Алгоритм удаления невидимых поверхностей на основе программных проверок видимости

В.И. Гонахчян <pusheax@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Визуализация больших трехмерных сцен занимает существенное время. Для сокращения количества обрабатываемых объектов используются методы удаления невидимых поверхностей. В статье рассматривается семейство интерактивных методов удаления невидимых поверхностей, обладающих пространственной и временной когерентностью. Наиболее распространенным является метод с использованием аппаратных запросов видимости. Однако посылка и получение результатов запросов видимости занимают значительное время при большом количестве объектов. Предлагается алгоритм удаления невидимых поверхностей на основе программных проверок видимости относительно составленного на графическом процессоре буфера глубины. Предлагается эвристика для определения высоты иерархии, соответствующей наибольшей эффективности проверок видимости. Предложенный алгоритм позволяет сократить количество команд визуализации, что улучшает производительность визуализации трехмерных сцен с большим количеством объектов по сравнению с алгоритмом, основанным на аппаратных запросах видимости.

Ключевые слова: визуализация трехмерных сцен; проверка видимости; окто-дерево.

DOI: 10.15514/ISPRAS-2018-30(2)-5

Для цитирования: Гонахчян В.И. Алгоритм удаления невидимых поверхностей на основе программных проверок видимости. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 81-98. DOI: 10.15514/ISPRAS-2018-30(2)-5

1. Введение

Визуализация больших трехмерных сцен является сложной вычислительной задачей. Для ускорения визуализации используются методы удаления невидимых поверхностей [1], [2]. Рассмотрим наиболее распространенные методы удаления невидимых поверхностей и некоторые их недостатки при визуализации больших трехмерных сцен.

В работе [3] предложен алгоритм удаления невидимых поверхностей на основе иерархического z-буфера. Z-буфер делится на 4 части так, что в каждой из них записывается самое дальнее значение z. Процесс деления повторяется

до пикселей. Z-пирамида позволяет быстро исключить треугольники с помощью сравнения минимального значения z треугольника и максимального значения z в области. При программной реализации происходит затратное обновление пирамиды при добавлении новых объектов, что снижает эффективность при работе с большими сценами.

В работе [4] предлагается составлять BSP-дерево с объектами статических архитектурных сцен и декомпозицией по осям архитектурных сцен. Получается иерархия, которая соответствует структуре комнат в здании. Информация о видимости хранится в графе комнат и порталов. Видимость комнат определяется путем растеризации порталов. Однако для составления графа требуется много вычислительных ресурсов, и этот метод работает только на статических сценах. Коммерческая программа Umbra вычисляет воксельное представление сцены [5]. Пустые воксели используются как порталы между разными частями сцены. Программная растеризация порталов используется для определения видимости разных частей сцены. Этот алгоритм эффективно находит загороженные объекты в статических сценах и широко используется в компьютерных играх.

В работе [6] используются маски покрытия (coverage mask) для определения видимости. На вход подается массив полигонов в порядке удаления от камеры (front-to-back). Происходит рекурсивное деление изображения на квадро-дерево до тех пор, пока видимость полигонов не может быть определена для каждого квадранта. Главное преимущество этого подхода заключается в низких требованиях к памяти и отсутствие перезаписи пикселей. Однако требуется специализированное аппаратное обеспечение для эффективной реализации.

В работе [7] предлагается использовать иерархические маски покрытия для отбраковки невидимых объектов. Сначала происходит растеризация потенциально блокирующих видимость объектов, затем обход иерархии, и выполняются проверки видимости. Алгоритм позволяет делать приблизительное определение видимости, когда порог видимости меньше единицы. Главный недостаток алгоритма заключается в сложном процессе отбора блокирующих объектов, который выбирает большие объекты с маленьким числом полигонов.

В данной статье рассматриваются методы удаления невидимых поверхностей, обладающих пространственной и временной когерентностью видимости, на основе аппаратных и программных проверок видимости [8]11]. Аппаратный запрос видимости – это способ нахождения видимых граней полиэдра на графическом процессоре [12]. Запрос видимости останавливается, когда находит первую видимую грань. Пространственная когерентность позволяет определить видимость объектов в области пространства (узле дерева). Например, если здание невидимо, то объекты внутри здания также невидимы. Временная когерентность позволяет определить видимость в будущем по текущей видимости. Например, если объект виден в текущем кадре, и камера

движется непрерывно, то можно считать его видимым следующие несколько кадров.

В работе [8] рассматриваются способы оптимальной посылки аппаратных запросов видимости с использованием расширения NV_occlusion_query. Результаты видимости в текущем кадре используются в следующем кадре. Главные проблемы с производительностью возникают из-за простаивания процессора (CPU starvation) и графического процессора (GPU starvation). Результаты запросов видимости проверяются в следующем кадре, чтобы избежать простаивания процессора. Ранее видимые объекты визуализируются в начале текущего кадра, чтобы избежать простаивания графического процессора. Авторы использовали k-мерное дерево, построенное с применением эвристики на основе площади узлов дерева [13]. Запросы видимости посылаются на листья и по результатам определяется видимость верхних уровней иерархии. Этот метод подходит для визуализации сложных трехмерных сцен, если взять иерархию, которая не требует существенных затрат при движении объектов.

В работе [9] предлагается использовать программную растеризацию для проверки видимости вместо аппаратных запросов видимости. Сначала происходит растеризация треугольников больших объектов на центральном процессоре, составляется z-буфер (full sized tiled z-buffer). Затем проверяется видимость ограничивающих параллелепипедов остальных объектов сначала относительно суммарного z-буфера, который задает максимальное значение z в пределах полоски (tile). Если треугольники загорожены относительно суммарного z-буфера, то объект невидим. Иначе требуется полноценная растеризация треугольников и сравнение глубины относительно полноразмерного буфера глубины. Это метод позволяет делать проверки видимости без синхронизации с графическим процессором, но для него требуется отбор блокирующих объектов, который лучше всего делается вручную. Растеризация блокирующих объектов с большим количеством полигонов может быть довольно дорогой, а версия объекта с низким уровнем детализации дает приблизительные результаты.

В работе [10] предложен вероятностный критерий для уменьшения количества аппаратных запросов видимости, который позволяет не посылать запросы видимости, когда визуализация объектов в узле иерархии дешевле. В работе [11] предлагаются дальнейшие способы уменьшения запросов видимости путем посылки одного запроса на группу узлов дерева. Авторы использовали иерархию p-hbvo (polygon-based hierarchical bounding volume decomposition [14], которая хорошо справляется со статическими сценами, но не подходит для динамических трехмерных сцен.

В данной статье предложен алгоритм, который является развитием идей, описанных в работах [8], [9]. Для хранения объектов сцены используется окто-дерево с множественными ссылками [15]. Окто-деревья имеют многочисленные приложения в компьютерной графике и вычислительной

геометрии: удаление объектов, не попадающих в усеченную пирамиду камеры, поиск соседей, определение коллизий, планирование движения [16][17][18].

Оставшаяся часть статьи имеет следующую структуру. В разд. 2 описаны основные формулы, которые используются в программной растеризации и проверках видимости. В разд. 3 описан предлагаемый алгоритм удаления невидимых поверхностей. В разд. 4 представлены результаты сравнения производительности предлагаемого алгоритма и алгоритма на основе аппаратных запросов видимости. В разд. 5 приводятся основные выводы.

2. Проверки видимости в программном режиме

В алгоритме, предлагаемом в данной статье, используется код для программной растеризации, основанный на векторных инструкциях [19]. Поскольку скорость проверок видимости имеет большое значение, и этому уделено недостаточно внимания в работе [9], рассмотрим более подробно то, как осуществляются программные проверки видимости.

Для определения принадлежности точки треугольнику используется функция ребра $F_{AB}(x,y) = (y_A - y_B)x + (x_B - x_A)y + (x_A\ y_B - x_B\ y_A)$ [20]. $F_{AB}(x,y) > 0$, если точка (x,y) находится слева от отрезка AB, $F_{AB}(x,y) = 0$, если точка (x,y) находится на прямой, проходящей через AB, $F_{AB}(x,y) < 0$, если точка (x,y) находится справа от отрезка AB. Треугольник ABC состоит из трех отрезков: AB, BC, CA. Точка (x,y) принадлежит треугольнику, когда $F_{AB}(x,y) \geq 0$, $F_{BC}(x,y) \geq 0$, $F_{CA}(x,y) \geq 0$.

Глубина z(p) точки p вычисляется с помощью барицентрической интерполяции: $z(p) = \alpha * z(A) + \beta * z(B) + \gamma * z(C)$, где $\alpha = \frac{F_{BC}(p)}{2*S_{ABC}}$, $\beta = \frac{F_{CA}(p)}{2*S_{ABC}}$, $\gamma = \frac{F_{AB}(p)}{2*S_{ABC}}$. Поскольку $\alpha + \beta + \gamma = 1$, $z(p) = z(A) + \beta * (z(B) - z(A)) + \gamma * (z(C) - z(A))$.

Точка p является видимой, если $z(p) \leq \text{depth\_buffer}(p)$; ближняя плоскость соответствует 0, дальняя плоскость соответствует 1 в буфере глубины. Треугольник ABC считается видимым, если видима хотя бы одна точка из ограничивающего прямоугольника треугольника ABC. Октант окто-дерева считается видимым, если виден хотя бы один треугольник октанта.

Для эффективных проверок видимости используются следующие методы: проекция вершин AABB, инкрементальное обновление коэффициентов функции ребра, использование векторных инструкций SSE. Рассмотрим более подробно каждую из них.

Для наивной проекции вершин AABB требуется умножить матрицу проекции на 8 вершин (128 умножений). Рассмотрим две вершины $v_0(x,y,z)$, $v_1(x + \Delta x, y, z)$. Заметим, что $M * v_1 = M * v_0 + M * (\Delta x, 0, 0)$. Для вычисления $M * v_1$ дополнительно требуется посчитать $m_{00} * \Delta x, m_{10} * \Delta x, m_{20} * \Delta x, m_{30} * \Delta x$.

Таким образом, для трансформации AABB требуется $16 + 4 * 3 = 28$ умножений, что значительно ускоряет трансформацию октантов.

Рассмотрим метод вычисления функции ребра для соседних пикселей треугольника (x, y), (x+1, y), $F_{AB}(x + 1, y) = (y_A - y_B)(x + 1) + (x_B - x_A)y + (x_A\, y_B - x_B * y_A) = F_{AB}(x, y) + (y_A - y_B)$. Таким образом, для перехода к следующему пикселю на данной строке требуется одно сложение. Для треугольника ABC нужно посчитать коэффициенты $(y_A - y_B), (x_B - x_A), (x_A\, y_B - x_B\, y_A)$ один раз, а потом просто добавлять их при обходе пикселей внутри ограничивающего прямоугольника ABC.

Векторные инструкции SSE позволяют выполнять 4 операции за одну инструкцию. AABB состоит из 12 треугольников. Поместив координаты вершин треугольников в SSE регистры, можно за одну инструкцию вычислять 4 коэффициента функции ребра (для 4 треугольников). При обходе пикселей треугольника рассматриваются сразу 4 соседних пикселя.

Рассмотрим суммарный буфер глубины, в котором проставлено максимальное значение z для каждой группы пикселей (tile). Для отбора видимых объектов с отбраковкой на основе буфера глубины производится проекция вершин объекта на экран и проверка относительно суммарного буфера глубины. Если находятся треугольник и группа пикселей, пересекающаяся с треугольником, такие что $\min Z_\Delta < \max Z_{tile}$, то треугольник считается видимым относительно суммарного буфера глубины. В этом случае делается проверка видимости каждого пикселя внутри треугольника. Если таких треугольников нет, то объект считается загороженным и вычисление глубины каждого пикселя не требуется (см. рис. 1). Проверки относительно буфера глубины осуществляются в основной памяти на CPU, что разгружает GPU.

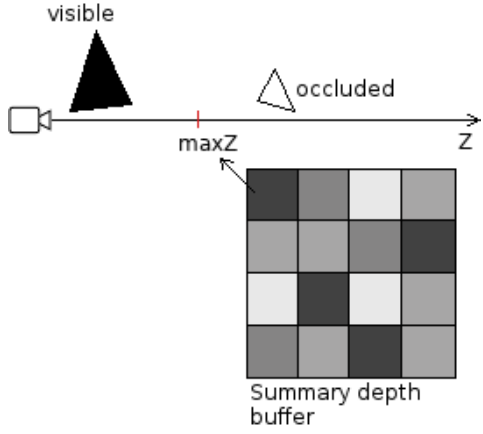


Рис. 1. Отбраковка треугольников на основе суммарного буфера глубины
Fig. 1. Rejecting triangles using summary depth buffer.

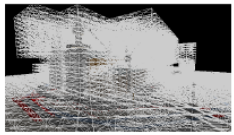
Описанные методы позволяют значительно сократить время проверок видимости октантов.

3. Алгоритм удаления невидимых поверхностей

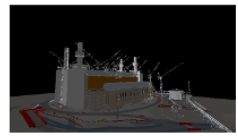
Предлагаемый алгоритм является развитием алгоритмов, описанных в работах [8], [9]. Однако, в отличие от Coherent hierarchical culling, аппаратные запросы видимости не используются. В отличие от Software occlusion culling, выполняются проверки видимости октантов в следующем кадре, т. е. учитывается пространственная и временная когерентность. Сначала рассмотрим основные шаги предлагаемого алгоритма, потом опишем отличия от алгоритма на основе аппаратных проверок видимости [21], эвристику для определения уровня иерархии, технику консервативного понижения глубины вершин октанта для устранения мерцаний.

Объекты сцены помещаются в окто-дерево с множественными ссылками. Для каждого кадра выполняются следующие шаги алгоритма. Сначала скачивается буфер глубины предыдущего кадра с графического процессора. Для этого нужно завершить визуализацию всех посланных объектов. Хотя это и вызывает простой процессора, затрачиваемое время оказывается меньше, чем затраты на посылку и прием запросов видимости при использовании аппаратных запросов видимости.

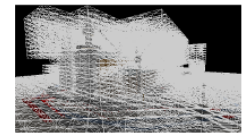
Hardware queries (АПВ)



1. Get occlusion query results that determine octant visibility.

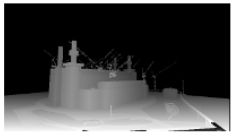


2. Render visible.

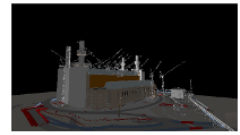


3. Send occlusion queries for each octant on last level.

Software queries (ППВ)



1. Download depth buffer of the previous frame. Perform software visibility checks.



2. Render visible.

Рис. 2. Основные этапы работы алгоритмов АПВ и ППВ
Fig. 2. Main stages of hardware and software occlusion culling algorithms.

Далее формируется массив октантов нижнего уровня, которые попадают в область видимости камеры (frustum culling). Потом выполняются программные проверки видимости относительно полученного буфера глубины, которые выполняются параллельно на всех доступных ядрах процессора с доступом к буферу глубины только на чтение. Результаты проверок видимости записываются в массив байт. Затем визуализируются объекты внутри видимых октантов.

На рис. 2 показаны этапы, которые занимают основное время при выполнении алгоритмов АПВ (аппаратные проверки видимости) и ППВ (программные проверки видимости). Более подробно АПВ описан в [21], главное отличие от ППВ – использование объектов “opengl query” для проверок видимости октантов [12]. Этап 2 занимает примерно одинаковое время, потому что результаты видимости в обоих случаях одинаковые. Этап 1 выполняется быстрее в ППВ при большом количестве уровней иерархии, потому что загрузка буфера глубины – разовая операция, которая происходит после составления буфера предыдущего кадра (см. табл. 1). Также ППВ дает прирост производительности за счет сокращения количества посылаемых на графический процессор команд (этап 3).

Табл. 1. Сравнение времени выполнения основных этапов АПВ и ППВ при визуализации сцены 2
Table 1. Comparison of execution time of hardware and software occlusion queries when rendering scene 2

Высота окто-дерева	Загрузка буфера глубины	Программные проверки видимости	Получение результатов аппаратных проверок видимости	Отправка аппаратных запросов видимости
3	1.2 мс	0.09 мс	0.24 мс	0.54 мс
4	1.2 мс	0.15 мс	1.42 мс	2.3 мс
5	1.2 мс	0.35 мс	8.25 мс	15.26 мс
6	1.2 мс	2.1 мс	57.9 мс	106.9 мс
7	1.2 мс	11.8 мс	-	-

Высота окто-дерева сильно влияет на производительность предлагаемого алгоритма. Рассмотрим эвристику для выбора оптимальной высоты окто-дерева. Пусть количество октантов $\mathrm{N} = 8^{m}\mathrm{C}$, где m – количество уровней окто-дерева, C – степень разреженности окто-дерева. Рассмотрим типовую сцену, в которой M объектов распределены равномерно по кубу из N октантов. Будем считать, что в камеру попадают три стороны куба.

Пусть $\mathrm{n} = \sqrt[3]{N}$, тогда сторона куба содержит $\frac{n^2}{N}M$ видимых объектов, три стороны содержат $\frac{\mathrm{n}^2+\mathrm{n}(\mathrm{n}-1)+(\mathrm{n}-1)(\mathrm{n}-1)}{N}M$ видимых объектов, визуализация видимых объектов занимает время $\frac{\mathrm{n}^2+\mathrm{n}(\mathrm{n}-1)+(\mathrm{n}-1)(\mathrm{n}-1)}{N}MT_{obj}$, проверки видимости занимают время NT_{check}. Предполагается, что суммарное время посылки октантов и объектов является узким местом работы алгоритма, а графический процессор справляется с визуализацией объектов. Эти формулы позволяют определить даст ли прирост производительности визуализации переход на следующий уровень.

При переходе на следующий уровень количество проверяемых октантов увеличится, а количество видимых объектов трех сторон куба уменьшится в два раза за счет деления по оси, перпендикулярной соответствующей стороне куба. Таким образом, количество проверок видимости увеличится, а количество объектов, которые посылаются на визуализацию, сократится. Это позволяет сделать примерную оценку высоты окто-дерева, которая дает минимальное время составления кадра: $(N_{next} - N_{prev})T_{check} < \frac{\mathrm{n}^2+\mathrm{n}(\mathrm{n}-1)+(\mathrm{n}-1)(\mathrm{n}-1)}{2N_{prev}}MT_{obj}$.

В табл. 2 приведены эвристические оценки высоты окто-дерева и соответствующие времена составления кадров, когда в область видимости камеры попадает вся сцена. Результаты показали, что предложенную эвристику можно использовать при визуализации трехмерных сцен с большим количеством объектов.

Табл. 2. Определение высоты окто-дерева для эффективных проверок видимости
Table 2. Determination of octree depth for effective occlusion culling

	Сцена 1	Сцена 2	Сцена 3	Сцена 4
Время составления кадра при высоте 4	77.5 мс	206.5 мс	113.0 мс	100.4 мс
Время составления кадра при высоте 5	55.1 мс	147.0 мс	96.3 мс	84.0 мс
Время составления кадра при высоте 6	54.5 мс	118.5 мс	98.0 мс	57.7 мс
Время сост. кадра при высоте 7	93.0 мс	138.7 мс	162.0 мс	125.5 мс
Эвристическая	6	6	6	6

оценка высоты окто-дерева				

Из-за того, что глубина z вычисляется по-разному на CPU и на GPU, накапливаются ошибки округления чисел с плавающей точкой, и тот же самый пиксель имеет другую глубину z в проверках видимости. Это может вызывать мерцания, когда октант попеременно то видим, то невидим. Для корректных проверок видимости нужно консервативно занизить глубину, вычисленную на CPU.

Рассмотрим задачу определения равенства двух чисел с плавающей точкой. Числа x и y можно считать равными, если $|x - y| < K * epsilon * |x + y|$, где epsilon примерно равняется 10^{-5} (разница между наименьшим числом с плавающей точкой больше единицы и единицей). Однако глубина может отличаться совсем мало, поскольку она пропорциональна $\frac{1}{z}$. Кроме того, неизвестно, какое значение K стоит брать, чтобы превысить накопившуюся ошибку округления.

Другой вариант – взять часть расстояния между ближней и дальней гранью октанта вдоль z для консервативного понижения глубины вершин. После проекции $z' = \frac{2fn}{(f-n)z} + \frac{f+n}{f-n}$, где n – расстояние до ближней плоскости, f – расстояние до дальней плоскости, z – расстояние от камеры до октанта вдоль оси Z. Посчитаем расстояние между гранями октанта: $\Delta z = \frac{2fn}{(f-n)}(\frac{1}{(z-a)} - \frac{1}{(z+a)})$, где a – характерный размер октанта. В предлагаемом алгоритме все вершины октанта приближаются к камере на расстояние $\frac{1}{10}\Delta z$. Это позволяет устранить мерцания во многих случаях. Преимущество этого подхода состоит в том, что расчет проводится не для каждого пикселя, а для всех вершин октанта один раз во время проверки видимости.

4. Сравнение производительности

Сравнивается время составления кадра при визуализации сцен с помощью двух алгоритмов, которые отличаются методами проверки видимости. В первом алгоритме используются аппаратные проверки видимости (АПВ). Во втором алгоритме, который предлагается в данной работе, используются программные проверки видимости (ППВ). Кроме того, приводятся результаты алгоритма отсечения объектов, не попадающих в область видимости камеры (Frustum Culling). Характеристики тестовой системы: Intel Core i7-7700 3600MHz 8192Kb L3, Intel HD Graphics 630, 16GB DDR4 2400MHz. Тест заключается в измерении времени составления кадра при движении камеры по сцене. Тестовые сцены (см. рис. 3–6):

- сцена 1: 10,827,713 треугольников, 71,961 объектов; примерно половину объема сцены занимают довольно большие объекты с маленьким количеством полигонов;
- сцена 2: 31,462,818 треугольников, 270,431 объектов;
- сцена 3: 11,536,541 треугольников, 109,991 объектов;
- сцена 4: 10,154,304 треугольников, 221,796 объектов; искусственная сцена, которая состоит из 36 зданий; в каждом здании имеется большая группа объектов, которая в большинстве случаев оказывается загороженной.

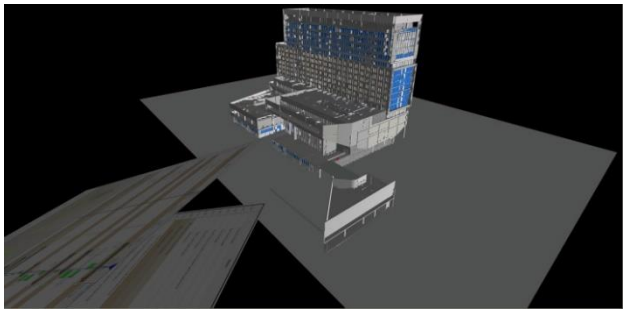


Рис. 3. Сцена 1 – архитектурная сцена из 10.8 миллионов треугольников
Fig. 3. Scene 1 – architectural scene that contains 10.8 million triangles

Рис. 4. Сцена 2 – архитектурная сцена из 31.5 миллионов треугольников
Fig. 4. Scene 2 – architectural scene that contains 31.5 million triangles

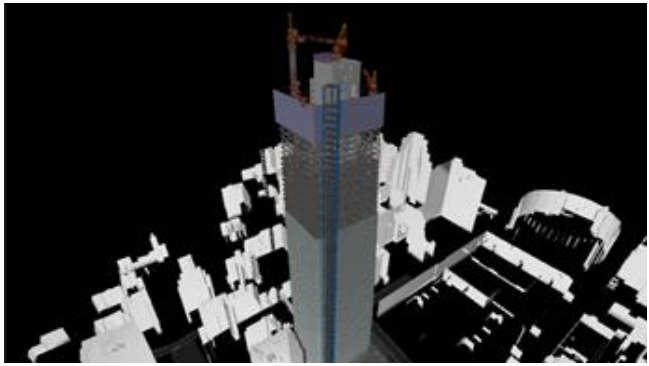


Рис. 5. Сцена 3 – архитектурная сцена из 11.5 миллионов треугольников
Fig. 5. Scene 3 – architectural scene that contains 11.5 million triangles.

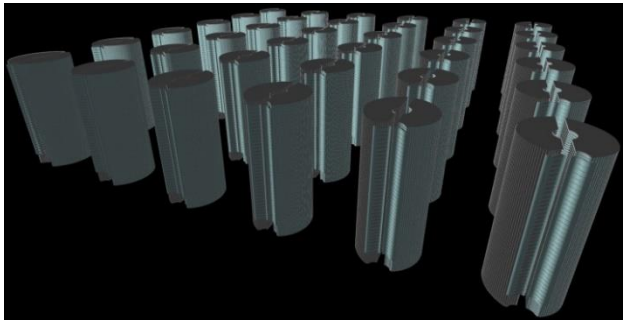


Рис. 6. Искусственная сцена со зданиями из 10.2 миллионов треугольников
Fig. 6. Artificial scene 3 with buildings having in total 10.2 million triangles

На рис. 7-10 и в табл. 3 показана производительность сравниваемых алгоритмов при проходе по тестовым сценам. Во всех тестах алгоритм ППВ показал выигрыш в производительности. Это происходит по двум причинам. Во-первых, выполнение проверок видимости в программном режиме происходит достаточно быстро, это позволило в алгоритме ППВ использовать окто-дерево с большей высотой, что, в свою очередь, сократило количество видимых объектов. Во-вторых, в ППВ посылается значительно меньше команд визуализации за счет выполнения проверок видимости на центральном процессоре.

Скачки на графиках связаны с изменением количества видимых объектов при движении камеры по сцене. Значительный объем сцены 1 занимают плоскости, на октанты которых посылается большое количество запросов видимости. За счет более быстрых проверок видимости ППВ оказывается эффективнее. В сцене 4 содержится большое количество объектов внутри зданий, и уменьшенный размер октантов позволяет сильно сократить количество визуализируемых объектов.

Хотя предложенный алгоритм дает существенный прирост производительности на рассмотренных сценах и данной конфигурации вычислительной системы, существуют и другие сценарии, в которых иерархические проверки видимости не имеют смысла. Например, когда в сцене имеется несколько тысяч объектов, которые всегда на виду, стоит использовать отсечение объектов, не попадающих в область видимости камеры, и не тратить лишние ресурсы на хранение иерархии и проверки видимости.

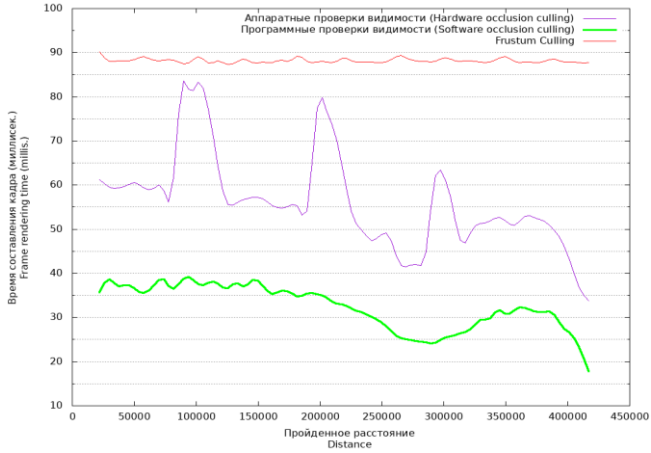


Рис. 7. Производительность во время прохода по сцене 1
Fig. 7. Performance during camera walkthrough of scene 1.

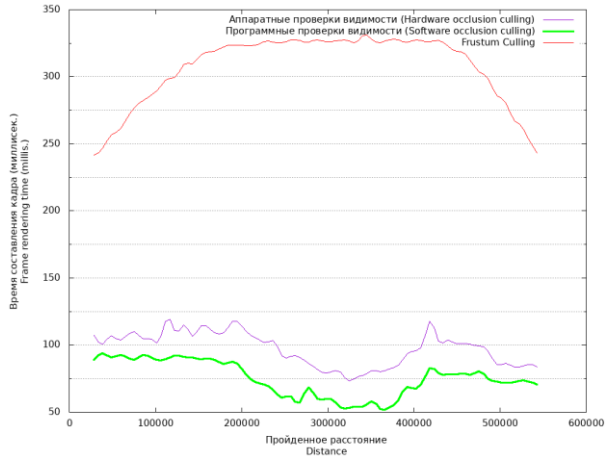


Рис. 8. Производительность во время прохода по сцене 2
Fig. 8. Performance during camera walkthrough of scene 2.

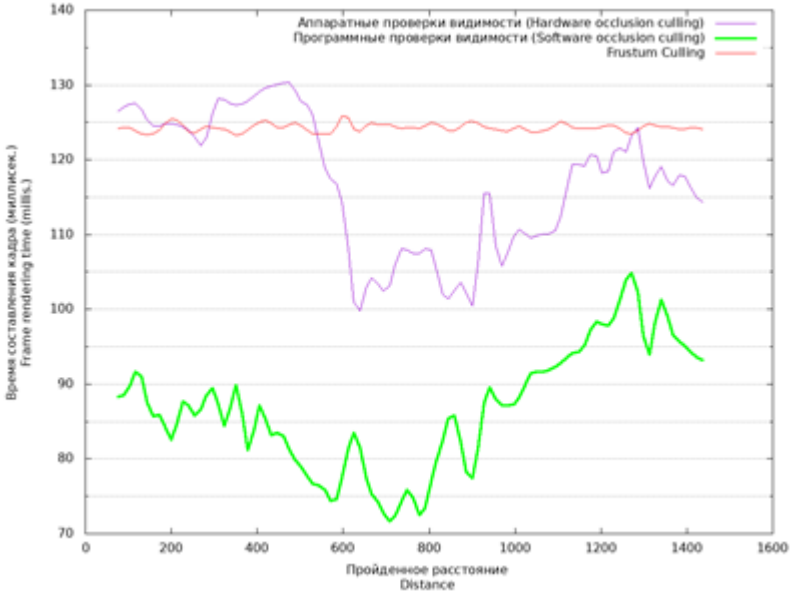


Рис. 9. Производительность во время прохода по сцене 3
Fig. 9. Performance during camera walkthrough of scene 3

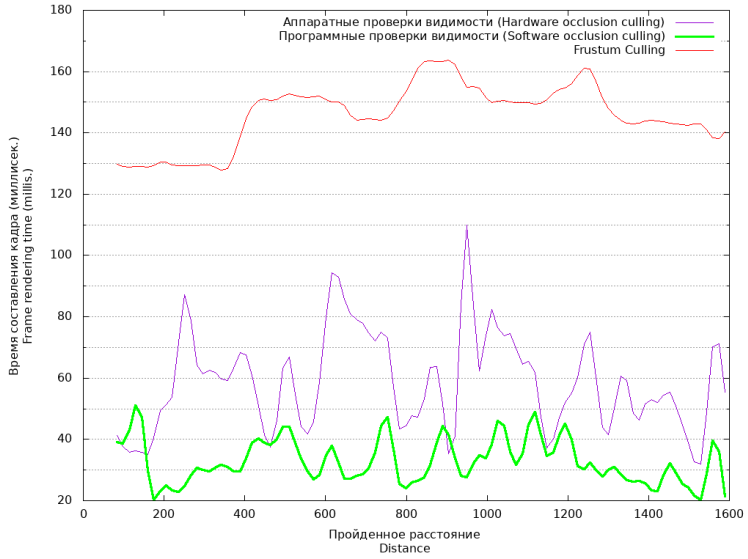


Рис. 10. Производительность во время прохода по сцене 4
Fig. 10. Performance during camera walkthrough of scene 4

Табл. 3. Среднее время составления кадра во время прохода камеры по сценам
Table 3. Average frame rendering time during scene walkthrough

Алгоритм	Сцена 1, мс	Сцена 2, мс	Сцена 3, мс	Сцена 4, мс
Frustum culling	88.2	305.0	124.3	145.2
АПВ	56.8	98.2	117.2	58.4
ППВ	33.5	77.2	86.9	34.5

5. Заключение

В работе предложен алгоритм удаления невидимых поверхностей на основе программных проверок видимости и окто-дерева. Описаны основные техники, которые позволяют ускорить проверки видимости на CPU. Предложена эвристика выбора уровня окто-дерева, который соответствует наиболее эффективным проверкам видимости и показана ее применимость на практике. Также предложена техника консервативного понижения глубины вершин октанта для устранения мерцаний. Предложенный алгоритм эффективнее справляется с визуализацией архитектурных сцен, чем алгоритм на основе аппаратных проверок видимости.

Список литературы

[1]. Bittner, J. and Wonka, P., 2003. Visibility in computer graphics. Environment and Planning B: Planning and Design, Vol. 30, No.5, pp.729–755.
[2]. Cohen-Or D. et al, 2003. A survey of visibility for walkthrough applications. IEEE Transactions on Visualization and Computer Graphic,Vol. 9, No. 3, pp. 412–431.
[3]. Greene, N. et al, 1993. Hierarchical Z-buffer visibility. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. ACM.
[4]. Teller, S., Sequin, C., 1991. Visibility preprocessing for interactive walkthroughs. Computer Graphics (Proceedings of SIGGRAPH 91), Vol. 25, No. 4, pp. 61–69.
[5]. Next Generation Occlusion Culling. http://www.gamasutra.com/view/feature/164660/sponsored_feature_next_generation_.php?print=1 (дата обращения 09.04.2018).
[6]. Greene, N., 1996. Hierarchical polygon tiling with coverage masks. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. pp. 65–74.
[7]. Zhang, H. et al, 1997. Visibility culling using hierarchical occlusion maps. Proceedings of the 24th annual conference on Computer graphics and interactive techniques. pp. 77–88.
[8]. Bittner, J. et al, 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. In Computer Graphics Forum, Vol. 23, No.3, pp. 615–624.
[9]. Chandrasekaran C. et al, 2013–2016. Software Occlusion Culling. https://software.intel.com/en-us/articles/ (дата обращения 09.04.2018).
[10]. Guthe, M. et al, 2006. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In Eurographics Symposium on Rendering. pp. 207–214.

[11]. Mattausch, O. et al, 2008. CHC++: Coherent Hierarchical Culling Revisited. EUROGRAPHICS, Vol. 27, No. 3.
[12]. GLAPI/glBeginQuery. https://www.opengl.org/wiki/GLAPI/glBeginQuery (дата обращения 09.04.2018).
[13]. Macdonald, J. D., Booth, K. S., 1990. Heuristics for ray tracing using space subdivision. Visual Computer, Vol. 6, No. 6, pp. 153–165.
[14]. Meissner, M. et al, 2001. Generation of Decomposition Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. In Vision, Modeling, and Visualization, Vol. 1, pp. 225–232.
[15]. Pharr M., Jakob W., Humphreys G. Physically based rendering: From theory to implementation. Morgan Kaufmann, 2016, 1233 p.
[16]. Morozov S., Semenov V., Tarlapan O., Zolotov V. (2018) Indexing of Hierarchically Organized Spatial-Temporal Data Using Dynamic Regular Octrees. In: Petrenko A., Voronkov A. (eds) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science, vol 10742, pp. 276–290.
[17]. Золотов В.А., Петрищев К.С., Семенов В.А. Исследование методов пространственного индексирования динамических сцен на основе регулярных октодеревьев. Программирование, 2016, № 6, стр. 59–66.
[18]. V.A. Semenov, K.A. Kazakov, V.A. Zolotov. Effective spatial reasoning in complex 4D modeling environments. eWork and eBusiness in Architecture, Engineering and Construction, eds. A.Mahdavi, B. Martens, R. Scherer, CRC Press, Taylor & Francis Group, London, UK, 2015, pp. 181–186.
[19]. Software Occlusion Culling Sample Application. https://github.com/GameTechDev/OcclusionCulling (дата обращения 09.04.2018).
[20]. Marschner, S. and Shirley, P., 2015. Fundamentals of computer graphics (3rd ed.). CRC Press, pp.45–49.
[21]. Gonakhchyan V. Comparison of hierarchies for occlusion culling based on occlusion queries. In Proceedings of the GraphiCon 2017 conference on Computer Graphics and Vision, pp. 32-36.

Occlusion culling algorithm based on software visibility checks

V.I. Gonakhchyan <pusheax@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Rendering of 3D scenes with big number of objects is computationally intensive. Occlusion culling methods are used to decrease the number of handled objects. We consider interactive occlusion culling methods that have spatial and time coherence. We propose algorithm to increase rendering performance by using occlusion checks implemented in software mode. We propose heuristic to determine hierarchy level that corresponds to the most efficient occlusion checking. The algorithm is compared with the algorithm based on hardware occlusion queries. Checking for occlusion on CPU avoids transmission overhead between CPU and GPU and as a result improves rendering performance of 3d scenes with big number of objects. Section 1 provides an overview of related work as well as general

purposes of given paper and its structure. Section 2 describes the basic formulas that are used in software rasterization and visibility checks. Section 3 describes the proposed algorithm for removing invisible surfaces. Section 4 presents the results of comparing the performance of the proposed algorithm and the algorithm based on hardware visibility requests. Section 5 summarizes the main conclusions.

Keywords: 3d rendering; occlusion query; octree.

DOI: 10.15514/ISPRAS-2018-30(2)-5

For citation: Gonakhchyan V.I. Occlusion culling algorithm based on software visibility checks. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue. 2, 2018, pp. 81-98 (in Russian). DOI: 10.15514/ISPRAS-2018-30(2)-5

References

[1]. Bittner, J. and Wonka, P., 2003. Visibility in computer graphics. Environment and Planning B: Planning and Design, Vol. 30, No.5, pp.729–755.

[2]. Cohen-Or D. et al, 2003. A survey of visibility for walkthrough applications. IEEE Transactions on Visualization and Computer Graphic,Vol. 9, No. 3, pp. 412–431.

[3]. Greene, N. et al, 1993. Hierarchical Z-buffer visibility. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. ACM.

[4]. Teller, S., Sequin, C., 1991. Visibility preprocessing for interactive walkthroughs. Computer Graphics (Proceedings of SIGGRAPH 91), Vol. 25, No. 4, pp. 61–69.

[5]. Next Generation Occlusion Culling. http://www.gamasutra.com/view/feature/164660/sponsored_feature_next_generation_.php?print=1 (accessed 09.04.2018).

[6]. Greene, N., 1996. Hierarchical polygon tiling with coverage masks. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. pp. 65–74.

[7]. Zhang, H. et al, 1997. Visibility culling using hierarchical occlusion maps. Proceedings of the 24th annual conference on Computer graphics and interactive techniques. pp. 77–88.

[8]. Bittner, J. et al, 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. In Computer Graphics Forum, Vol. 23, No.3, pp. 615–624.

[9]. Chandrasekaran C. et al, 2013–2016. Software Occlusion Culling. https://software.intel.com/en-us/articles/ (accessed 09.04.2018).

[10]. Guthe, M. et al, 2006. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In Eurographics Symposium on Rendering. pp. 207–214.

[11]. Mattausch, O. et al, 2008. CHC++: Coherent Hierarchical Culling Revisited. EUROGRAPHICS, Vol. 27, No. 3.

[12]. GLAPI/glBeginQuery. https://www.opengl.org/wiki/GLAPI/glBeginQuery (accessed 09.04.2018).

[13]. Macdonald, J. D., Booth, K. S., 1990. Heuristics for ray tracing using space subdivision. Visual Computer, Vol. 6, No. 6, pp. 153–165.

[14]. Meissner, M. et al, 2001. Generation of Decomposition Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. In Vision, Modeling, and Visualization, Vol. 1, pp. 225–232.

[15]. Pharr M., Jakob W., Humphreys G. Physically based rendering: From theory to implementation. Morgan Kaufmann, 2016, 1233 p.

[16]. Morozov S., Semenov V., Tarlapan O., Zolotov V. (2018) Indexing of Hierarchically Organized Spatial-Temporal Data Using Dynamic Regular Octrees. In: Petrenko A., Voronkov A. (eds) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science, vol 10742, pp. 276–290.

[17]. Zolotov V.A., Petrishchev K.S., Semenov V.A. Methods of spatial indexing of dynamic scenes based on regular octrees. Programming and Computer Software, vol. 42, No. 6, 2016, pp. 375–381. DOI: 10.1134/S0361768816060098

[18]. V.A. Semenov, K.A. Kazakov, V.A. Zolotov. Effective spatial reasoning in complex 4D modeling environments. // eWork and eBusiness in Architecture, Engineering and Construction, eds. A.Mahdavi, B. Martens, R. Scherer, CRC Press, Taylor & Francis Group, London, UK, 2015, pp. 181–186.

[19]. Software Occlusion Culling Sample Application. https://github.com/GameTechDev/OcclusionCulling (accessed 09.04.2018).

[20]. Marschner, S. and Shirley, P., 2015. Fundamentals of computer graphics (3rd ed.). CRC Press, pp.45–49.

[21]. Gonakhchyan V. Comparison of hierarchies for occlusion culling based on occlusion queries. In Proceedings of the GraphiCon 2017 conference on Computer Graphics and Vision, pp. 32-36.