ИНФОРМАТИКА

INFORMATICS

Вестник Сыктывкарского университета.
Серия 1: Математика. Механика. Информатика. 2023.
Выпуск 2 (47)
Bulletin of Syktyvkar University.
Series 1: Mathematics. Mechanics. Informatics. 2023; 2 (47)

Научная статья

УДК 539.3
https://doi.org/10.34130/1992-2752_2023_2_17

ОБ ОСОБЕННОСТЯХ ФИЛЬТРАЦИИ КОЛЛИЗИЙ В ФИЗИЧЕСКОМ ДВИЖКЕ ДЛЯ ТРЁХМЕРНЫХ ИГР

Вадим Андреевич Мельников
Сыктывкарский государственный университет
им. Питирима Сорокина, muller95@yandex.ru

Аннотация. В статье рассматриваются параллельный и последовательный подход к реализации фильтрации коллизий на основе сортировки массивов и проведены замеры производительности различных сортировок с различным числом потоков.

Ключевые слова: физика, коллизии, столкновения, фильтрация, AABB, сортировка

Для цитирования: Мельников В. А. Об особенностях фильтрации коллизий в физическом движке для трёхмерных игр // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика.* 2023. Вып. 2 (47). С. 17–28. https://doi.org/10.34130/1992-2752_2023_2_17

Article

About architectural features of collisions filtering in the physics engine for 3D games

Vadim A. Melnikov
Pitirim Sorokin Syktyvkar State University, muller95@yandex.ru

Abstract. The article discusses parallel and sequential approaches to the implementation of collision filtering based on array sorting and measures the performance of various sorts with different numbers of threads.

Keywords: physics, collisions, filtering, AABB, sorting

For citation: Melnikov V. A. About architectural features of collisions filtering in the physics engine for 3D games. *Vestnik Syktyvkarskogo universiteta. Seriya 1: Matematika. Mekhanika. Informatika* [Bulletin of Syktyvkar University, Series 1: Mathematics. Mechanics. Informatics], 2023, no 2 (47), pp. 17−28. https://doi.org/10.34130/1992-2752_2023_2_17

Введение Современные игры представляют собой сложную систему, состоящую из множества игровых объектов, работающих в реальном времени и обращающихся к различным движкам (от английского engine) — рендер, звук, физика, сеть. Графический движок (иногда движок рендеринга, рендерер, визуализатор) отвечает за отрисовку изображения и симуляцию освещения. На основе движка рендеринга строится движок пользовательского интерфейса [1; 2]. Аудиодвижок отвечает за проигрывание звука на устройстве, позиционирование звуковых источников в пространстве для проигрывания объёмного звука, генерацию процедурного звука. Физический движок симулирует взаимодействие твёрдых и деформируемых тел в динамике, динамики жидкости, а также контролирует выполнение ограничений, установленных на взаимное расположение тел [3].

Игровые объекты могут состоять из одного или нескольких компонентов, которые в реальном времени обновляют своё состояние в зависимости от происходящего в игре и взаимодействуют с различными системами для обновления игрового мира. Например, в логике пользовательского интерфейса в главном меню может быть заложена логика асинхронной загрузки последних данных игрока с сервера, а в этот момент игроку показывается прогресс загрузки данных. Таким образом,

в реальном времени сразу задействованы две системы — рендер для отображения интерфейса и сеть для получения данных. Второй пример — полёт снаряда в игровом пространстве. В таком случае задействуются рендер для отрисовки снаряда и физика для расчётов баллистики и столкновений; в каждом кадре пересчитывается физическое состояние мира и обновляется изображение на экране.

В работе [4] игры рассматриваются как системы, с которыми взаимодействует игрок. С такого ракурса дополнительную важность приобретает физический движок, поскольку он обеспечивает симуляцию в реальном времени, расширяющую и углубляющую возможность интерактивного взаимодействия. Например, симуляция автомобиля в гоночной игре или тренажёре, связанном с вождением любого транспорта. К тому же проработанность симуляции будет напрямую влиять и на геймплей, это может быть простое движение твёрдого тела в пространстве без учёта сложных взаимодействий, что подойдёт для аркадных гонок; или же в программе может быть реализована сложная симуляция контакта колёс с ландшафтом для реконструкции поведения автомобиля на сложном рельефе [5]. Пуля, выпущенная из автомата, может быть представлена лучом, пересекающимся с препятствиями, или же будет рассчитано движение по баллистической траектории с учетом сопротивления воздуха.

Одной из важных задач физического движка является регистрация и обработка столкновений между телами. Данную задачу можно разделить на несколько этапов:

1. Фильтрация и обнаружение коллизий, если они есть.
2. Разрешение коллизий.
3. Обновление скоростей сталкивающихся тел.

Целью представленной работы является изучения возможностей оптимизации и особенностей фильтрации коллизий с помощью сортировки массивов [6], а также оптимизация фильтрации с применением современных алгоритмов сортировки [7], включая рассмотрение применимости параллелизма к данной задаче.

Постановка задачи регистрации коллизий и фильтрации

Для проведения численных экспериментов будем рассматривать движение абсолютно твёрдых идеально гладких сфер диаметром 1 метр

и весом 1 килограмм. На старте сферы будут разгоняться стартовой силой в случайном направлении, и после этого сфера будет двигаться под действием только силы тяжести. Максимальное число частиц 8000.

Движение тел описывается принципом Даламбера

$$\sum_{i}^{n} F_i = ma, \tag{1}$$

где F_i — силы, действующие на частицу; m — масса частицы; a — ускорение.

А для того, чтобы сферы считались столкнувшимися, достаточно проверить, что расстояние (в качестве расстояния будем использовать евклидову норму) между центрами сфер строго меньше суммы их радиусов [8]:

$$||P_i - P_j||_2 < r_i + r_j, \tag{2}$$

где $P_i, P_j \in R^3$ — координаты частиц в пространстве, а $r_i, r_j \in R^1$ — радиусы частиц.

Если регистрировать столкновения без дополнительной фильтрации, то для 800 сфер получим $\frac{2}{8000} = 31996000$ проверок с вычислением глубины взаимопроникновения сфер. Вычисление взаимопроникновения использует евклидову норму, а наличие возведений в квадрат и вычисление квадратного корня будет негативно сказываться на производительности программы, при этом будет множество отрицательных проверок, хотя при диаметре мира 160 метров регистрируется всего $450-550$ столкновений при описанном эксперименте, т. е. всего 0.0017 % положительных срабатываний.

Таким образом, для физического движка необходим способ снижения числа отрицательных проверок. Первое, что можно сделать, – это построить для сфер ограничивающие параллелепипеды, выровненные по осям (Axis-aligned bounding box, в дальнейшем AABB) [6], и воспользоваться теоремой о том, что AABB пересекаются тогда и только тогда, когда пересекаются проекции AABB на все оси пространства [9]. Таким образом уже можно отфильтровать часть ложных проверок с вычислением расстояний. Данная теорема основана на основном свойстве AABB, следующем из названия, что грани ограничивающего параллелепипеда направлены вдоль осей координат и координаты углов параллелепипеда находятся в основной («мировой») системе координат,

а не в локальной системе геометрического объекта. А для двумерного случая будет ограничивающий прямоугольник со сторонами, выровненными по осям, что можно видеть на рисунке.

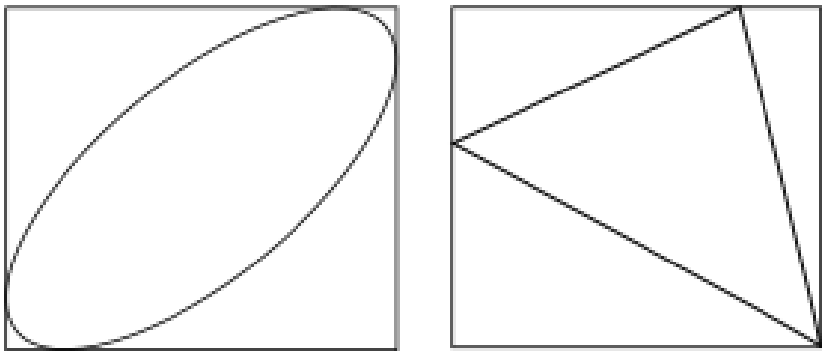

Рис. Пример AABB в 2D пространстве

Дальнейшая структуризация данных позволит ещё сильнее отфильтровать ложные проверки, поскольку если отсортировать AABB по одной из осей, например X, то при итерировании элементов по i и j ($j > i$), если левый край j-го элемента лежит правее, чем правый край i-го, можно прервать итерирование по j и перейти к следующему i.

Фильтрация коллизий с применением сортировки массива

Кроме сортировки массивов существует много способов фильтрации коллизий, например разбиение пространства с помощью октодерева. Недостаток использования деревьев заключается в сложности их организации, они не представляют собой непрерывный участок памяти и менее эффективны в кэшировании. А отсортированный массив линеен в памяти, и его обход линеен и упорядочен, что ведёт к более эффективному использованию кэша [10].

В структуру данных для сферы входят следующие компоненты:

- данные о предыдущих скорости и ускорении, так как интегрирование сил для вычисления движения сфер происходит по методу Верле [11];
- текущая позиция сферы;
- кешированный AABB.

Для представления чисел будем использовать числа с плавающей точкой с одинарной точностью (float), одно такое число занимает 4 байта [12], таким образом, только текущие координаты сфер будут занимать 12 байт. Сортировка таких больших структур будет замедляться большим числом действий по копированию элементов при упорядочивании. Можно хранить указатели на элементы и переставлять только их, но в таком варианте, если сферы создаются просто с помощью оператора *new*, нарушается линейность памяти. Поскольку память нужна и для хранения частицы и указателя, можно хранить сферы в массиве, а в дополнительном, который будет сортироваться, хранить указатель на сферу или индекс сферы. Индекс является более предпочтительным, поскольку размер индекса можно подобрать под максимальное число объектов, 8000 объектов можно разместить и в 16-битном беззнаковом числе. Таким способом получаются две линейные области памяти, в одной – структуры данных сфер, в которых обновляются данные, во второй – индексы, которые можно быстро переставлять местами.

Реализация замеров производительности проводилась на движке Godot 4 [13], а для сравнения эффективности метода использовалось несколько сортировок:

- stable_sort из стандартной библиотеки C++ [14];
- sort_custom из стандартной библиотеки Godot [15];
- array sort [7].

Большинство игр рендерится с частотой 30 или 60 кадров в секунду, т. е. для 30 кадров в секунду построение всего кадра не должно занимать более 33 миллисекунд. В работе рассматриваются замеры производительности только на центральном процессоре (CPU).

Особенность array sort, предложенного в статье [7], — частичная «предсортировка» и определение возрастающих последовательностей в массиве, что в дальнейшем приводит к ускорению сортировки, поскольку слияние выполняется для тегов, обозначающих начала возрастающих последовательностей. Также авторами предлагаются две схемы параллельной сортировки. На самом деле, схема с разделением массива на части и параллельной сортировкой частей может выполняться любой сортировкой, только последний шаг должен быть слиянием (как и в сортировке слиянием, просто не для двух последовательностей, а для n, где n — число ядер процессора) [17].

Результаты

Все замеры производительности проводились на процессоре Ryzen 3900X (12 физических ядер, 24 потока, 3.6 ГГц – базовая частота). Результаты отражают сумму времени за цикл при 30 итерациях разрешения коллизий. Время разрешения коллизий после фильтрации составляет 4.5–5.0 мс. Число коллизий на кадр для 8000 сфер 450–550 коллизий. В таблице приведены результаты замеров сортировок.

Таблица

Результаты измерений скорости сортировки для двух алгоритмов для различного числа потоков

	Array sort, мс	Stable sort, мс
1 поток	4.5–6.0	9.0–10.0
2 потока	6.0–7.0	9.5–10.5
4 потока	8.4–9.1	9.5–11.0
8 потоков	14.0–16.0	20.0–21.0
12 потоков	19.0–20.0	20.0–21.0
24 потока	33.0–36.0	36.0–38.0

Из-за особенности реализаций стандартной библиотеки Godot нельзя отсортировать просто область памяти в отличие от стандартной библиотеки C++. Поэтому для custom sort приводится только один поток: 14.7–16.0 мс. Исходя из двух остальных замеров можно сделать однозначный вывод о неприменимости распараллеливания для оптимизации сортировки в реальном времени.

С ростом числа потоков наблюдается заметное увеличение времени выполнения алгоритма, что свидетельствует о непригодности распараллеливания для выполнения фильтрации коллизий в реальном времени, поскольку обращение к очереди заданий и постановка задачи в неё происходят через обращение к функциям ядра, занимающим много времени. Просто запустить потоки и дождаться их без сортировки занимает 5 мс, что составляет почти 30 % бюджета кадра при частоте 60 кадров в секунду.

Следует отметить, что симуляция физики с использованием вычислительных шейдеров показывает хорошие результаты [16] и параллельная реализация алгоритмов сортировки возможна и для графического процессора (GPU)[1]. Для реализации на GPU достаточно передать массив и места, с которых начинается сортировка.

Применимость параллельных вычислений в статье [16] объясняется отсутствием необходимости синхронизировать в конце потоки и редким обращением к параллелизму, обновление положения частиц происходит всего один раз за кадр. Также параллельные сортировки в конце сводятся фактически к сортировке вставками для слияния результатов всех потоков. И чем больше потоков, тем больше будет сходство с сортировкой вставками.

Возможно параллелизация показала бы лучший результат на построении деревьев, например октодерева [6], так как 8 начальных частей октодерева строятся независимо друг от друга и не требуется дополнительных действий по слиянию после выполнения алгоритма.

Заключение

По результатам работы можно сделать вывод о том, что алгоритмы распараллеливания для сортировок в реальном времени не подходят, поскольку цена постановки задачи на выполнение в многопоточную очередь или создания потока с последующим слиянием результатов выше, чем сортировка всех объектов сразу. О нелинейности оптимизации и ухудшении результатов оптимизации также говорят и в статье [7]. Но применение современного метода сортировки array sort показало его значительное превосходство в скорости работы по сравнению со стандартными методами.

Список источников

1. **Мельников В. А.** Процесс разработки движка для 2D-игр и интерфейсов Sad Lion Engine // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика*. 2019. Вып. 4 (33). С. 21—37.

[1]GPU в персональных компьютерах выделен на отдельную плату и имеет собственную дополнительную память, в мобильных устройствах имеет общий чип памяти с CPU.

2. **Melnikov V. A., Yermolenko A. V.** Development of XML-based Markup Language // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика*. 2022. Вып. 1 (42). С. 61–73.

3. **Gregory J.** Game engine architecture, 3rd edition. Boca Raton: CRC Press, 2019. 1200 p.

4. **Зубек Р.** Элементы геймдизайна. Как создавать игры, от которых невозможно оторваться. М.: Бомбора, 2022. 272 с.

5. **Страшнов Е. В., Торгашев М. А.** Алгоритмы определения коллизий аппроксимирующих цилиндров с моделью рельефа местности // *International Journal of Open Information Technologies*. 2020. Vol. 8. No 7. С. 40–49.

6. **Ericson C.** Real-time collision detection. Amsterdam, Boston, Heidelberg, London, New York, Oxford, Paris, San Diego, San Francisco, Singapore, Sydney, Tokyo: Morgan Kaufman Publishers, 2005. 593 p.

7. **Huang X., Liu Z., Li J.** Array sort: an adaptive sorting algorithm on multi-thread // *The Journal of Engineering*. 10.1049/joe.2018.5154. 2019. Pp. 3455–3459.

8. **Millington I.** Game physics engine development. Amsterdam, Boston, Heidelberg, London, New York, Oxford, Paris, San Diego, San Francisco, Singapore, Sydney, Tokyo: Morgan Kaufman Publishers, 2007. 456 p.

9. **Huynh J.** Separating axis theorem for oriented bounding boxes [Электронный ресурс]. URL: http://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf (дата обращения: 30.05.2023).

10. **Bhagrav N.** Cache-friendly code [Электронный ресурс] // Baeldung. URL: https://www.baeldung.com/cs/cache-friendly-code (дата обращения: 30.05.2023).

11. **House D. H., Keyser J. C.** Foundations of physically based modelling and animation. Boca Raton: CRC Press, 2017. 382 p.

12. Fundamental types [Электронный ресурс] // C++ reference. URL: https://en.cppreference.com/w/cpp/language/types (дата обращения 30.05.2023).

13. Godot [Электронный ресурс] // Godot. URL: https://godotengine.org/ (дата обращения 30.05.2023).

14. std::stable_sort [Электронный ресурс] // C++ reference. URL: https://en.cppreference.com/w/cpp/algorithm/stable_sort (дата обращения 30.05.2023).

15. Array [Электронный ресурс] // Godot docs. URL: https://docs.godotengine.org/en/stable/classes/class_array.html (дата обращения 30.05.2023).

16. **Озерицкий А. В.** Моделирование методом частиц на GPU с использованием языка GLSL // Выч. мет. программирование. 2023. Вып. 1 (24). С. 37–54.

17. **Кнут Д.** Искусство программирования. Том 3. Сортировка и поиск. М.: Вильямс. 2001. 824 с.

References

1. **Melnikov V. A.** Development Process of game engine core for 2D-games and interfaces Sad Lion Engine. *Vestnik Syktyvkarskogo universiteta. Seriya 1: Matematika. Mekhanika. Informatika* [Bulletin of Syktyvkar University. Series 1: Mathematics. Mechanics. Informatics], 2019, 4 (33), pp. 21–37. (In Russ.)

2. **Melnikov V. A., Yermolenko A. V.** Development of XML-based Markup Language. *Vestnik Syktyvkarskogo universiteta. Seriya 1: Matematika. Mekhanika. Informatika* [Bulletin of Syktyvkar University. Series 1: Mathematics. Mechanics. Informatics], 2022, 1 (42), pp. 61–73.

3. **Gregory J.** *Game engine architecture, 3rd edition.* Boca Raton: CRC Press. 2019, 1200 p.

4. **Zubek P.** *Elementy geymdizayna. Kak sozdavat' igry, ot kotorykh nevozmozhno otorvat'sya* [Elements of game design. How to create

games from which it is impossible to break away]. M.: Bombora, 2022. 272 p. (In Russ.)

5. **Strashnov E. V., Torgrashev M. A.** Collision detection algorithms of bounding cylinders with terrain model. *International Journal of Open Information Technologies*. 2020, vol. 8, no 7, pp. 40–49. (In Russ.)

6. **Ericson C.** *Real-time collision detection*. Amsterdam, Boston, Heidelberg, London, New York, Oxford, Paris, San Diego, San Francisco, Singapore, Sydney, Tokyo: Morgan Kaufman Publishers, 2005. 593 p.

7. **Huang X., Liu Z., Li J.** Array sort: an adaptive sorting algorithm on multi-thread. *The Journal of Engineering*. 10.1049/joe.2018.5154. 2019, pp. 3455–3459.

8. **Millington I.** *Game physics engine development*. Amsterdam, Boston, Heidelberg, London, New York, Oxford, Paris, San Diego, San Francisco, Singapore, Sydney, Tokyo: Morgan Kaufman Publishers, 2007. 456 p.

9. **Huynh J.** Separating axis theorem for oriented bounding boxes [Electronic resource]. Available at: http://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf (accessed: 30.05.2023).

10. **Bhagrav N.** Cache-friendly code [Electronic resource].*Baeldung*. Available at: https://www.baeldung.com/cs/cache-friendly-code (accessed: 30.05.2023).

11. **House D. H., Keyser J. C.***Foundations of physically based modelling and animation*. Boca Raton: CRC Press, 2017. 382 p.

12. Fundamental types [Electronic resource]. *C++ reference*. Available at: https://en.cppreference.com/w/cpp/language/types (accessed: 30.05.2023).

13. Godot [Electronic resource]. *Godot*. Available at: https://godotengine.org/ (accessed: 30.05.2023).

14. std::stable_sort [Electronic resource]. *C++ reference*. Available at: https://en.cppreference.com/w/cpp/algorithm/stable_sort (accessed: 30.05.2023).

15. Array [Electronic resource]. *Godot docs.* Available at: https://docs.godotengine.org/en/stable/classes/class_array.html (accessed: 30.05.2023).

16. **Ozeritskiy A. V.** Computational simulation using particles on GPU and GLSL language. *Vych. met. programmirovaniye*[Numerical Methods and Programming]. 2023, issue 1 (24), pp. 37–54. (In Russ.)

17. **Knuth D.** *Iskusstvo programmirovaniya. T. 3. Sortirovka i poisk.* [The art of computer programming. Vol. 3. Sorting and searching]. M.: Vilyams, 2001. 824 p.

Сведения об авторе / Information about author
Мельников Вадим Андреевич / Vadim A. Melnikov
аспирант / postgraduate student
Сыктывкарский государственный университет им. Питирима Сорокина / Pitirim Sorokin Syktyvkar State University
167001, Россия, г. Сыктывкар, Октябрьский пр., 55 / 167001, Russia, Syktyvkar, Oktyabrsky Ave., 55

Статья поступила в редакцию / The article was submitted 01.06.2023
Одобрено после рецензирования / Approved after reviewing 06.06.2023
Принято к публикации / Accepted for publication 08.06.2023