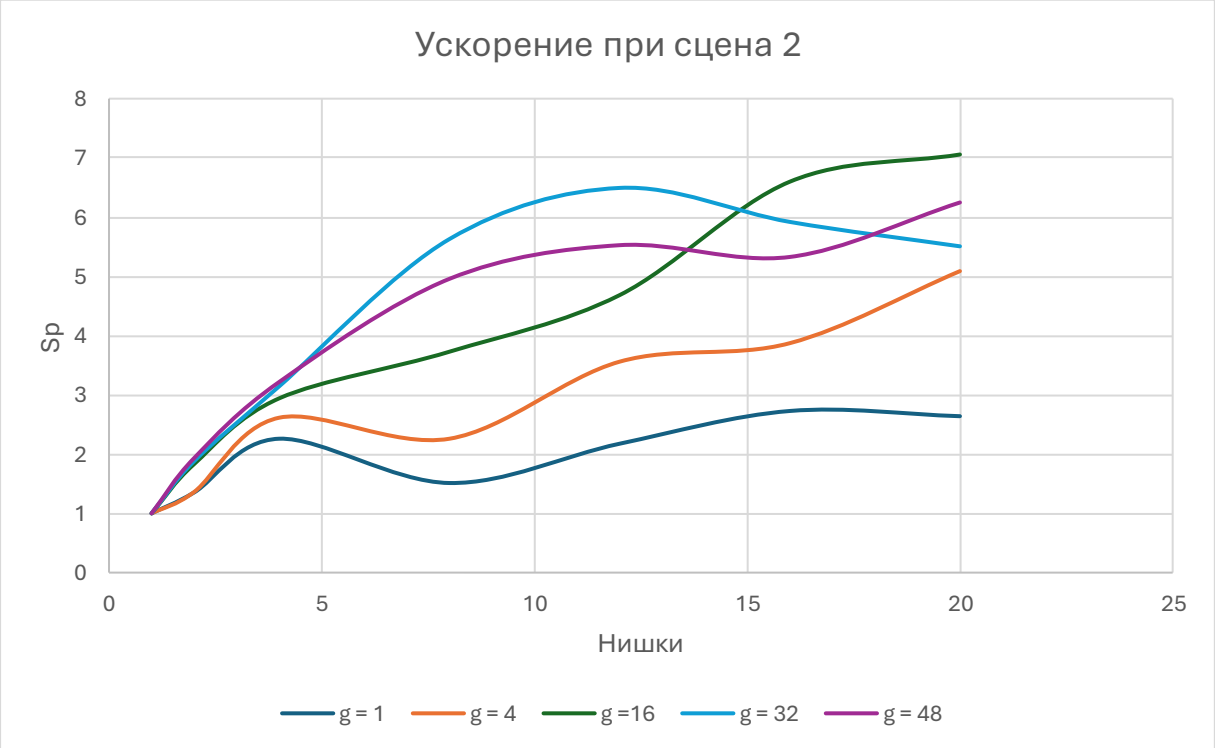
8.2. Втора сцена

8.2.1. Статично балансиране

Тестове извършени с Lenovo Yoga Pro 7 14APH8:

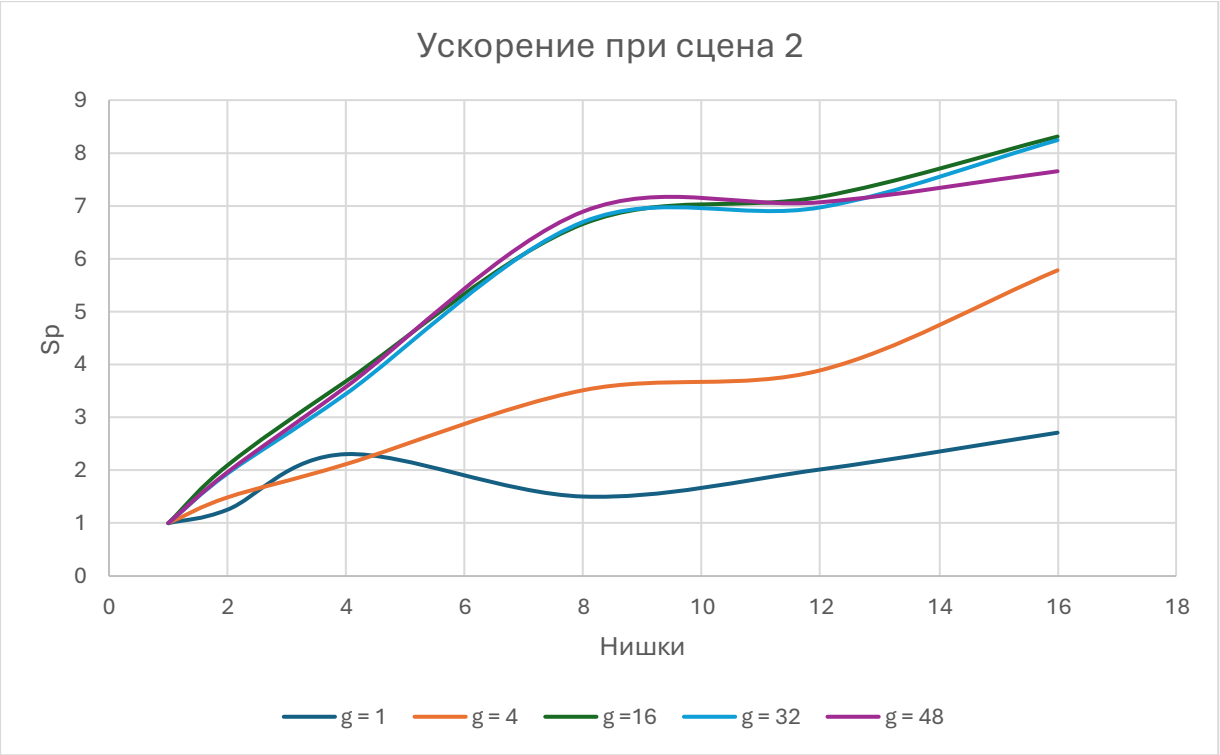


Тестове извършени с Lenovo ThinkPad P1 G5:

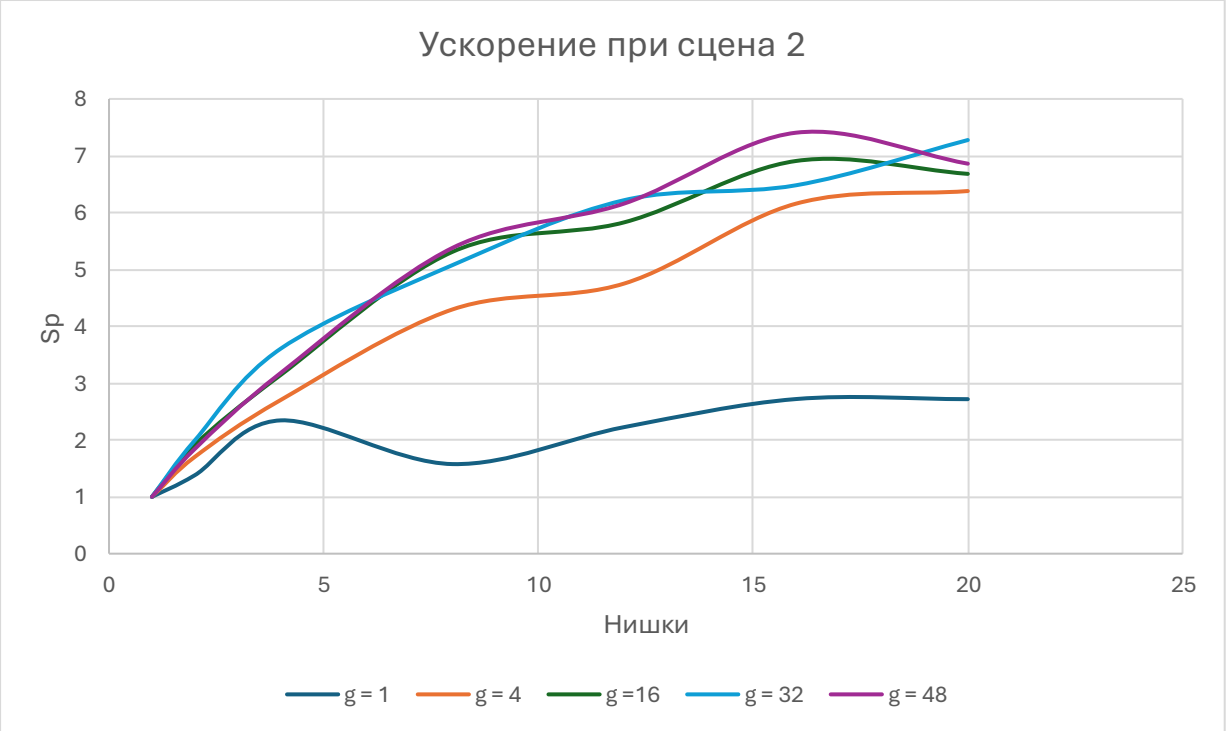


8.2.2. Динамично балансиране

Тестове извършени с Lenovo Yoga Pro 7 14APH8:



Тестове извършени с Lenovo ThinkPad P1 G5:



8.2.3. Анализ на резултатите

Тук наблюденията са почти същите като в миналата сцена. Този път, изненадващо статичното балансиране също изглежда сравнително подредено, за разлика от миналите диаграми. Интересното там е, че при AMD процесора се държи сравнително близко до варианта с динамичното балансиране.

При Intel процесора са малко по-различни нещата. Там интересното е, че получаваме най-добро ускорение със средната грануларност 16. Което ни навява на факта, че отново статичното балансиране е доста по-хаотично.

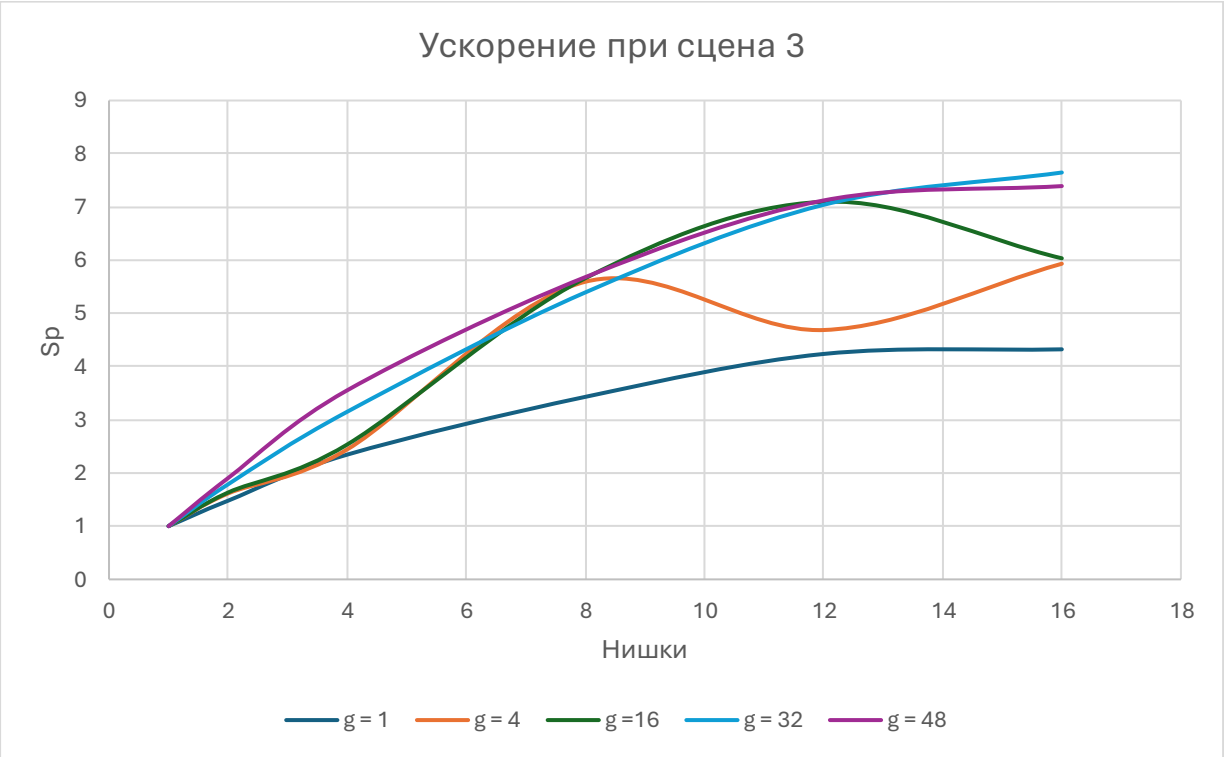
В тази сцена обектите се намират на центъра и заемат около 30% от мястото. Останалото място е празно и не се правят сложни изчисления. Вероятно това е причината за такива аномалии.

Тук отново неоспоримо е, че динамичното балансиране ни дава по-предвидими и по-добри резултати от статичното балансиране.

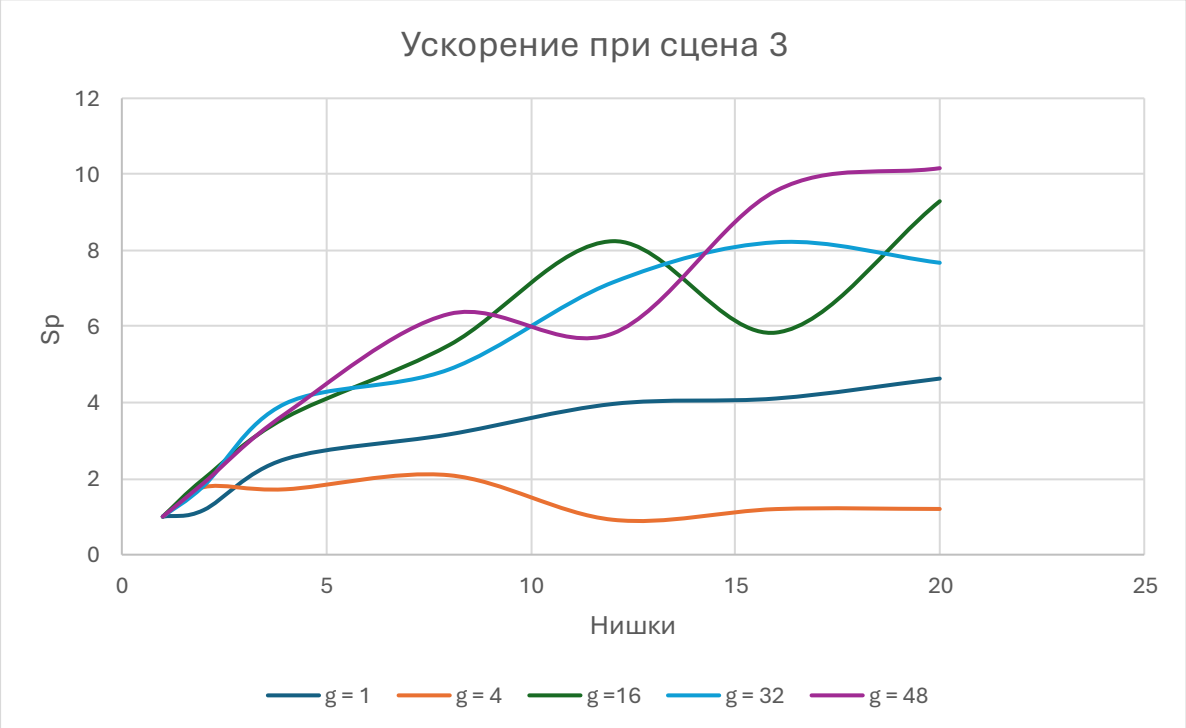
8.3. Трета сцена

8.3.1. Статично балансиране

Тестове извършени с Lenovo Yoga Pro 7 14APH8:

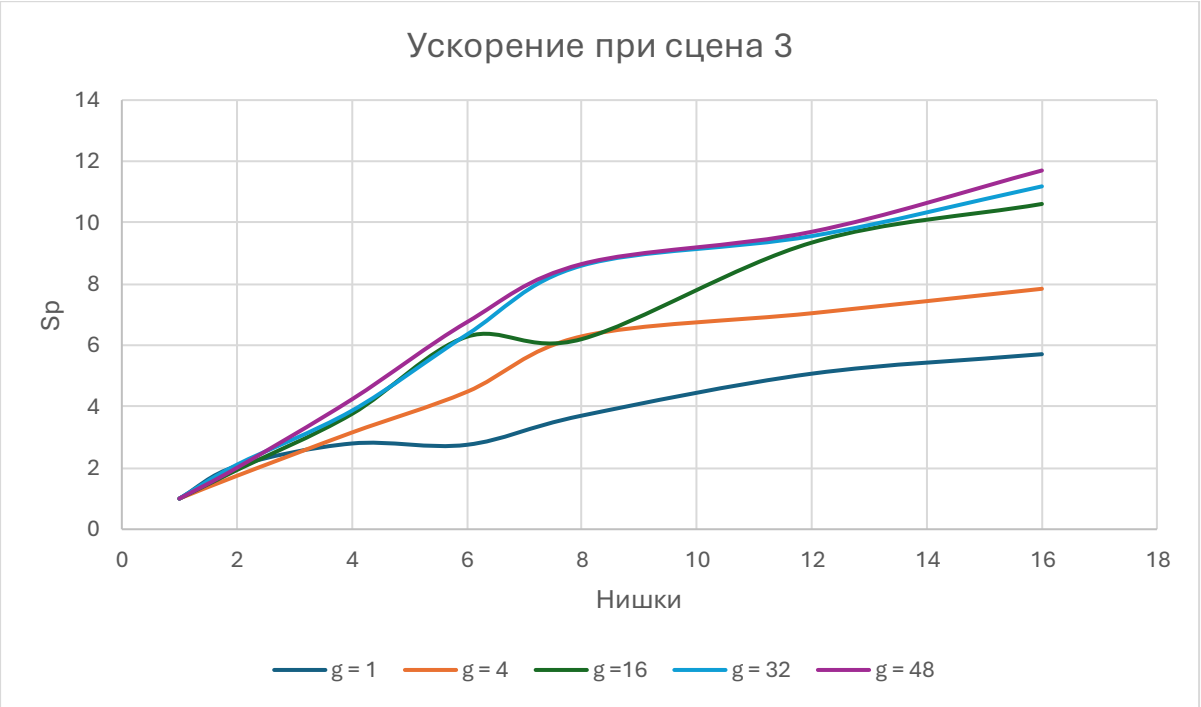


Тестове извършени с Lenovo ThinkPad P1 G5:

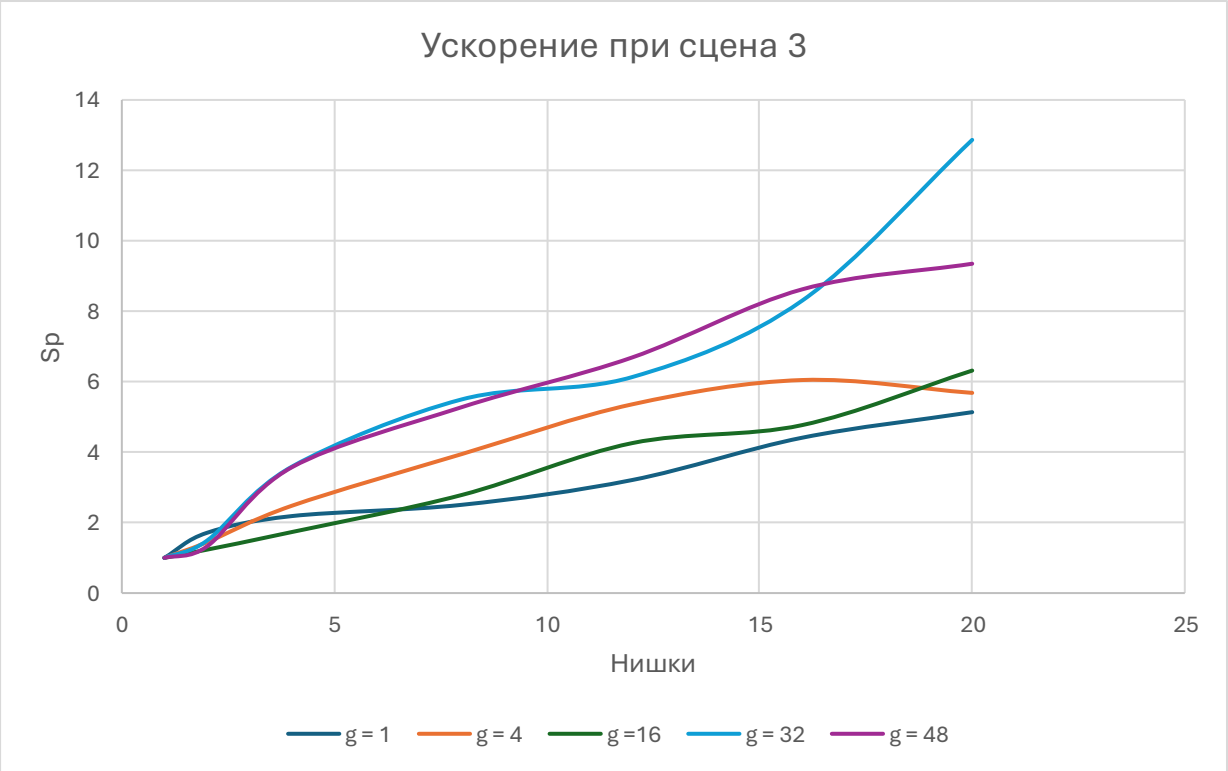


8.3.2. Динамично балансиране

Тестове извършени с Lenovo Yoga Pro 7 14APH8:

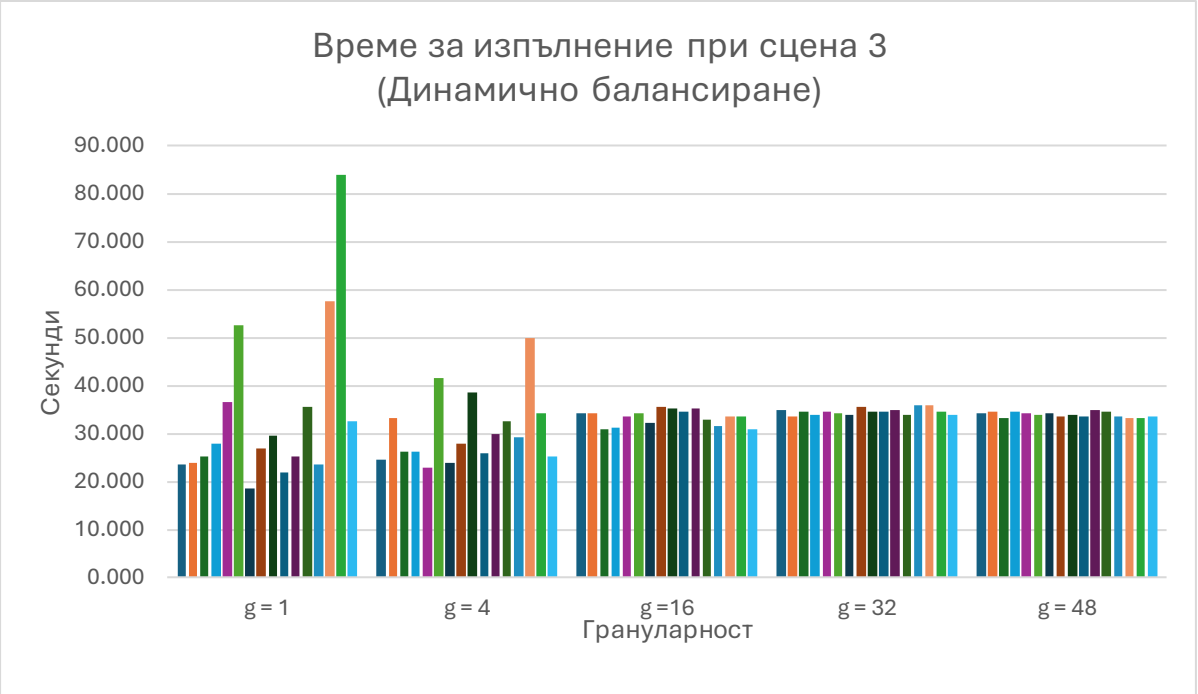


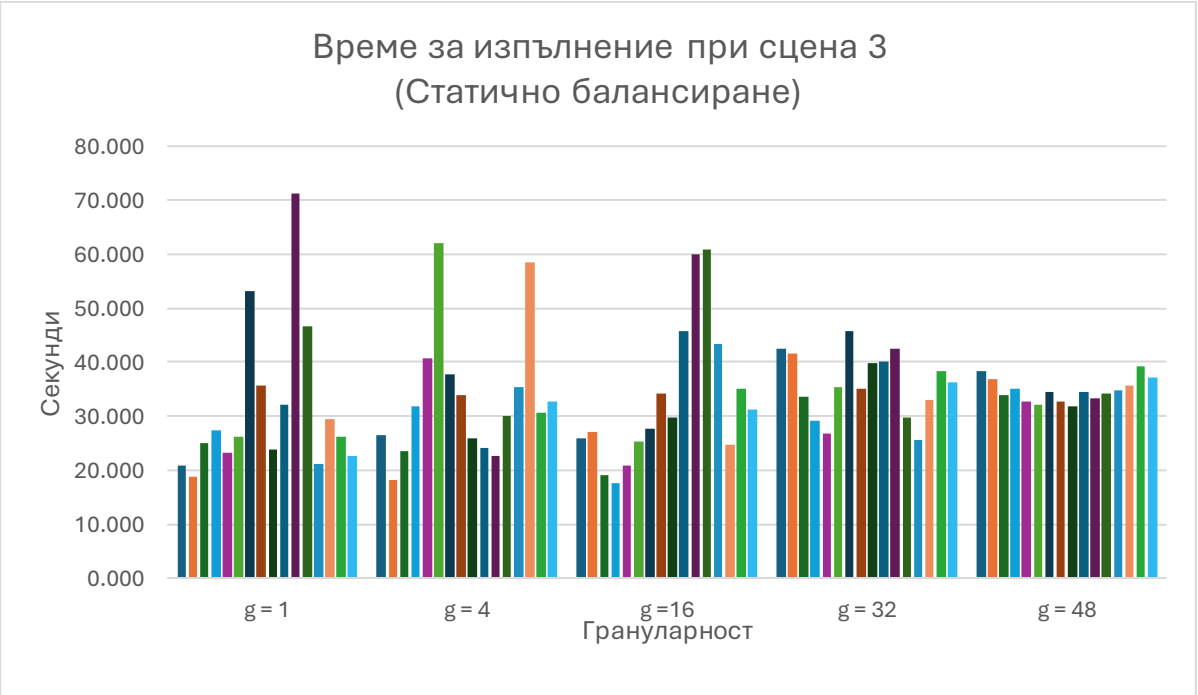
Тестове извършени с Lenovo ThinkPad P1 G5:



8.3.3. Време за изпълнение

Сравнение между разпределеното време на изпълнение при статичното и динамичното балансиране при изпълнение на 16 нишки върху Lenovo Yoga Pro 7 14APH8:





8.3.4. Анализ на резултатите

Тук нещата са много по-интересни от колкото предишните сцени. Тази сцена е най-сложа от тестваните, така като тя съдържа в себе си всеки материал, който се поддържа от програмата, както и голямо количество от триъгълници. Това е и единствената сцена в която се симулира светлината напълно(чрез Global Illumination).

Нещо, което се набива на очи е фактът, че при Intel процесора, се получава следната аномалия: тестовете с грануларност 16 при динамичното балансиране дават по-ниско ускорение, отколкото тези с грануларност 4. Това е коренно различно поведение от наблюдаваното до момента.

Друг интересен случай е неестествено високата ефикасност на AMD процесора при динамичното балансиране. Тествано е няколко пъти и резултатът е един и същ.

Тук ускоренията са най-съществени отново при динамичното балансиране, въпреки странното държание на Intel-ската машина. Този път и двата метода на балансиране са сравнително хаотични и не е много лесно да се предвиди дадено ускорение в сравнение с миналите тестове.

Най-интересното е обаче метриката за времето на изпълнение на нишките спрямо избрания метод на балансиране. Лесно се вижда, че с увеличаване на грануларността, разпределението на задачите става все по равностойно между нишките. Разликата е, че при динамичното балансиране, задачите се уравновесяват още при средна грануларност(16), а при статичното балансиране – чак при фина грануларност(48).

И при тези тестове се вижда лесно, че динамичното балансиране е по-добрият вариант.

9. Заключение

При всички тестове забелязахме как ускорението се покачва страшно бързо с увеличаване на броя нишки, докато не достигнем броя на физическите ядра. Разбира се това е напълно нормално явление. Логическите ядра обещават до 30% увеличение на скоростта. Въпреки това ние можем да се възползваме и от тях и да изстискаме възможно най-много производителност от машината.

Спрямо проведените тестове, статичното и динамичното балансиране имат своите добри и лоши страни. Но за програма от този вид, динамичното балансиране е по-добрият вариант. Дава по-добра предвидимост на резултатите, както и разпределя задачите добре още при средната грануларност, за разлика от статичното балансиране.

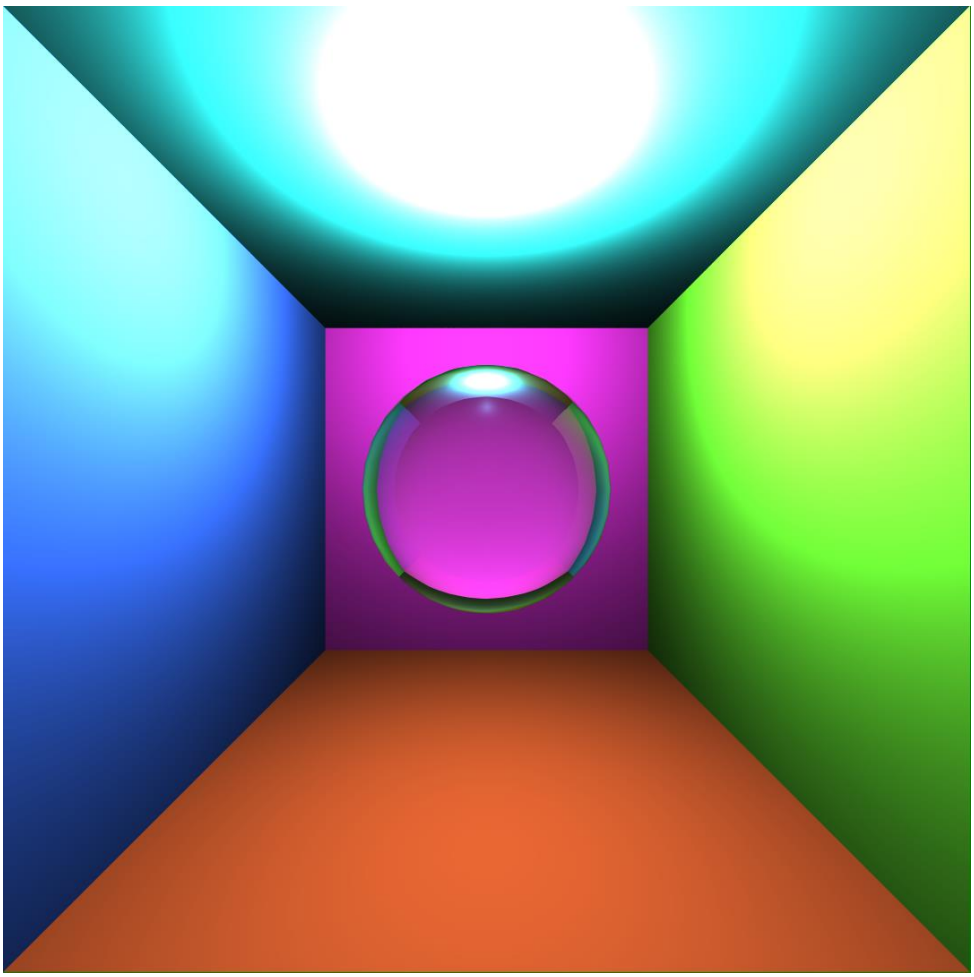
Забелязахме, че в някои случаи е възможно да се използва статично балансиране, като при по-едрите грануларности. Разбира се, зависи от целта и сцената, която изобразяваме. Но в общия случай динамичното балансиране е по-добрият метод в такава програма.

10. Източници

- Код на проекта - https://github.com/PoinP/Iris
- Scratchapixel - https://www.scratchapixel.com/index.html
- Ray Tracing in One Weekend – Peter Shirley - https://raytracing.github.io/
- Techpowerup - https://www.techpowerup.com/

11. Сцени

Първа сцена:



Втора сцена:

Трета сцена: