Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ВЫСШАЯ ШКОЛА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

Направление: 09.04.04 Программная инженерия

Профиль: Разработка программно-информационных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Быстрые булевы операции над полигональной сеткой

Студент 2 курса
группы 11-712
«___»_____2019 г. Гималтдинов Р.А.

Научный руководитель
Доктор физ.-мат. наук,
профессор кафедры
программной инженерии
«___»_____2019 г. Елизаров А.М.

Консультант
старший преподаватель
кафедры программной инженерии
«___»_____2019 г. Кугуракова В. В.

Директор Высшей школы ИТИС
«___»_____2019 г. Хасьянов А. Ф.

Казань – 2019 г.

Оглавление

Введение

В сфере трёхмерной компьютерной графики при взаимодействии в реальном времени существует проблема изменения полигональных моделей при их соприкосновении с другими полигональными объектами. Как, например, при использовании симуляции взаимодействия ножа и органических тканей, ткани должны менять свою топологическую структуру при касании ножа, что необходимо при создании хирургических симуляторов [1; 2]. Так как в данный момент мало исследований посвящённых обработке взаимодействий объектов в реальном времени, приводящих к изменению их топологии, эта проблема является актуальной.

При условии создания универсального инструмента, модифицирующего объекты логическими операциями в реальном времени, данная проблема будет частично решена.

В данный момент большинство разработок булевых операций для полигональных сеток входят в состав программных обеспечений для 3D-моделирования и решают проблему создания моделей на этапе разработки какого-либо продукта [3; 4]. Проблемы стабильности и точности алгоритмов логических операций над полигональными сетками в таких случаях решаются предоставлением дизайнеру нескольких способов обработки, тем самым пользователь выбирает соотношение между качеством и скоростью выполнения данных действий.

Данный вариант реализован в Blender – открытом программном обеспечении для создания трёхмерной компьютерной графики, где имеется два набора алгоритмов, используемых для вычисления логических операций. Carve использует внешнюю библиотеку Carve, а BMesh использует встроенную библиотеку Blender и даёт лучшие результаты [4].

Для пользователей различных симуляторов и игр, не владеющих базовыми навыками 3D-моделирования, такие процессы могут вызывать

дополнительные сложности. Так же в таких продуктах изменение полигональных сеток может происходить без участия пользователя, предоставляя лишь результаты вычислений. В таком случае важно сохранение приложением свойств системы реального времени. Так же точность алгоритмов может держаться в пределах малоотличимых человеческим глазом.

В игровых движках возможно создание множества типа продуктов, таких как симуляторы различных сфер деятельности человека, игры любых жанров и видов, демонстрации интерьеров или исторических событий и предметов, визуальные материалы: видео, фильмы, мультфильмы [5]. Большинство объектов в игровых движках представляется в виде полигональных сеток. Возможность обрабатывать логические операции над ними в готовых проектах, созданных в такой среде, не теряя свойств системы реального времени увеличило бы реалистичность создаваемых разработчиками виртуальных сцен.

Объектом исследования является реализация инструмента, позволяющая производить булевы операции в реальном времени над двумя полигональными объектами. Предметами исследования являются методы булевых операций, способные производить вычисления в реальном времени.

Целью исследования является создание библиотеки на языке C# в игровом движке Unity для изменения топологической структуры игровых объектов путём применения логических операций.

Для реализации цели были поставлены следующие задачи:

- оценить существующие методы булевых операций над 3D объектами;
- выбрать наиболее подходящий для визуализации в реальном времени метод логических операций над полигональной сеткой;

- задать такую временную характеристики ПО как дедлайн для обеспечения работы в реальном времени;
- разработать алгоритм с использованием найденного метода;
- создать библиотеку для игрового движка Unity с помощью языка программирования C#;
- протестировать реализацию и дать оценку полученному результату.

1 Обзор предметной области

1.1 Обзор литературных источников

Многие задачи обработки геометрии высокого уровня основаны на низкоуровневых операциях построения геометрии твердого тела. Некоторые методы бывают быстрыми, но не позволяют получить замкнутый вывод без самопересечения. Другие методы являются надежными, но на входе накладывают чрезмерно строгие допущения, например, нет полых полостей или самопересечений. В статье [6] предлагается систематический подход, основанный на двух этапах. Двухэтапный метод не делает общих предположений о положении. Метод является вариативным, работает с любым количеством входных сеток. Авторы демонстрируют превосходную эффективность и надежность метода на наборе данных из 10 000 полигональных моделей из онлайн-хранилища.

Проект с открытым исходным кодом CGAL обеспечивает легкий доступ к эффективным и надежным геометрическим алгоритмам в форме библиотеки C++, предлагая геометрические структуры данных и алгоритмы, которые являются эффективными, надежными, простыми в использовании и легко интегрируются в существующее программное обеспечение [7]. Использование стандартных библиотек повышает производительность, поскольку позволяет разработчикам программного обеспечения сосредоточиться на прикладном уровне. Документация [8] предназначена для разработчиков программного обеспечения с геометрическими потребностями, в ней продемонстрированы возможности выбора и использования алгоритмов и структур данных, предоставляемых CGAL.

В статье [9] был предложен новый метод логических операций для триангулированных мешей, состоящий из 3 подэтапов: тэтрализация Делоне входных данных; восстановление утраченных граничных вершин, вставкой точек Штейнера; подготовка выходных данных. В отличие от тех

существующих алгоритмов, которые просто поддерживают соответствующую сетку поверхности, данный алгоритм поддерживает граничную сетку объема одновременно с вычислением пересечений поверхности. Эта объемная сетка не только обеспечивает фоновую структуру, помогая повысить эффективность вычислений пересечений, но и позволяет разработать набор эффективных и надежных процедур заполнения типа потока для извлечения булевых выходных данных.

Статья решает проблему генерации меша при логических операциях над триангулированными гранями. Работа вносит новый подход к решению задачи в виде алгоритмов для повышения надёжности при использовании тетраэдризация Делоне и вставки точек Штейнера.

Был реализован алгоритм, однако надёжность метода имеет место для дальнейшего развития. Так при тесте операции над идентичными кубами при сдвиге одного из них на расстояние 1.0e-8 и меньше метод не мог выдавать правильные выходные данные, тогда как CGAL переставал работать при сдвиге 1.0e-15 и меньше.

Методы, которые напрямую используют сетки для вычисления логических операций, последовательно учитывают пересечения между двумя гранями без учета компланарных столкновений. Таким образом, они либо мешают входным сеткам, когда сталкивающиеся грани компланарны, либо просто игнорируют этот вид столкновений. В [10] предлагается надежный, точный и простой метод для управления булевыми операциями между сталкивающимися оболочками без преобразования и использования поверхностного подхода. Предлагаемый метод состоит из трех этапов:

1. вычисление пересечений входных оболочек как для некомпланарных, так и для компланарных столкновений;
2. разложение всей новой сетки на ее компоненты многообразия;
3. сохранение только компонентов, связанных с запрошенной операцией (объединение или пересечение).

Операции вычитания рассматриваются путем изменения ориентации поверхности вычитаемой оболочки с использованием операции пересечения. Выходные данные сохраняют точную геометрию входной сетки при добавлении вершин для пересекающихся граней столкновения. По сравнению с существующими методами, которые используют сетку напрямую, основным преимуществом предлагаемого метода является то, что он обрабатывает компланарные столкновения без геометрической модификации, что позволяет избежать создания множества маленьких оболочек, когда два объекта разделяют одну и ту же часть поверхности. По сравнению с методами, использующими объемное представление, предлагаемый метод является более быстрым.

В статье [11] авторы разработали алгоритм под названием Re2l, подходящий для вычислений булевых операций над многоугольниками с дугами. Метод, по утверждению авторов, прост в реализации, без потери эффективности. Превосходство предложенного алгоритма также подтверждается эмпирическим исследованием.

В исследовании [12] разработан эффективный подход для выполнения булевой операции для триангулированных сеток, представленных B-rep. Этот подход, по утверждениям авторов, намного быстрее и надежнее многих существующих методов. Техника Octree [13] адаптирована для облегчения разделения общего пространства двух ячеек с целью сокращения времени построения и определения пересечения Octree. Затем анализируются арифметические ошибки с плавающей точкой и особенности пересечений, чтобы гарантировать уникальное пересечение между отрезком и гранью и непрерывность пересечений. Наконец, предлагается новый метод, основанный на пересекающихся треугольниках, для создания необходимых моделей, основанных на типе булевых операций. Некоторые экспериментальные результаты и сравнения с другими методами представлены, чтобы доказать, что предложенный метод является быстрым и надежным.

В статье [14] описан арифметический алгоритм с плавающей точкой, предназначенный для решения обычных булевых операций (пересечение, объединение и разность) на произвольных многоугольных и многогранных сетках. Этот метод не аппроксимирует входные данные, которые могут быть как объемными сетками, так и поверхностными сетками или по одной из каждой. Он обеспечивает точные полигональные сетки при выходе. Основная идея состоит в том, чтобы рассматривать любую конфигурацию как многоугольное облако. Сначала полигоны триангулируются, пересечения решаются, затем клетки реконструируются из облака конформных треугольников и, наконец, их треугольные грани воссоединяются в полигоны. Этот подход обеспечивает большую гибкость в отношении допустимых топологий: обрабатываются неплоские грани, вогнутые грани или ячейки и некоторая немногообразность. Алгоритм подробно описан и показаны некоторые результаты.

В [15] преодолеваются проблемы устойчивости, расширяется класс входных/выходных сеток, предоставляют точный и надежный подход, действующий на общих сетках, которые должны быть только замкнутыми и ориентируемыми. Метод основан на нескольких геометрических и топологических предикатах, которые позволяют обрабатывать все случаи ввода/вывода, рассматриваемые как вырожденные в существующих решениях. Эксперименты авторов показали, что более общий подход также более надежен и эффективен, чем реализация Maya (в 3 раза), и Nef (в 5 раз), входящий в CGAL. В исследовании также представлен полный тест, предназначенный для проверки булевых алгоритмов в соответствующих и сложных сценариях.

Для применения нескольких входных многогранников и произвольной логической операции операция разлагается по двоичному дереву CSG, причем каждый узел обрабатывается отдельно. Для больших деревьев это является как подверженным ошибкам из-за промежуточной геометрии и накопления

ошибок, так и неэффективным, потому что каждый узел приводит к определенным издержкам. В [16] вводится принципиально новый подход к оценке многогранной CSG, касающийся общего случая N-многогранника. Предлагается вершинно-ориентированный взгляд на проблему, который упрощает алгоритм вычисления результирующих геометрических вкладов и значительно облегчает его пространственную декомпозицию. Метод не только повышает надежность алгоритма, но и дает ему фундаментальное преимущество в скорости, поскольку сложность вывода зависит от размера выходного меша, а не от размера ввода. В дополнение к распараллеливанию этот алгоритм обеспечивает хорошую производительность, ускоряя процесс обработки, при логических операциях над двумя или несколькими десятками многогранников.

В [17] представляется новый метод для выполнения логических операций на моделях, представленных в виде треугольных сеток. В отличие от существующих методов, которые рассматривают сетки как трехмерные многогранники и пытаются разделить грани на точных кривых пересечения, авторы рассматривают сетки как адаптивные поверхности, которые могут быть произвольно уточнены. Вместо того чтобы рассчитывать точные пересечения граней, данный подход уточняет входные сетки в областях пересечения, а затем отбрасывает пересекающиеся треугольники и заполняет получающиеся отверстия высококачественными треугольниками. Исходные кривые пересечения аппроксимируются с точностью, определяемой пользователем, и метод может идентифицировать и сохранять складки и острые элементы. Преимущества метода включают способность обменивать скорость на точность, поддержку открытых сеток и способность включать допуски для обработки случаев, когда большое количество граней слегка взаимопроникает или почти совпадает.

В статье [18] представлен полный теоретический и экспериментальный анализ последовательного алгоритма для теста точка-полигон, который

требует меньше времени выполнения, чем предыдущие алгоритмы, и может обрабатывать все вырожденные случаи. Последовательный алгоритм может быстро определить, находится ли точка внутри или снаружи многоугольника, и точно определить контуры входного многоугольника. Он описывает все вырожденные случаи и одновременно предоставляет соответствующее решение для каждого вырожденного случая, чтобы обеспечить стабильность и надежность. Алгоритмы применимы к любому многоугольнику, который может быть самопересекающимся или с отверстиями. Так же нет необходимости предварительно сортировать вершины по часовой стрелке или против часовой стрелки. Следовательно, они обрабатывают все ребра по одному в любом порядке для входных полигонов. Это позволяет легко реализовать параллельную реализацию каждого алгоритма. Также доказаны теоремы, гарантирующие корректность алгоритмов. Но методы представлены для двумерного пространства.

В статье [19] предлагается система, основанная на дереве BSP, которая, как утверждают авторы, в 16–28 раз быстрее в выполнении итеративных вычислений, чем система на основе Nef Polyhedra в CGAL, что являлось на момент выхода работы лучшей практикой в устойчивых булевых операциях, и в то же время она работает в два раза медленнее, чем инструмент, представленный Maya. Использование логического алгоритма на основе BSP-дерева позволяет явно обрабатывать все геометрические вырождения, не рассматривая большое количество случаев.

Так как алгоритм, основанный на BSP, легкомодифируемый и принимает во внимание множество случаев логических операций над полигональными объектами, поддерживая при этом высокую скорость вычисления, данная работа будет брать за основу статью [19].

1.2 Двоичное разбиение пространства

Для обработки трёхмерных объектов, воспользуемся двоичным разбиением пространства (BSP). Таким образом данные о полигональных объектах будут структурированы в виде двоичных деревьев.

При использовании в компьютерной графике со сценами, состоящими из плоских многоугольников, плоскости разбиения часто выбираются так, чтобы они совпадали с плоскостями, определенными многоугольниками в сцене [20].

Грани полигонального объекта совпадают с плоскостью, разделяющей пространство на две области, полупространства. Таким образом, полигональные модели в BSP-деревьях задаются в виде совокупности полупространств.

Недостатком двоичного разделения пространства является то, что генерация BSP-дерева может занимать много времени [21]. Как правило, поэтому она выполняется один раз, как этап предварительного расчета, перед рендерингом или другими операциями в реальном времени на сцене.

Данный вид представления данных о трёхмерных объектах уже используются в компьютерной графике, как средство визуализации.

1.2.1 Выбор разбивающей плоскости

Для построения BSP-дерева на каждом этапе необходим выбор плоскости для узла получаемого графа, проходящей через один или более полигонов заданной трёхмерной модели. Порядок выбора, не влияет на качество обработки логических операций, однако сказывается на скорости генерации и обработки логических операций.

Увеличение скорости можно достичь при наиболее сбалансированном дереве, когда плоскость выбирается так, что число полигонов, находящихся с разных её сторон, приблизительно равно [22]. Однако стоит заметить, что

сбалансировать BSP-дерево выпуклого многогранника невозможно, так как любая плоскость, проходящая через одну из граней будет разделять пространство на то, где есть оставшиеся необработанные полигоны, и на то, где полигоны отсутствуют. Так например для куба возможно получить дерево, как представлено на рисунке 1.

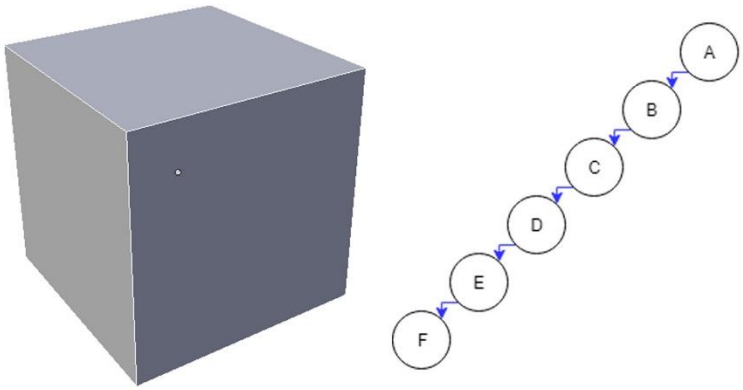

Рисунок 1 – BSP-дерево, сгенерированное для куба

1.3 Инструменты и технологии

1.3.1 Игровой движок Unity

Unity является средой разработки видеоигр. Данный игровой движок можно использовать для создания трехмерных и двумерных проектов с возможностью реализации дополненной или виртуальной реальности. Достоинствами данной среды являются:

- поддержка более 25 платформ [23];
- быстрая модифицируемость среды разработки [24];
- универсальность скриптов для создаваемого проекта и среды разработки [25];
- популярность игрового движка, что способствует созданию универсальных инструментов и модулей [26];
- большое количество поддерживаемых библиотек;
- поддержка языка программирования C#;

- подробная документация на официальном сайте [27];
- возможность создания и редактирования мешей.

В данной работе воспользуемся версией Unity 2018.3.0f2 Personal, новейшей на момент начала разработки требуемой системы. Unity Personal является бесплатной версией Unity, не включающая в себя дополнительную поддержку, обучение и услуги, имеющая ряд ограничений.

1.3.2 Интегрированная среда разработки Visual Studio

Microsoft Visual Studio – это полнофункциональная, расширяемая и интегрированная среда разработки для создания современных приложений [28]. Достоинствами данной среды являются:

- удобство редактирования кода;
- легкая работа с документацией;
- IntelliSense [29];
- легкость подключения дополнительных библиотек;
- большое количество поддерживаемых плагинов;
- возможность подключения к среде разработки Unity и поддержка проектов, созданных в ней;
- возможность отладки при использовании Play mode в Unity;
- Unity Project Explorer.

Разработаем необходимую библиотеку для игрового движка Unity в Visual Studio Community 2017 версии 15.7.4, выпущенной в феврале 2018 года [30].

2 Реализация инструментария для выполнения булевых операций

2.1 Определение возможностей разрабатываемого проекта

Перед разработкой проекта, способного предоставлять требуемый функционал, определим временные рамки, для обработки полигональных моделей в игровом движке Unity. Так как игры, симуляторы и другие проекты, разрабатываемые в игровых движках, являются системами мягкого реального времени [31], то частота выдаваемых прорисовок кадра не должна падать ниже тридцати кадров в секунду.

Для работы в таких приложениях вычисления не должны превышать показателя в 33 миллисекунды, так как при меньшей скорости, разрабатываемая система будет заметно отставать от взаимодействия пользователя с объектами. Из этого следует, что такая временная характеристика, как дедлайн должна быть равна 33 миллисекундам для нашего вычисления логических операций над полигональными объектами.

2.2 Необходимые компоненты игрового движка Unity и их особенности

Рассмотрим классы и методы, связанные с обработкой полигональных моделей и игровых объектов, в пространстве имён UnityEngine.

GameObject – базовый класс для всех сущностей в Unity Scenes. Данный класс будет использоваться для получение игровых объектов в сцене.

GameObject.GetComponent() – возвращает компонент типа Type, если к игровому объекту он прикреплен, и null, если нет. GetComponent является основным способом доступа к другим компонентам. С помощью этой функции есть возможность получить доступ как к встроенным компонентам, так и к сценариям. Данный метод будем использовать для получения таких компонентов как MeshFilter, Renderer и т.д.

Object.Destroy(); – удаляет игровой объект, компонент или ассет. Объект obj будет уничтожен сейчас или если время будет указано через t секунд.

Фактическое уничтожение объекта всегда задерживается до окончания текущего цикла обновления, но всегда будет выполняться перед рендерингом.

Object.Instantiate – клонирует объект оригинал и возвращает клон. Данные методы необходимы в случае замены исходной сетки получившейся.

Mesh – класс, который позволяет создавать или изменять полигональные сетки из скриптов. Сетки содержат вершины и несколько массивов треугольников. Массивы треугольников – это индексы в массивах вершин; три индекса для каждого треугольника. Для каждой вершины может быть нормаль, две координаты текстуры, цвет и касательная. Они не являются обязательными и могут быть удалены по желанию. Вся информация о вершинах хранится в отдельных массивах одинакового размера, поэтому, если сетка имеет 10 вершин, также будут массивы из 10 элементов для нормалей и других атрибутов. Существуют 3 варианта изменения меша имеющихся объектов [32]:

1. Создание сетки с нуля: всегда следует делать в следующем порядке:
    a. назначить вершины;
    b. назначить треугольники.
2. Изменение атрибутов вершин каждого экземпляра:
    a. получить вершины;
    b. изменить их;
    c. назначить их обратно в сетку.
3. Постоянно менять сетку треугольников и вершин:
    a. вызвать Clear, чтобы начать все заново;
    b. назначить вершины и другие атрибуты;
    c. назначить индексы треугольника.

MeshFilter – класс для получения полигональной модели из компонента mesh filter.

MeshFilter.mesh – возвращает созданный экземпляр Mesh, назначенный mesh filter. Если mesh filter не назначен сетке, будет создана и назначена новая сетка. Если сетка уже назначена mesh filter, то первый запрос свойства сетки создаст его дубликат, и эта копия будет возвращена. Дальнейшие запросы свойства сетки вернут этот дублированный экземпляр сетки. Как только запрашивается свойство сетки, ссылка на исходную общую сетку теряется, а свойство MeshFilter.sharedMesh становится псевдонимом для сетки. Если необходимо избежать этого автоматического дублирования сетки, следует использовать вместо этого MeshFilter.sharedMesh, который в случае изменения заменит меш у всех импортированных файлов.

Используя свойство сетки, возможно изменить сетку только для одного объекта. Другие объекты, которые использовали ту же сетку, не будут изменены.

При манипуляциях с мешем необходимо учитывать уничтожение автоматически созданной сетки при разрушении игрового объекта. Resources.UnloadUnusedAssets также уничтожает сетку, но обычно он вызывается только при загрузке нового уровня.

2.3 Структура создаваемой библиотеки

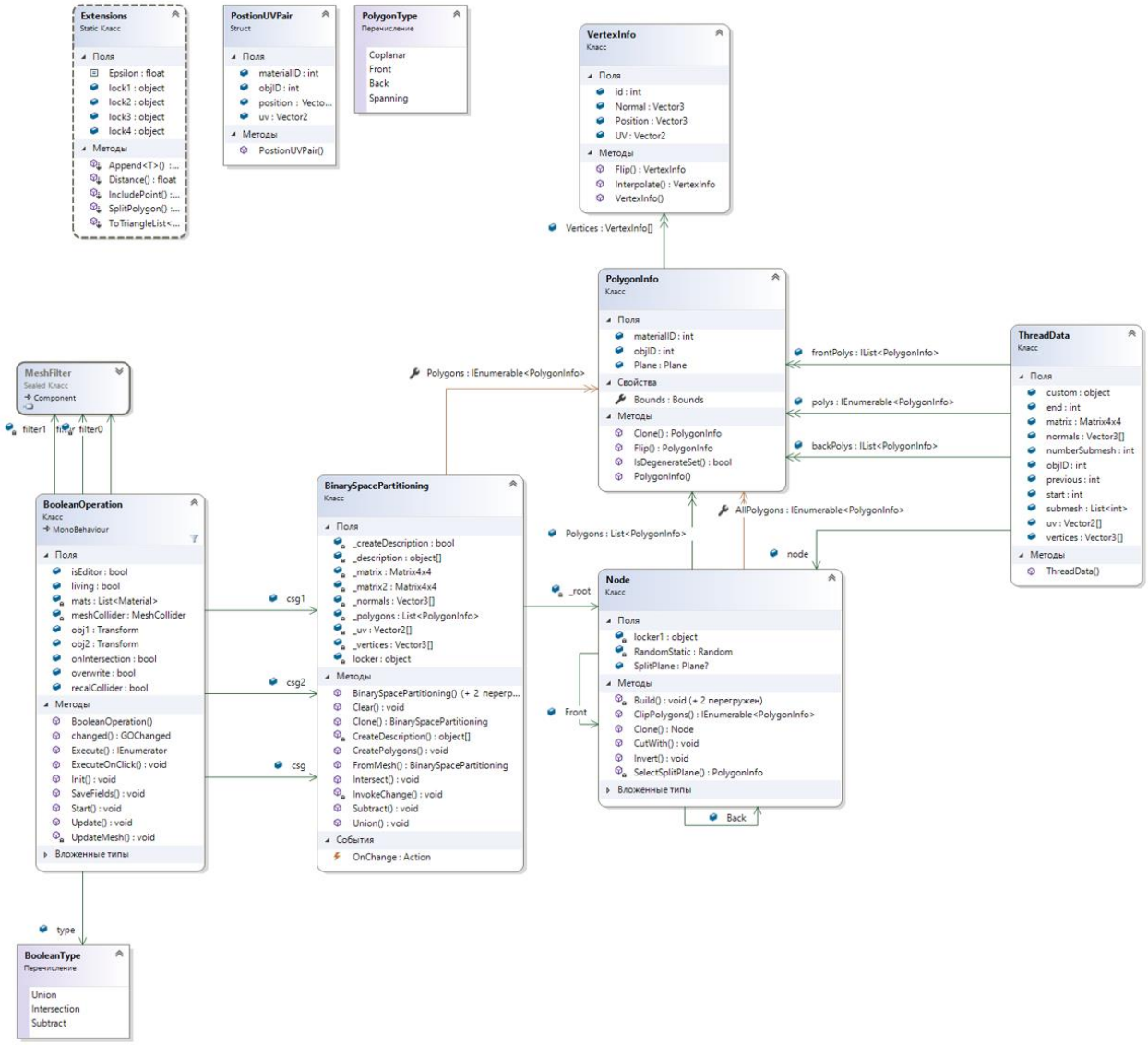

Рисунок 2 – Диаграмма классов создаваемой библиотеки

На рисунке 2 изображена структура создаваемой библиотеки для обработки топологической сетки игровых объектов путём применения логических операций.

2.3.1 Класс Extensions

Для взаимодействия с классами из пространства имен UnityEngine, добавим некоторые методы для работы с такими классами как:

- Plane,

- Bounds.

Опишем необходимые методы для реализуемого проекта.

SplitPolygon. С помощью предоставленного Plane переданный полигон заносится в один из списков: полигонов, находящихся спереди от плоскости, находящихся с обратной стороны, лежащие на данной плоскости. Если плоскость пересекает полигон, то данный полигон разделяется на две составляющие, которые уже в последствии заносятся в нужные списки.

IncludePoint. Переданная точка заносится в границы, предоставленные классом Bound.

2.3.2 Структура PostionUVPair

Создадим структуру для хранения информации о точках трёхмерных объектов с четырьмя полями:

- Position – информация о положении точки в трёхмерном пространстве;
- UV – отображении данной точки на UV карте;
- ObjID – принадлежность точки к одному из объектов преобразования;
- MaterialID – применяемый материал к данной точке.

2.3.3 Перечисление PolygonType

Для более удобного разделения плоскостью полигонов определим типы существующих полигонов по отношению к плоскости:

- Coplanar,
- Front,
- Back,
- Spanning.

2.3.4 Класс VertexInfo

Создадим класс для взаимодействия с вершинами объектов. Помимо конструктора добавим два метода:

- Flip() – для возвращения точки с обратной нормалью;
- Interpolate() – для получения точки на отрезке между данной вершиной и вершиной переданной в качестве параметра.

2.3.5 Перечисление BooleanType

В разрабатываемом приложении будет реализовано три типа логических операций: разность, объединение и пересечение объектов. Представим их в перечислении BooleanType в виде:

- Union,
- Intersection,
- Subtract.

Для выполнения логических операций над объектами нами был выбран способ с применением BSP-деревьев, поэтому необходимо создать классы, отвечающие за реализацию данных деревьев.

2.3.6 Класс Node

Для построения BSP-деревьев необходимо реализовать разделение на узлы. В каждом узле необходимо содержать информацию о выбранной плоскости, о полигонах, находящихся на ней, и о смежных узлах.

Взаимодействие будет происходить с помощью методов и свойств:

- AllPolygons – для получения всех полигонов дерева, основанием которого является данный узел;
- Invert() – инвертирование узла, переворот разделяющей плоскости;

- ClipPolygons() – отсеивание полигонов, находящихся внутри пространства ограниченного плоскостями данного BSP-дерева;
- CutWith() – отсечение полигонов с помощью дерева, переданного в виде ссылки на корневой узел;
- Clone() – создание дубликата дерева, начиная с этого узла;
- Build() – включение узла в BSP-дерево;
- Статистический SelectSplitPlane() – выбор разделяющей поверхности, проходящей через один из полигонов;
- Статистический Build() – построение BSP-дерева с данного узла и при помощи имеющихся полигонов.

2.3.7 Класс BinarySpacePartitioning

Операции над BSP-деревьями реализуем в отдельном классе BinarySpacePartitioning. Для работы над понадобятся такие методы как:

- Статистический FromMesh() – создание дерева и извлечение необходимой информации из полученного меша;
- Subtract() – вычитание переданного в качестве параметра объекта из данного;
- Intersect() – пересечение двух объектов;
- Union() – объединение двух объектов.

2.3.8 Класс BooleanOperation

Для изменения двух игровых объектов разработчик будет взаимодействовать с классом BooleanOperation. В данном классе реализовывается обработка двух экземпляров GameObject. Для каждого объекта строится BSP-дерево, производится необходимая нам булева операция и добавляются изменения в Mesh.

2.4 Вычисление булевых операций

Опишем процесс обработки булевых операций. Данный процесс будет состоять из трёх этапов:

- этап инициализации;
- вычисление необходимой булевой операции;
- заключительный этап.

2.4.1 Этап инициализации

Перед непосредственной обработкой булевой операции, каждый используемый объект проходит этап инициализации. Данный этап состоит из следующей последовательности действий:

1. Приведение всех точек к общей системе координат;
2. Приведение информации о полигонах и вершинах в структурированную форму;
3. Построение BSP-деревьев для взаимодействующих полигональных сеток.

2.4.2 Построение BSP-деревьев

Для построения BSP-деревьев был составлен рекурсивный алгоритм, который можно представить в виде пяти этапов:

1. Выбор разделяющей поверхности, проходящей через один из полигонов;
2. Разделение полученных полигонов на три группы:
    - Компланарные,
    - Находящиеся в положительном полупространстве,
    - Находящиеся в отрицательном полупространстве;

3. Добавление компланарных полигонов в текущий узел;
4. Создание двух узлов Front и Back;
5. Выполнение пунктов 1-5 для новых узлов с фронтальными полигонами для Front узла и обратными – для Back.

На рисунке 3 изображен пример одного из вариантов построения BSP-дерева для данной четырнадцатиполигональной трёхмерной модели. Зеленым выделена плоскость, проходящая через один или более полигонов. Так, например, плоскости D и F проходят через три полигона, а B и C – два. На данных узлах бинарного дерева будет хранится информация о них.

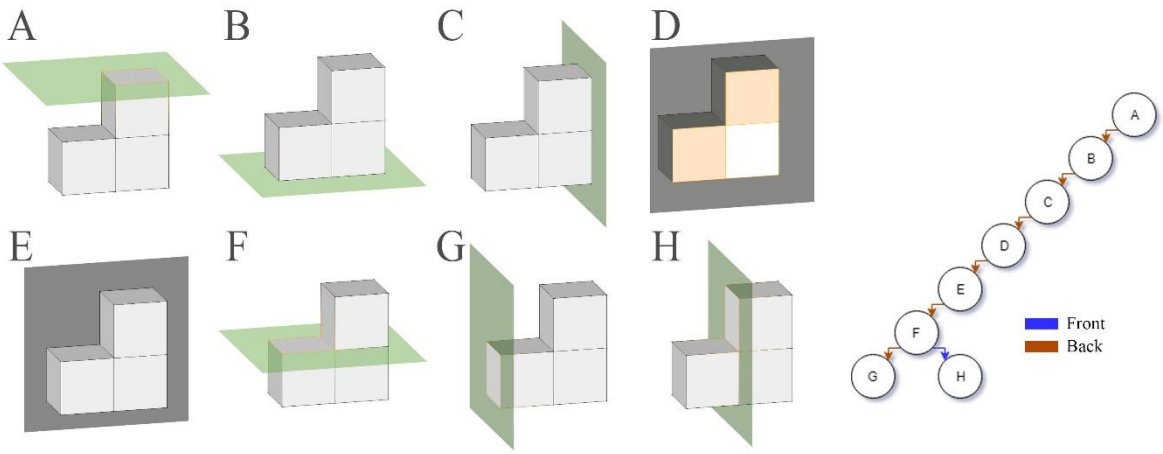

Рисунок 3 – Пример построения BSP-дерева для полигональной модели

2.4.3 Объединение

На рисунке 4 отображена последовательность действий необходимая при объединении двух полигональных моделей. Опишем данную очередность действий:

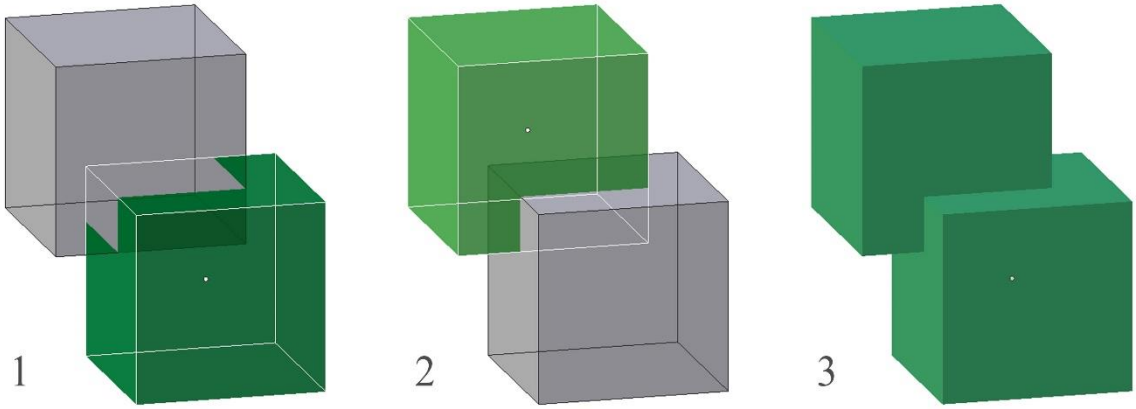

Рисунок 4 – Последовательность действий булевой операции объединения

1. Получение полигонов первого меша находящихся снаружи от второго;
2. Получение полигонов второго меша находящихся снаружи от первого;
3. Добавление в BSP-дерево первого объекта, полигоны второго.

Если разделяющая плоскость проходит через полигон и часть его находится в положительном полупространстве, а другая – в отрицательном. То полигон разбивается на два полигона. Из-за особенностей разработки в Unity, то при получении полигона, имеющего более трёх вершин, необходима его дальнейшая триангуляция.

2.4.4 Разность

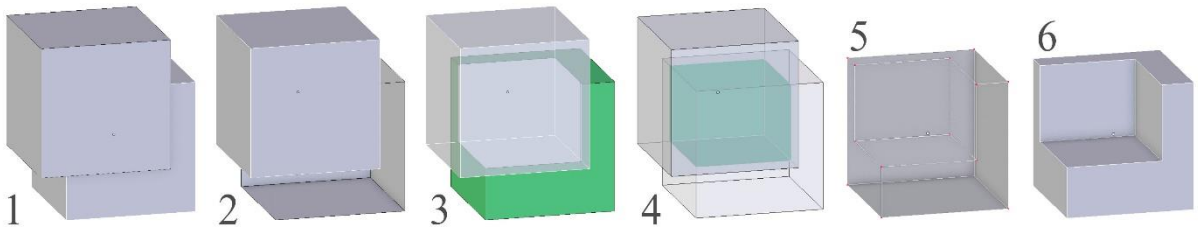

Рисунок 5 – Последовательность действий булевой операции разности

Для булевой операции разности необходимо использование той же последовательности действий, что при объединении, но с инвертированием BSP-дерева уменьшаемого объекта (см. рисунок 5). После получения результата объединения BSP-деревьев, результат инвертируется.

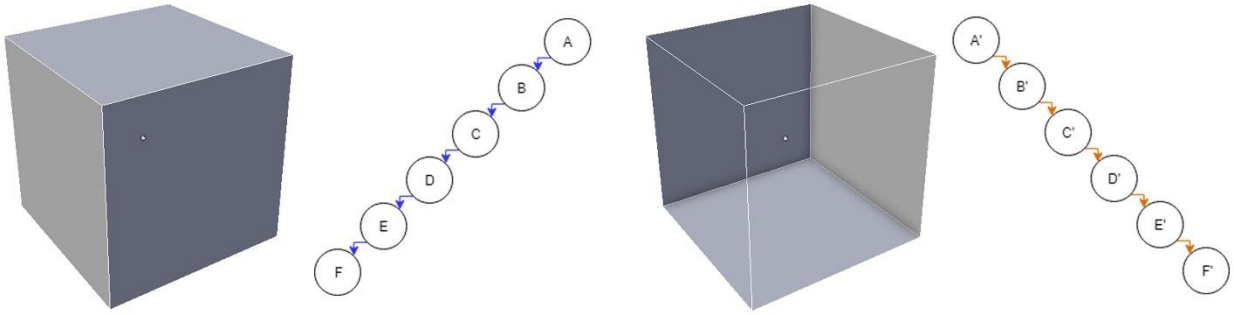

Рисунок 6 – Пример инвертирования BSP-дерева

Инвертирование преобразует дерево так положительные полупространства становятся отрицательными, а отрицательные – положительными. Как, например, при инвертировании BSP-дерева куба (см. рисунок 6) положительным становится пространство, ограниченное границами куба.

Для инвертирования необходимо:

- Перевернуть поверхность полигонов и изменить направление нормали разделяющей поверхности на противоположную;
- Изменить структуру дерева, перестановкой узлов Front и Back;
- Выполнить все пункты для дочерних узлов.

2.4.5 Пересечение

Для булевой операции пресечения необходима следующая последовательность действий (см. рисунок 7):

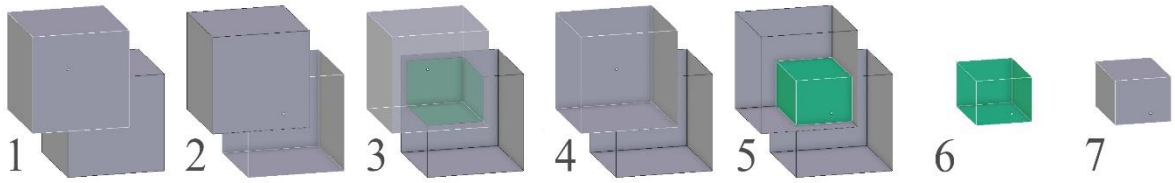

Рисунок 7 – Последовательность действий булевой операции пересечения

1. Инвертирование BSP-дерева 1-го объекта;
2. Получение полигонов 2-го меша находящихся снаружи от инвертированного 1-го;
3. Инвертирование BSP-дерева 2-го объекта;
4. Получение полигонов 1-го меша находящихся снаружи от инвертированного 2-го;
5. Добавление в BSP-дерево первого объекта, полигоны второго;
6. Инвертирование полученного дерева.

2.4.6 Заключительный этап

На заключительном этапе программы вносятся изменения в меш объекта с учётом полученного BSP-дерева.

2.5 Параллелизация обработки

Для увеличения скорости обработки требуемых вычислений, воспользуемся возможностью распараллеливания алгоритма. Для этого на узлах BSP-деревьев добавляем взаимоисключающую блокировку узла перед выполнением определенных операций над ним, а затем при переходе на дочерний узел снимаем блокировку.

Так для объединения получим построение результирующего дерева по мере поступления информации о необходимых узлах и полигонах с двух одновременно обрабатываемых деревьев.

Для обработки разности к необходимым шагам добавляется необходимость инвертирования дерева уменьшаемого объекта и дерева, полученного после объединения инвертированной и второй моделей. Поэтому старт вычислений объединения может происходить только после инвертирования первого узла первого BSP-дерева. Завершающее инвертирование необходимого узла происходит после получения всех его дочерних узлов.

В случае операции пересечения этапы инвертирования должны в обязательном порядке происходить до получения полигонов второго BSP-дерева из положительного пространства инвертируемого. Таким образом при параллелизации обязательным условием будет являться то, что получение полигонов на обрабатываемом узле второго древа может происходить только после его инвертирования.

Опыты показали, что дальнейшее разбиение на большее количество потоков не имеет смысла. Так как распределение на потоки занимает больше времени, чем синхронная обработка. Так, например, вычисление необходимости полигонов путем определения его местонахождения относительно разделяющих плоскостей второго BSP-дерева при параллелизации дает существенное снижение скорости проделываемых операций, примерно на 30-40 процентов, при расположении только нескольких полигонов на узле. Распараллеливание обработки разных листов дерева при работе с выпуклыми многогранниками тоже даст только снижение производительности.

Прирост производительности будет заметен на многоядерных процессорах. Так как максимальным количеством потоков является шесть, то максимальный прирост дадут шестиядерные процессоры.

2.6 Результаты работы системы

Рассмотрим, результаты разработанной программы. Для наглядного демонстрирования функциональности программы используем два объекта разной формы и размерности: куб и сферу (см. рисунок 8).

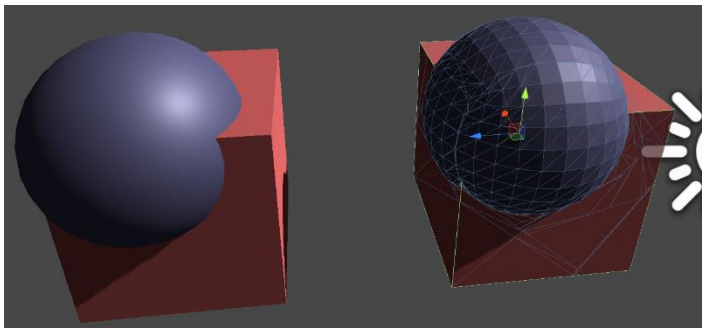

Рисунок 8 – Пример работы программы при объединении трёхмерных моделей

В результате операции объединения получим объект, состоящий из видимых полигонов двух изначальных мешей. При разности, когда куб является уменьшаемым объектом, а сфера – вычитаемым, результирующей полигональной моделью будет часть куба, заключенная вне границ полигональной сетки сферы (см. рисунок 9). Для булевой операции

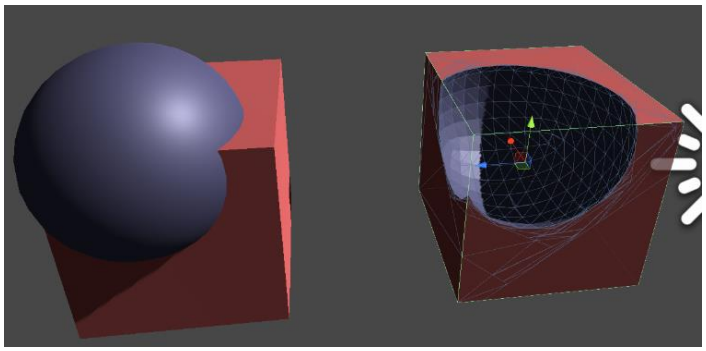

Рисунок 9 – Пример работы программы при разности трёхмерных моделей

пересечения результатом работы программы будет являться объект, находящийся внутри границ обоих заданных объектов (см. рисунок 10).

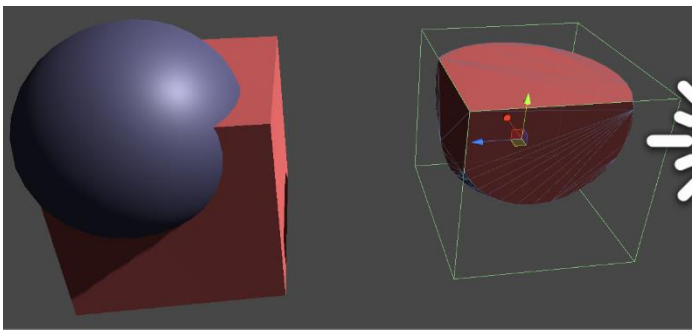

Рисунок 10 – Пример работы программы при пересечении трёхмерных моделей

2.6.1 Скорость обработки

Замерим результаты обработки полученной программы. Результаты получены с помощью компьютера с процессором AMD A10-5700. Так как данный процессор имеет два физических ядра, то замеры осуществим на операции объединения. Так как скорость может зависеть от других программ и игрового движка возьмем наименьшее значение из десяти одинаковых вычислений.

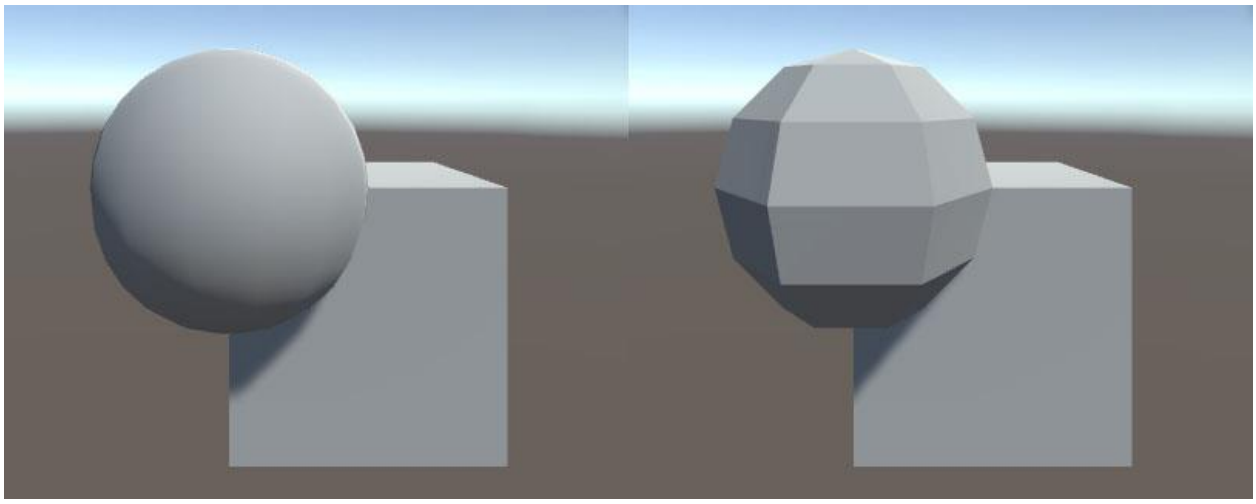

Рисунок 11 – Расположение сферы и куба при замерах времени вычисления булевой операции объединения

Расположим куб с ребром длиной 1 метр на точку начала координат, а сферу с радиусом 0,5 метра на угол куба, а именно на (0,5; 0,5; 0,5), как показано на рисунке 11. Замеры произведены без этапа инициализации.

В таблице 1 представим результаты замеров вычисления булевых операций при росте количества полигонов у куба, тем самым увеличивая количество хранимых полигонов на узлах BSP-деревьев.

Таблица 1 – Время вычисления операции объединения при росте количества полигонов у куба

Полигонов куба	Полигонов сферы	Общее количество полигонов	Минимальное время обработки, мс
12	60	72	3
48	60	108	4
108	60	168	6
192	60	252	7
300	60	360	9
432	60	492	12
1200	60	1260	14

В таблице 2 представим результаты замеров вычисления булевых операций при росте количества полигонов у сферы, тем самым увеличивая количество узлов BSP-деревьев. Так как сфера является выпуклым многогранником, то у его дерева увеличивается, как общее число узлов, так и глубина самого дерева. Как можно заметить, при увеличении глубины дерева, сильно снижается производительность разработанной программы и при следующем подразбиении сферы, программа не укладывается в рамки реального времени, так как не укладывается в установленный нами дедлайн.

Таблица 2 – Время вычисления операции объединения при росте количества полигонов у сферы

Полигонов куба	Полигонов сферы	Общее количество полигонов	Минимальное время обработки, мс
12	60	72	3
12	84	96	5
12	144	156	8
12	364	376	25

Попробуем заменить сферу тором. Так данная фигура является невыпуклым многогранником, то BSP-дерево данного объекта будет более сбалансированным, но топология усложнится из-за пересечения разделяющих плоскостей самого объекта, что росту полигонов на этапе инициализации. Замерим результаты вычислений (см. таблицу 3). Время вычислений примерно сопоставимо с временем объединения куба и сферы.

Таблица 3 – Время вычисления операции объединения куба с тором при росте количества полигонов у тора

Полигонов куба	Полигонов тора	Общее количество полигонов	Минимальное время обработки, мс
12	32	44	2
12	50	62	3
12	72	84	4
12	98	110	8

Полигонов куба	Полигонов тора	Общее количество полигонов	Минимальное время обработки, мс
12	128	140	8
12	162	174	9
12	288	300	16

2.6.2 Недостатки разработанной системы

Наряду с описанными достоинствами разработанной программы, в ней присутствуют и два недостатка. Одним из таких является вышеописанная проблема снижения производительности при увеличении генерируемого BSP-дерева, что накладывает ограничения при работе с данной системой. Так при работе с высокополигональными моделями для сохранения свойств реального времени вычисления необходимо производить над частью выбранного меша, а не над цельной полигональной сеткой.

Так же у созданного инструментария в случае несбалансированного расположения вершин у полигональных сеток объектов, проявляются

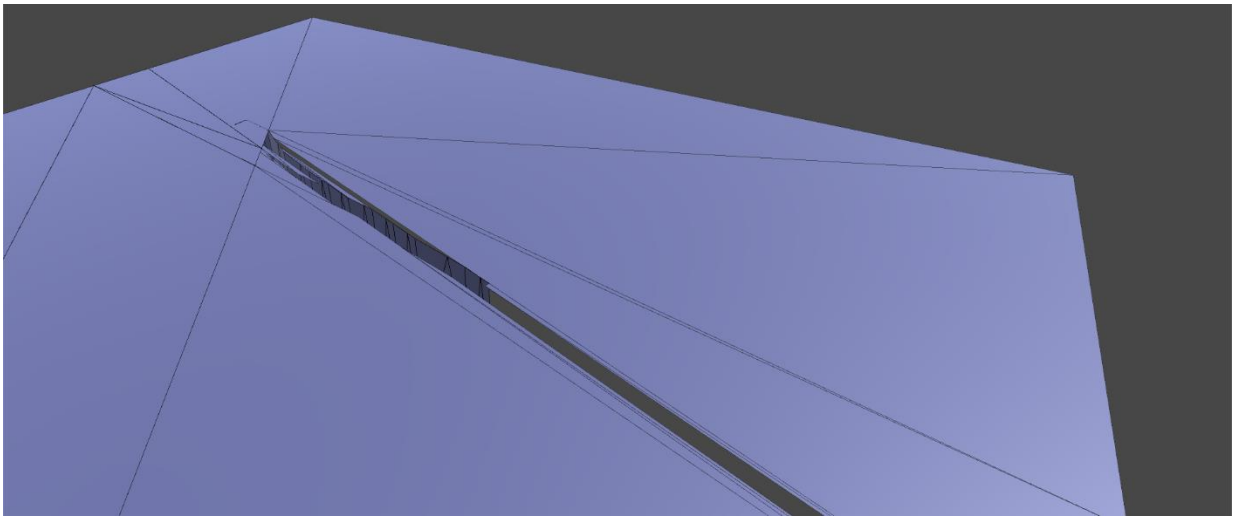

Рисунок 12 – Пример проявления нестабильности системы

проблемы со стабильностью булевых операций (см. рисунок 12). Что влечет за

собой необходимость перепроверки полученных результатов вычисления булевых операций.

Заключение

В данной работе была выделена проблема изменения полигональных моделей при их соприкосновении с другими полигональными объектами. Так как создание универсального инструмента, позволяющего производить булевы операции над полигональными объектами, частично решает данную проблему, была поставлена цель создания библиотеки на языке C# в игровом движке Unity для изменения топологической структуры игровых объектов путём применения логических операций.

Все поставленные цели были достигнуты, а задачи были успешно выполнены. Были изучены новейшие исследования по изменению полигональных трёхмерных моделей с помощью логических операций над ними, выявлены достоинства и недостатки существующих методов решения выделенной проблемы. В данной работе была создана необходимая библиотека классов, способная выполнять быстрые булевы операции:

- объединения,
- разности,
- пересечения.

При условии дальнейшего развития данной работы возможно увеличение стабильности и скорости работы системы с высокополигональными объектами. Поиск новейших методов триангуляции и последующее их применение для полигонов, полученных при их разбиении с помощью, разделяющей плоскости. Создание генератора трёхмерных моделей для дальнейшего изучения достоинств и недостатков реализованной программы.

Список использованных источников

1. Kugurakova V., Khafizov M., Akhmetsharipov R., Lushnikov A., Galimova D., Abramov V., Madrigal O.C. Virtual surgery system with realistic visual effects and haptic interaction // ICAROB 2017: Proceedings of the 2017 International Conference on Artificial Life and Robotics, 2017. pp. 86-89.
2. Abramov V.D., Kugurakova V.V., Sultanova R.R., Talanov M.O., Civilskij I.V. Virtual Reality-Based Immersive Simulation for Invasive Surgery Training // European journal of clinical investigation, Vol. 48, 2018. pp. 224-225.
3. Boolean Compound Object [Электронный ресурс] // Autodesk knowledge network: [сайт]. [2018]. URL: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/3DSMax-Modeling/files/GUID-3DBEB7C2-43CC-4B78-9463-5DD448FD921C-htm.html (дата обращения: 08.01.2019).
4. Boolean Modifier [Электронный ресурс] // Blender Manual: [сайт]. URL: https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/booleans.html (дата обращения: 08.01.2019).
5. Решения [Электронный ресурс] // Unity: [сайт]. [2019]. URL: https://unity.com/ru/solutions (дата обращения: 04.01.2019).
6. Zhou Q., Grinspun E., Zorin D., Jacobson A. Mesh arrangements for solid geometry // ACM Transactions on Graphics (TOG), Vol. 35, No. 4, 2016. pp. 39-50.
7. Fogel E., Teillaud M. The computational geometry algorithms library CGAL // ACM Communications in Computer Algebra, Vol. 49, No. 1, 2015. pp. 10-12.
8. Alliez P., Fabri A. CGAL - the Computational Geometry Algorithms // ACM SIGGRAPH 2016 Courses, 2016. pp. 1-195.

9. Xiao Z., Chen J., Zheng, Y. Z.J., Wang D. Booleans of triangulated solids by a boundary conforming tetrahedral mesh generation approach // Computers & Graphics, Vol. 59, 2016. pp. 13-27.

10. Charton J., Kim L., Kim Y. Boolean operations between two colliding shells: a robust, exact, and simple method // Journal of Advanced Mechanical Design, Systems, and Manufacturing, Vol. 11, No. 4, 2017.

11. Wang Z.J., Lin X., Fang M.E., Yao B., Peng Y., Guan H., Guo M. Re2l: An efficient output-sensitive algorithm for computing Boolean operations on circular-arc polygons and its applications // Computer-Aided Design, Vol. 83, 2017. pp. 1-14.

12. Jiang X., Peng Q., Cheng X., Dai N., Cheng C., Li D. Efficient booleans algorithms for triangulated meshes of geometric modeling // Computer-Aided Design and Applications, Vol. 13, No. 4, 2016. pp. 419-430.

13. Feito F.R., Ogáyar C.J., Segura R.J., Rivero M.L. Fast and accurate evaluation of regularized Boolean operations on triangulated solids // Computer-Aided Design, Vol. 45, No. 3, 2013. pp. 705-716.

14. Landier S. Boolean operations on arbitrary polygonal and polyhedral meshes // Computer-Aided Design, Vol. 85, 2017. pp. 138-153.

15. Barki H., Guennebaud G., Foufou S. Exact, robust, and efficient regularized Booleans on general 3D meshes // Computers & Mathematics with Applications, Vol. 70, No. 6, 2015. pp. 1235-1254.

16. Douze M., Franco J.S., Raffin B. QuickCSG: Arbitrary and faster boolean combinations of n solids // Inria-Research Centre Grenoble–Rhône-Alpes, 2015. pp. 1-41.

17. Schmidt R., Brochu T. Adaptive mesh booleans // arXiv preprint arXiv:1605.01760., 2016. pp. 1-10.

18. Hao J., Sun J., Chen Y., Cai Q., Tan L. Optimal Reliable Point-in-Polygon Test and Differential Coding Boolean Operations on Polygons // Symmetry, Vol. 10, No. 10, 2018. pp. 477-502.

19. Bernstein G., Fussell D. Fast, exact, linear booleans // Computer Graphics Forum, Vol. 28, No. 5, 2009. pp. 1269-1278.

20. 3D Theory - Binary Space Partitioning (BSP) tree [Электронный ресурс] // EuclideanSpace - Mathematics and Computing: [сайт]. [2017]. URL: https://euclideanspace.com/threed/solidmodel/spatialdecomposition/bsp/index.htm (дата обращения: 14.05.2019).

21. Ranta-Eskola S., Olofsson E. Binary space partioning trees and polygon removal in real time 3d rendering // Uppsala master's theses in computing science, 2001.

22. Naylor B.F. A Tutorial on Binary Space Partitioning Trees, 2004.

23. Мобильные игры [Электронный ресурс] // Unity: [сайт]. [2019]. URL: https://unity.com/ru/solutions/mobile (дата обращения: 17.05.2019).

24. Расширение редактора Unity через Editor Window, Scriptable Object и Custom Editor [Электронный ресурс] // habr: [сайт]. [2018]. URL: https://habr.com/ru/post/431856/ (дата обращения: 17.04.2019).

25. Scripting API: Editor [Электронный ресурс] // Unity: [сайт]. [2019]. URL: https://docs.unity3d.com/ScriptReference/Editor.html (дата обращения: 15.04.2019).

26. Schiavullo R. What Game Engine Should I Use?(Unreal Engine 4, Cry Engine V, Unity 5, Amazon Lumberyard, Frostbite, Source) // Virtual Reality, Vol. 5, 2018. pp. 1-11.

27. Руководство Unity [Электронный ресурс] // Unity: [сайт]. [2018]. URL: https://docs.unity3d.com (дата обращения: 12.11.2018).

28. Visual Studio Community [Электронный ресурс] // Microsoft: [сайт]. [2019]. URL: https://visualstudio.microsoft.com/ru/vs/community (дата обращения: 08.05.2019).

29. Ritchie P. Practical Microsoft Visual Studio 2015. Berkeley: Apress, 2016. 209 pp.

30. Visual Studio 2017 version 15.7 Release Notes [Электронный ресурс] // Microsoft Docs: [сайт]. [2018]. URL: https://docs.microsoft.com/en-us/visualstudio/releasenotes/vs2017-relnotes-v15.7 (дата обращения: 25.09.2018).

31. Jansson E.S.V. Performance Effect of Flyweight in Soft Real-Time Applications // eriksvjansson, October 2015. pp. 1-8.

32. Scripting API: Mesh [Электронный ресурс] // Unity: [сайт]. [2019]. URL: https://docs.unity3d.com/ScriptReference/Mesh.html (дата обращения: 02.05.2019).

33. Topology [Электронный ресурс] // Polycount: [сайт]. [2018]. URL: http://wiki.polycount.com/wiki/Topology (дата обращения: 02.12.2018).

Глоссарий

B-Rep – метод представления границ, основанный на параметрических поверхностях.

BSP (binary space partitioning) — метод рекурсивного разбиения евклидова пространства в выпуклые множества и гиперплоскости.

Play mode – режим в Unity, созданный для тестирования созданных проектов внутри среды.

Ассет – цифровой объект, преимущественно состоящий из однотипных данных, неделимая сущность, которая представляет часть игрового контента и обладает некими свойствами. Понятие «игрового ассета» используется при разработке компьютерных игр по отношению к тем элементам контента, которые обрабатываются ресурсной системой как неделимые (атомарные, элементарные) сущности.

Блокировка – механизм синхронизации, позволяющий обеспечить исключительный доступ к разделяемому ресурсу между несколькими потоками. Блокировки – это один из способов обеспечить политику управления распараллеливанием.

Булевы операции над многоугольниками – это набор булевых операций (AND, OR, NOT, XOR, ...) с одним или несколькими наборами многоугольников в компьютерной графике.

Виртуальная реальность – созданный техническими средствами мир, передаваемый человеку через его ощущения: зрение, слух, осязание и другие.

Выпуклый многогранник – частный случай многогранника, пересечение конечного числа замкнутых полупространств.

Глубина дерева графа G – минимальная высота леса F со свойством, что любое ребро графа G соединяет пару вершин, связанных отношением предок-потомок.

Двоичное разбиение пространства – см. BSP.

Дедлайн – критический срок обслуживания, предельный срок завершения какой-либо работы.

Дополненная реальность – результат введения в поле восприятия любых сенсорных данных с целью дополнения сведений об окружении и улучшения восприятия информации.

Игровой движок – базовое программное обеспечение компьютерной видеоигры. В общем случае термин применяется для того программного обеспечения, которое пригодно для повторного использования и расширения, и тем самым может быть рассмотрено как основание для разработки множества различных игр без существенных изменений.

Компланарность – характеристика векторов. Три вектора (или большее число) называются компланарными, если они, будучи приведёнными к общему началу, лежат в одной плоскости.

Конформное отображение — непрерывное отображение, сохраняющее углы между кривыми, а значит и форму бесконечно малых фигур.

Меш – см. Полигональная сетка.

Подразделение поверхности в области трехмерной компьютерной графики представляет собой метод представления гладкой поверхности посредством спецификации более грубой кусочно-линейной многоугольной сетки. Гладкая поверхность может быть представлена в виде грубой сетки, как предел рекурсивного разбиения каждой многоугольной грани на более мелкие грани, которые приближаются к гладкой поверхности.

Полигон – многоугольник, минимальная поверхность для визуализации в трёхмерной графике.

Полигональная сетка – это совокупность вершин, рёбер и граней, которые определяют форму многогранного объекта в трёхмерной компьютерной графике и объёмном моделировании.

Полупространство, ограниченное гиперплоскостью α, – это геометрическая фигура в пространстве, для которой выполняется следующее:

1. эта фигура включает в себя плоскость α, но не сводится к ней;
2. любой отрезок, ограниченный произвольными точками этой фигуры A и B, не принадлежащими α, не имеет пересечений с плоскостью α;
3. любой отрезок, ограниченный произвольными точками этой фигуры A и B, где A принадлежит α, а B — нет, имеет пересечение с плоскостью α.

Поток выполнения – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса.

Реальное время – количественная характеристика, которая может быть измерена реальными физическими часами, в отличие от логического времени, определяющего лишь качественную характеристику, выражаемую относительным порядком следования событий.

Система реального времени – это система, которая должна реагировать на события во внешней по отношению к системе среде или воздействовать на среду в рамках требуемых временных ограничений.

Системы мягкого реального времени – это системы реального времени, нарушения характеристик которых приводят лишь к снижению качества работы системы.

Тетраэдризация – триангуляция трёхмерного пространства.

Топология – это макет модели, способ размещения вершин и ребер для создания поверхности сетки [33].

Триангулированная модель – полигональная модель, состоящая из треугольников.

Триангуляция *T* пространства R^{n+1} – это подразбиение R^{n+1} на (n+1)-мерные симплексы, такие что: любые два симплекса в T пересекаются в общей грани ребра или вершины, или вообще не пересекаются; любое ограниченное множество в R^{n+1} пересекает конечное количество симплексов с T.

Приложение А. Исходный код разработанной библиотеки классов

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using UnityEngine;

namespace MyBoolean
{
    [Serializable]
    public class BinarySpacePartitioning
    {
        static object locker = new object();
        static List<PolygonInfo> _polygons;
        static Matrix4x4 _matrix;
        static Matrix4x4 _matrix2;
        static Vector3[] _vertices;
        static Vector3[] _normals;
        static Vector2[] _uv;
        Node _root;

        public event Action OnChange;

        public IEnumerable<PolygonInfo> Polygons
        {
            get
            {
                var res = _root.AllPolygons;
                return res;
            }
        }

        public BinarySpacePartitioning(bool createDescription)
          : this(new Node())
        {
        }
```

```
        protected BinarySpacePartitioning(IEnumerable<PolygonInfo> polygons)
          : this(true)
        {
            _root.Build(polygons);
        }

        private BinarySpacePartitioning(Node root)
        {
            _root = root;
        }

        public BinarySpacePartitioning Clone()
        {
            var res = new BinarySpacePartitioning(_root.Clone());
            return res;
        }

        public virtual void Union(BinarySpacePartitioning bInput)
        {
            Node root = _root;
            Node other = bInput._root.Clone();

            Parallel.Invoke(
                () => root.CutWith(other),
                () => other.CutWith(root),
                () => root.Build(other.AllPolygons)
                );
        }

        public static BinarySpacePartitioning FromMesh(Mesh mesh, Transform
transform, Transform transform2, int objID, int previous = 0)
        {
            _polygons = new List<PolygonInfo>(mesh.triangles.Length);
            _matrix = (transform == null) ? Matrix4x4.identity :
transform.localToWorldMatrix;
            _matrix2 = (transform2 == null) ? Matrix4x4.identity :
transform2.worldToLocalMatrix;
            _normals = mesh.normals;
            _uv = mesh.uv;
```

```
            _vertices = mesh.vertices;
            List<List<int>>       intTrianglesAllSubMeshes       =       new
List<List<int>>(mesh.subMeshCount);
            for (int index = 0; index < mesh.subMeshCount; ++index)
                intTrianglesAllSubMeshes.Add(new
List<int>(mesh.GetTriangles(index)));
            for (int i = 0; i < intTrianglesAllSubMeshes.Count; ++i)
            {
                List<int> currentIntTriangles = intTrianglesAllSubMeshes[i];
                int     length     =     Mathf.Min(SystemInfo.processorCount,
currentIntTriangles.Count);
                if (length <= 0)
                    length = 1;
                int    num    =    currentIntTriangles.Count    /    length    -
currentIntTriangles.Count / length % 3;
                Thread[] threadArray = new Thread[length];
                int threadNumber;
                for (threadNumber = 0; threadNumber < length; threadNumber++)
                {
                    ThreadData threadData = new ThreadData(num * threadNumber,
threadNumber   !=   length   -   1   ?   num   *   (threadNumber   +   1)   :
currentIntTriangles.Count)
                    {
                        objID = objID,
                        previous = previous,
                        submesh = currentIntTriangles,
                        numberSubmesh = i
                    };
                    threadArray[threadNumber]      =      new      Thread(new
ParameterizedThreadStart(CreatePolygons))
                    {
                        Priority = System.Threading.ThreadPriority.Highest
                    };
                    threadArray[threadNumber].Start(threadData);
                }
                foreach (Thread thread in threadArray)
                    thread.Join();
            }
            var res = new BinarySpacePartitioning(_polygons);
            return res;
```

```
        }

        public static void CreatePolygons(object objThreadData)
        {
            ThreadData threadData = objThreadData as ThreadData;
            int threadI = threadData.start;
            while (threadI < threadData.end)
            {
                var csgVertexList = new VertexInfo[3];
                for (int i = 0; i < 3; i++)
                {
                    var id = threadData.submesh[threadI + i];
                    var                          pos                          =
_matrix2.MultiplyPoint3x4(_matrix.MultiplyPoint3x4(_vertices[id]));
                    var csgVertex = new VertexInfo(pos, _normals[id], _uv[id],
id);
                    csgVertexList[i] = csgVertex;
                }
                lock (locker)
                    _polygons.Add(new               PolygonInfo(csgVertexList,
threadData.objID, threadData.previous + threadData.numberSubmesh));
                threadI += 3;
            }
        }

        public virtual void Subtract(BinarySpacePartitioning bspInput)
        {
            Node root = _root;
            Node other = bspInput._root.Clone();
            Parallel.Invoke(
                () => root.Invert(),
                () => root.CutWith(other),
                () =>
                {
                    Thread.Sleep(2);
                    other.CutWith(root);
                },
                () =>
                {
```

```
            root.Build(other.AllPolygons);
            root.Invert();
        }
        );
}

// Объединение bsp с bInput
public virtual void Intersect(BinarySpacePartitioning bInput)
{
    Node root = _root;
    Node other = bInput._root.Clone();
    Parallel.Invoke(
    () => root.Invert(),
    () =>
    {
        Thread.Sleep(2);
        other.CutWith(root);
    },
    () => other.Invert(),
    () =>
    {
        Thread.Sleep(2);
        root.CutWith(other);
    },
    () =>
    {
        root.Build(other.AllPolygons);
        root.Invert();
    }
    );
}
public virtual void Clear()
{
    _root = new Node();
}
```

```
        public void ToTriangleList<V, I>(Func<Vector3, Vector2, int, int, V>
uvPair, Func<V, I> insertVertex, Action<I, I, I, PostionUVPair> createTriangle,
IEnumerable<PolygonInfo> polygons)
        {
            foreach (var polygon in polygons)
            {
                I[] array = polygon.Vertices.ToList().FindAll(x => x !=
null).Select(a => uvPair(a.Position, a.UV, polygon.objID,
polygon.materialID)).Select(a => insertVertex(a)).ToArray();
                for (int index = 2; index < array.Length; ++index)
                    createTriangle(array[0], array[index - 1], array[index],
new PostionUVPair(Vector3.zero, Vector2.zero, polygon.objID,
polygon.materialID));
            }
        }
    }
}
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace MyBoolean
{
    public static class Extensions
    {
        public static object lock1 = new object();
        public static object lock2 = new object();
        public static object lock3 = new object();
        public static object lock4 = new object();
        public const float Epsilon = 0.0001f;
        public static void SplitPolygon(this Plane plane, PolygonInfo polygon,
IList<PolygonInfo> coPlanarFront, IList<PolygonInfo> coPlanarBack,
IList<PolygonInfo> front, IList<PolygonInfo> back, bool needToSplit = false)
        {
            var vxCount = 0;
            var vertexTypeList = new PolygonType[10];
            var polygonType = PolygonType.Coplanar;
            if (plane.Equals(polygon.Plane))
```

```
            {
                vertexTypeList[vxCount++] = PolygonType.Coplanar;
            }
            else
            {
                for (int index = 0; index < polygon.Vertices.Length; ++index)
                {
                    if (polygon.Vertices[index] == null)
                        break;
                    float num = plane.GetDistanceToPoint(polygon.Vertices[index].Position);
                    PolygonType vertexType = num < -Epsilon ? PolygonType.Back : (num > Epsilon ? PolygonType.Front : PolygonType.Coplanar);
                    polygonType |= vertexType;
                    vertexTypeList[vxCount++] = vertexType;
                }
            }
            switch (polygonType)
            {
                case PolygonType.Coplanar:
                    Vector3 normalPlane = plane.normal;
                    Plane plane1 = polygon.Plane;
                    Vector3 normalPolygon = plane1.normal;
                    if (Vector3.Dot(normalPlane, normalPolygon) > 0.0)
                    {
                        lock (lock3)
                        {
                            coPlanarFront.Add(polygon);
                            break;
                        }
                    }
                    else
                    {
                        lock (lock4)
                        {
                            coPlanarBack.Add(polygon);
                            break;
                        }
```

```
                }
            case PolygonType.Front:
                lock (lock1)
                {
                    front.Add(polygon);
                    break;
                }
            case PolygonType.Back:
                lock (lock2)
                {
                    back.Add(polygon);
                    break;
                }
            case PolygonType.Spanning:
                var vertexFrontList = new VertexInfo[10];
                var vertexBackList = new VertexInfo[10];
                var iF = 0;
                var iB = 0;
                for (int index1 = 0; index1 < vxCount; ++index1)
                {
                    int index2 = (index1 + 1) % vxCount;
                    PolygonType vertexType1 = vertexTypeList[index1];
                    VertexInfo vertex1 = polygon.Vertices[index1];
                    VertexInfo vertex2 = polygon.Vertices[index2];
                    if (vertexType1 != PolygonType.Back)
                        vertexFrontList[iF++] = vertex1;
                    if (vertexType1 != PolygonType.Front)
                        vertexBackList[iB++] = vertex1;
                    if   ((vertexType1   |   vertexTypeList[index2])   ==
PolygonType.Spanning)
                    {
                        // Добавляется копланарная точка
                        float      t      =      (-plane.distance      -
Vector3.Dot(plane.normal,   vertex1.Position))   /   Vector3.Dot(plane.normal,
vertex2.Position - vertex1.Position);
                        VertexInfo            vertexCoplanar            =
vertex1.Interpolate(vertex2, t);
                        vertexFrontList[iF++] = vertexCoplanar;
                        vertexBackList[iB++] = vertexCoplanar;
```

```
                    }
                }


                void AddPoly(ref VertexInfo[] vertices, ref int count, ref
IList<PolygonInfo> list)
                {
                    if (!PolygonInfo.IsDegenerateSet(vertices))
                    {
                        if (count == 3)
                            lock (lock1)
                                list.Add(new         PolygonInfo(vertices,
polygon.objID, polygon.materialID));
                        else
                            for (int i = 1; i < count - 1; i++)

                                lock (lock1)
                                    list.Add(new           PolygonInfo(new
VertexInfo[] { vertices[0], vertices[i], vertices[i + 1] }, polygon.objID,
polygon.materialID));

                    }
                }

                AddPoly(ref vertexFrontList, ref iF, ref front);
                AddPoly(ref vertexBackList, ref iB, ref back);
                break;
            }
        }


        public static Bounds IncludePoint(this Bounds bound, Vector3 point)
        {
            Vector3 vector3_1;
            vector3_1.x = Math.Min(bound.min.x, point.x);
            vector3_1.y = Math.Min(bound.min.y, point.y);
            vector3_1.z = Math.Min(bound.min.z, point.z);
            Vector3 vector3_2;
            vector3_2.x = Math.Max(bound.max.x, point.x);
```

```
            vector3_2.y = Math.Max(bound.max.y, point.y);
            vector3_2.z = Math.Max(bound.max.z, point.z);
            bound.SetMinMax(vector3_1, vector3_2);
            return bound;
        }
    }
}
using System;
namespace MyBoolean
{
    public enum PolygonType
    {
        Coplanar = 0,
        Front = 1,
        Back = 2,
        Spanning = Back | Front,
    }
}
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
namespace MyBoolean
{
    public class BooleanOperation : MonoBehaviour
    {
        public Transform Transform1;
        public BooleanType OperationType;
        public bool RecalCollider;
        public BinarySpacePartitioning tree;
        public BinarySpacePartitioning tree1;
        public BinarySpacePartitioning tree2;
        public bool onIntersection;
        public Transform Transform2;
        private MeshFilter _filter;
        private MeshFilter _filter0;
        private MeshFilter _filter1;
        private List<Material> _mats = new List<Material>();
```

```
        private MeshCollider _meshCollider;
        public void SaveFields()
        {
            _filter = GetComponent<MeshFilter>();
            _filter0 = Transform1.GetComponent<MeshFilter>();
            _filter1 = Transform2.GetComponent<MeshFilter>();
            _mats.Clear();

_mats.AddRange(Transform1.GetComponent<Renderer>().sharedMaterials);

_mats.AddRange(Transform2.GetComponent<Renderer>().sharedMaterials);
        }
        public void Apply(int num)
        {
            if (Transform1 == null || Transform2 == null)
            {
                return;
            }
            SaveFields();
            if (num > 0 && _filter0 != null && _filter1 != null &&
_filter0.sharedMesh != null && _filter1.sharedMesh != null)
            {
                if (tree1 == null || num == 1 || num == 3)
                    tree1                                                    =
BinarySpacePartitioning.FromMesh(_filter0.sharedMesh, null, null, 0, 0);
                if (tree2 == null || num == 2 || num == 3)
                    tree2                                                    =
BinarySpacePartitioning.FromMesh(_filter1.sharedMesh, Transform2, Transform1,
1, 1);
                tree = tree1.Clone();
                Timer.Start("un");

                switch (OperationType)
                {
                    case BooleanType.Intersection:
                        tree.Intersect(tree2);
                        break;
                    case BooleanType.Subtract:
                        tree.Subtract(tree2);
                        break;
```

```
                    case BooleanType.Union:
                        tree.Union(tree2);
                        break;
                }
                UpdateMesh();
                Timer.End();
            }
            Debug.Log(Timer.ToString());
        }

        private void UpdateMesh()
        {
            if (tree == null)
            {
                return;
            }

            List<Vector3> vertices = new
List<Vector3>(_filter0.sharedMesh.vertexCount +
_filter1.sharedMesh.vertexCount);
            List<Vector2> uvs = new
List<Vector2>(_filter0.sharedMesh.uv.Length + _filter1.sharedMesh.uv.Length);
            List<int> intTriangles = new
List<int>(_filter0.sharedMesh.triangles.Length +
_filter1.sharedMesh.triangles.Length);
            List<int>[] submeshes = new List<int>[30];

            //
            PostionUVPair newPostionUVPair(Vector3 p, Vector2 n, int o, int m)
=> new PostionUVPair(pos: new Vector3(p.x, p.y, p.z), uv: n, objID: o,
materialID: m);
            //
            int insertVertex(PostionUVPair v)
            {
                vertices.Add(v.Position);
                uvs.Add(v.UV);
                return vertices.Count - 1;
            }
            //
            void createTriangle(int a, int b, int c, PostionUVPair k)
```

```
            {
                if (submeshes[k.MaterialID] == null)
                    submeshes[k.MaterialID] = new List<int>(100);
                submeshes[k.MaterialID].Add(a);
                submeshes[k.MaterialID].Add(b);
                submeshes[k.MaterialID].Add(c);
            }
            //
            tree.ToTriangleList(newPostionUVPair,                insertVertex,
createTriangle, tree.Polygons.Append(tree2.Polygons));
            Mesh mesh = _filter.sharedMesh;
            if (mesh == null)
                mesh = new Mesh();
            mesh.vertices = vertices.ToArray();
            int subMeshCount = 0;
            for (int index = 0; index < submeshes.Length; ++index)
            {
                List<int> intList2 = submeshes[index];
                if (intList2 != null && intList2.Count > 0)
                    ++subMeshCount;
            }
            mesh.subMeshCount = subMeshCount;
            int num2 = 0;
            for (int index = 0; index < submeshes.Length; ++index)
            {
                List<int> intList2 = submeshes[index];
                if (intList2 != null && intList2.Count > 0)
                {
                    mesh.SetTriangles(intList2.ToArray(), num2);
                    ++num2;
                }
            }
            mesh.uv = uvs.ToArray();
            mesh.RecalculateNormals();
            mesh.RecalculateBounds();
            _filter.sharedMesh = mesh;
            _filter.GetComponent<Renderer>().sharedMaterials                =
_mats.ToArray();
```

```
            if (RecalCollider)
            {
                if (_meshCollider == null)
                {
                    _meshCollider = GetComponent<MeshCollider>();
                    if (_meshCollider == null)
                        _meshCollider                                                    =
gameObject.AddComponent<MeshCollider>();
                }
                if (_meshCollider == null)
                    _meshCollider.sharedMesh = _filter.sharedMesh;
            }
        }
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using UnityEngine;
namespace MyBoolean
{
    public class Node
    {
        public List<PolygonInfo> Polygons = new List<PolygonInfo>();
        public Plane? SplitPlane;
        public Node Front;
        public Node Back;
        public System.Object Locker = new System.Object();

        [ThreadStatic]
        private static System.Random RandomStatic;
        private static object locker1 = new object();

        public IEnumerable<PolygonInfo> AllPolygons
        {
            get
```

```
    {
        lock (Locker)
        {
            foreach (PolygonInfo polygon in Polygons)
                yield return polygon;
        }
        if (Front != null)
        {
            lock (Front.Locker)
            {
                foreach (PolygonInfo allPolygon in Front.AllPolygons)
                    yield return allPolygon;
            }
        }
        if (Back != null)
        {
            lock (Back.Locker)
            {
                foreach (PolygonInfo allPolygon in Back.AllPolygons)
                    yield return allPolygon;
            }
        }
    }
}

public void Invert()
{
    if (!SplitPlane.HasValue)
    {
        return;
    }

    for (int index = 0; index < Polygons.Count; ++index)
        Polygons[index] = Polygons[index].Flip();
    Plane plane1 = SplitPlane.Value;
    Vector3 vector3 = -plane1.normal;
    Plane plane2 = SplitPlane.Value;
    double num = -plane2.distance;
```

```
            SplitPlane = new Plane(vector3, (float)num);

            lock (Locker)
            {
                Node front = Front;
                Front = Back;
                Back = front;
            }

            if (Front != null)
                Front.Invert();
            if (Back != null)
                Back.Invert();
        }

        public    IEnumerable<PolygonInfo>    ClipPolygons(IList<PolygonInfo>
polygons)
        {
            if (!SplitPlane.HasValue)
            {
                return polygons;
            }

            var frontList = new List<PolygonInfo>();
            var backList = new List<PolygonInfo>();
            foreach (PolygonInfo polygon in polygons)
            {

                SplitPlane.Value.SplitPolygon(polygon: polygon, coPlanarFront:
frontList, coPlanarBack: backList, front: frontList, back: backList, true);
            }

            if (Front != null)
                frontList = Front.ClipPolygons(frontList).ToList();
            if (Back != null)
                backList = Back.ClipPolygons(backList).ToList();
            else
                backList.Clear();
```

```
            var res = frontList.Concat(backList);
            return res;
        }
        public void CutWith(Node other)
        {
            if (Polygons.Count == 0)
                SplitPlane = null;
            lock (Locker)
                Polygons = other.ClipPolygons(Polygons).ToList();
            if (Front != null)
                Front.CutWith(other);
            if (Back != null)
                Back.CutWith(other);
        }

        private static PolygonInfo SelectSplitPlane(IEnumerable<PolygonInfo>
polygons)
        {
            int num = polygons.Count();
            switch (num)
            {
                case 0:
                    return null;
                case 1:
                    return polygons.First();
                default:
                    if (RandomStatic == null)
                        RandomStatic = new System.Random();
                    var res = polygons.Skip(RandomStatic.Next(0, num -
1)).First();
                    return res;
            }
        }

        private static void Build(Node node, IEnumerable<PolygonInfo> polygons)
        {
            if (!node.SplitPlane.HasValue)
            {
```

```
                PolygonInfo splitPolygon = SelectSplitPlane(polygons);
                if (splitPolygon == null)
                {
                    return;
                }
                lock (node.Locker)
                    node.SplitPlane = splitPolygon.Plane;
            }
            List<PolygonInfo>           frontPolys           =           new
List<PolygonInfo>(polygons.Count());
            List<PolygonInfo>           backPolys            =           new
List<PolygonInfo>(polygons.Count());
            lock (node.Locker)
            {
                foreach (PolygonInfo polygon in polygons)
                    node.SplitPlane.Value.SplitPolygon(polygon, coPlanarFront:
node.Polygons,   coPlanarBack:   node.Polygons,   front:   frontPolys,   back:
backPolys);
            }

            if (frontPolys.Count > 0)
            {
                if (node.Front == null)
                    lock (node.Locker)
                        node.Front = new Node();
                Build(node.Front, frontPolys);
            }
            if (backPolys.Count > 0)
            {
                if (node.Back == null)
                    lock (node.Locker)
                        node.Back = new Node();
                Build(node.Back, backPolys);

            }
        }
        private static void Build(object inf)
        {
            var info = (BuildInfo)inf;
```

```
            Build(info.Node, info.Polygons);
        }

        struct BuildInfo
        {
            public Node Node;
            public IEnumerable<PolygonInfo> Polygons;
            public Queue<KeyValuePair<Node, IEnumerable<PolygonInfo>>> ToDo;
        }

        public void Build(IEnumerable<PolygonInfo> polygons)
        {
            var todo = new Queue<KeyValuePair<Node,
IEnumerable<PolygonInfo>>>(100);
            todo.Enqueue(new KeyValuePair<Node,
IEnumerable<PolygonInfo>>(this, polygons));
            int num = 0;
            while (todo.Count > 0)
            {
                ++num;
                KeyValuePair<Node, IEnumerable<PolygonInfo>> keyValuePair =
todo.Dequeue();
                var info = new BuildInfo()
                {
                    Node = keyValuePair.Key,
                    Polygons = keyValuePair.Value,
                    ToDo = todo
                };
                Build(info);
            }
        }


        public Node Clone()
        {
            Node node = new Node();
            if (SplitPlane.HasValue)
                node.SplitPlane = new Plane?(SplitPlane.Value);
            if (Front != null)
```

```
                node.Front = Front.Clone();
            if (Back != null)
                node.Back = Back.Clone();
            node.Polygons.AddRange(Polygons.Select(a => a.Clone()));
            return node;
        }
    }
}
using System;
using UnityEngine;

namespace MyBoolean
{
    [Serializable]
    public class VertexInfo
    {
        public Vector3 Position;
        public Vector3 Normal;
        public Vector2 UV;
        public int id;

        public VertexInfo(Vector3 position, Vector3 normal, Vector2 UV1, int
id)
        {
            Position = position;
            Normal = Vector3.Normalize(normal);
            UV = UV1;
            this.id = id;
        }

        public VertexInfo Flip()
        {
            var res = new VertexInfo(Position, -Normal, UV, id);
            return res;
        }
        public VertexInfo Interpolate(VertexInfo other, float t)
        {
```

```
            var res = new VertexInfo(Vector3.Lerp(Position, other.Position, t),
Vector3.Lerp(Normal, other.Normal, t), Vector2.Lerp(UV, other.UV, t), id);
            return res;
        }
    }
}
using System;

namespace MyBoolean
{
    public enum BooleanType
    {
        Union,
        Intersection,
        Subtract,
    }
}
using UnityEngine;

namespace MyBoolean
{
    public struct PostionUVPair
    {
        public Vector3 Position;
        public Vector2 UV;
        public int ObjID;
        public int MaterialID;

        public PostionUVPair(Vector3 pos, Vector2 uv, int objID, int
materialID)
        {
            Position = pos;
            this.UV = uv;
            this.ObjID = objID;
            this.MaterialID = materialID;
        }
    }
}
```

```
using System.Collections.Generic;
using UnityEngine;
namespace MyBoolean
{
    public class ThreadData
    {
        public int start;
        public int end;
        public Matrix4x4 matrix;
        public Vector3[] vertices;
        public Vector3[] normals;
        public Vector2[] uv;
        public List<int> submesh;
        public int objID;
        public int previous;
        public int numberSubmesh;
        public object custom;
        public IEnumerable<PolygonInfo> polys;
        public IList<PolygonInfo> frontPolys;
        public IList<PolygonInfo> backPolys;
        public Node node;

        public ThreadData(int s, int e)
        {
            start = s;
            end = e;
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace MyBoolean
{
    [Serializable]
```

```
    public class PolygonInfo
    {
        public readonly Plane Plane;
        public VertexInfo[] Vertices;
        public int objID;
        public int materialID;

        public PolygonInfo(VertexInfo[] vertices, int objID, int materialID)
        {
            Vertices = vertices;
            this.objID = objID;
            this.materialID = materialID;
            Plane        =        new        Plane(vertices.First().Position,
vertices.Skip(1).First().Position, vertices.Skip(2).First().Position);
        }

        public Bounds Bounds
        {
            get
            {
                Bounds res = new Bounds();
                var value = false;
                var verts = Vertices.ToList().Select(a => a.Position);
                foreach (var current in verts)
                {
                    if (!value)
                    {
                        res = new Bounds(current, Vector3.zero);
                        value = true;
                    }
                    ref Bounds bound = ref res;
                    Vector3 vector3_1;
                    vector3_1.x = Math.Min(bound.min.x, current.x);
                    vector3_1.y = Math.Min(bound.min.y, current.y);
                    vector3_1.z = Math.Min(bound.min.z, current.z);
                    Vector3 vector3_2;
                    vector3_2.x = Math.Max(bound.max.x, current.x);
                    vector3_2.y = Math.Max(bound.max.y, current.y);
```

```
                vector3_2.z = Math.Max(bound.max.z, current.z);
                bound.SetMinMax(vector3_1, vector3_2);
            }
            return res;
        }
    }
    public PolygonInfo Flip()
    {
        VertexInfo v;
        int index;
        for (int i = 0; i < Vertices.Length / 2; i++)
        {
            v = Vertices[i].Flip();
            index = Vertices.Length - i - 1;
            Vertices[i] = Vertices[index].Flip();
            Vertices[index] = v;
        }
        if (Vertices.Length % 2 == 1)
            Vertices[Vertices.Length / 2] = Vertices[Vertices.Length / 
2].Flip();
        var res = new PolygonInfo(Vertices, objID, materialID);
        return res;
    }


    public PolygonInfo Clone()
    {
        var res = new PolygonInfo(Vertices, objID, materialID);
        return res;
    }
    public static bool IsDegenerateSet(VertexInfo[] set)
    {
        var uniqCount = 0;
        for (int i = 0; i < set.Length; i++)
        {
            var uniq = true;
            for (int j = 0; j < i; j++)
            {
```

```
                    if (set[i] == null)
                        break;
                    if (set[i] == set[j])
                    {
                        uniq = false;
                        break;
                    }
                }
                if (uniq)
                    uniqCount++;
            }
            var res = uniqCount < 3;
            return res;
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BooleanController : MonoBehaviour
{
    public GameObject result;

    public GameObject A;
    public GameObject B;

    public void Click()
    {
        var timer = new System.Diagnostics.Stopwatch();
        if (!result)
            result = new GameObject();
        result.name = "1";
        CreateMesh.CreateVertices(A, B, ref result);
        timer.Stop();
        Debug.Log("all = " + timer.ElapsedMilliseconds);
    }
}
```