Муниципальное Бюджетное Общеобразовательное Учреждение  
города Костромы Лицей №32

ИНДИВИДУАЛЬНЫЙ ИТОГОВЫЙ ПРОЕКТ

«Математика внутри видеоигры: сложное внутри простого»

Автор проекта:  
Иванов Артём Андреевич,  
ученик 9 «А» класса.

Руководитель проекта:  
Люстров Константин Сергеевич,  
учитель информатики и ИКТ.

Работа допущена к защите «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 202\_\_ г.

Подпись руководителя проекта \_\_\_\_\_\_\_\_ (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

Кострома  
2024-2025 уч. год

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ3

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ5

1.1. Алгоритмы обработки коллизий в двумерных видеоиграх5

1.1.1. Проверка нахождения точка внутри ориентированного по осям прямоугольника5

1.1.2. Проверка нахождения точки внутри многоугольника9

1.1.3. Проверка пересечения двух многоугольников11

1.2. Алгоритм корректировки диагонального движения в двумерных видеоиграх17

1.2.1. Нормализация вектора скорости17

1.3. Вспомогательные алгоритмы19

1.3.1. Генерация координат вершин многоугольника19

1.4. Архитектура двумерных видеоигр на языке JavaScript21

1.4.1. Холст и система координат21

1.4.2. Игровой цикл и отрисовка22

ГЛАВА 2. ПРАКТИЧЕСКАЯ ЧАСТЬ23

2.1. Реализация алгоритмов обработки коллизий в двумерных видеоиграх на языке JavaScript23

2.1.1. Реализация проверки нахождения точка внутри ориентированного по осям прямоугольника23

2.1.2. Реализация проверки нахождения точки внутри многоугольника24

2.1.3. Реализация проверки пересечения двух многоугольников29

2.2. Реализация алгоритмов корректировки диагонального движения в двумерных видеоиграх на языке JavaScript38

2.2.1. Реализация нормализации вектора скорости38

2.3. Реализация вспомогательных алгоритмов на языке JavaScript39

2.3.1. Реализация генерации координат вершин многоугольника39

2.4. Рекомендации для начинающих разработчиков двумерных видеоигр на языке JavaScript46

2.4.1. Рекомендации по обработке коллизий и движения объектов46

2.4.2. Общие рекомендации по разработке47

ЗАКЛЮЧЕНИЕ49

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ51

ВВЕДЕНИЕ

Многим нравится играть в видеоигры, но мало кто подозревает какие знания необходимы для разработки даже самых простых из них. Например, создание простой двумерной видеоигры может порой потребовать достаточное количество глубоких знаний в математике.

Независимо от уровня имеющихся под рукой технологий, начинающие разработчики часто сталкиваются с проблемой: при реализации своих идей им просто не хватает математических знаний. Это приводит к тому, что многие новички бросают программирование видеоигр или даже программирование в целом, ошибочно полагая, что им не хватает способностей. Однако на практике оказывается, что освоение ключевых математических концепций, часто используемых в разработке видеоигр, не только решает эту проблему, но и делает процесс разработки более понятным и доступным.

Поэтому целью данного проекта является создание компьютерной программы на языке JavaScript в виде двумерной видеоигры для демонстрации того, как освоение конкретных математических концепций решает типичные проблемы разработчиков при создании двумерных видеоигр.

Из-за нехватки математических знаний у начинающих разработчиков видеоигр часто возникают сложности при реализации коллизий и движения объектов. Исходя из этого для достижения указанной цели в проекте поставлены следующие задачи:

1. Исследовать и разработать на языке JavaScript следующие алгоритмы для обработки коллизий в двумерных видеоиграх:

а) Проверка нахождения точки внутри ориентированного по осям прямоугольника.

б) Проверка нахождения точки внутри.

в) Проверка пересечения двух многоугольников.

2. Исследовать и разработать на языке JavaScript алгоритм нормализации векторов скорости для корректного диагонального движения.

3. Разработать на языке JavaScript вспомогательные математические алгоритмы для реализации дополнительного функционала.

4. Создать простую двумерную видеоигру на языке JavaScript, используя разработанные алгоритмы, чтобы продемонстрировать работоспособность алгоритмов и упрощение процесса разработки.

5. Предложить рекомендации для начинающих разработчиков видеоигр.

Основным объектом исследования в проекте выступают математические методы обработки коллизий и движения в двумерных видеоиграх, а предметом исследования - применение конкретных алгоритмов из векторной алгебры и вычислительной геометрии в игровых механиках.

В проекте применяются следующие методы исследования:

1. Теоретический анализ математических основ игровых механик.

2. Практическая реализация алгоритмов на JavaScript.

3. Сравнительный анализ различных подходов к обработке коллизий.

4. Эмпирическая проверка решений в рабочем прототипе видеоигры.

Теоретическая значимость проекта заключается в систематизации математических методов обработки коллизий и движения, а также в разработке методики их применения в игровой разработке. Практическая ценность проекта проявляется в создании библиотеки готовых алгоритмов на JavaScript и демонстрационном игровом прототипе, которые наглядно показывают эффективность математического подхода к решению типичных задач при разработке двумерных видеоигр.

Продуктом данного проекта является интерактивная двумерная видеоигра, написанная на языке JavaScript, демонстрирующая применение математических алгоритмов в реальных игровых механиках. Видеоигра доступна для тестирования по адресу: <https://wisp80.github.io/project-9th-grade>, а полный исходный код, включающий реализацию всех исследованных алгоритмов, опубликован в открытом репозитории: <https://github.com/Wisp80/project-9th-grade>.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. Алгоритмы обработки коллизий в двумерных видеоиграх

При разработке двумерных видеоигр редко возникают проблемы, когда необходимо просто двигать объекты. Но когда стоит задача заставить объекты взаимодействовать друг с другом, например, чтобы персонаж не проходил сквозь стены, то без специальных познаний в математике тут уже не обойтись.

В этом параграфе будет исследовано несколько популярных алгоритмов, используемых при обработке коллизий объектов в двумерных видеоиграх.

1.1.1. Проверка нахождения точки внутри ориентированного по осям прямоугольника

Наверное, самыми распространенными формами объектов в двумерных видеоиграх являются точка и прямоугольник, и поэтому часто приходится использовать алгоритм, проверяющий коллизию объектов таких форм.

Алгоритм проверки нахождения точки внутри ориентированного по осям прямоугольника достаточно интуитивен. Для начала определим, что ориентированным по осям прямоугольником является прямоугольник, стороны которого параллельны осям координат (Рис. 1).

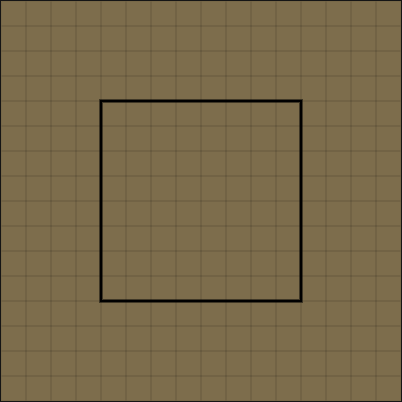
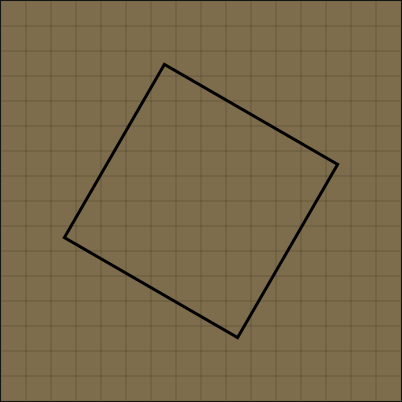
 

Рис. 1 - Ориентированный по осям и не ориентированный по осям прямоугольники

Чтобы понять, как работает этот алгоритм, проведем через стороны прямоугольника прямые, которые будут создавать две полуплоскости, а затем через пересечение определенных полуплоскостей найдем область, занимаемую прямоугольником (Рис. 2).

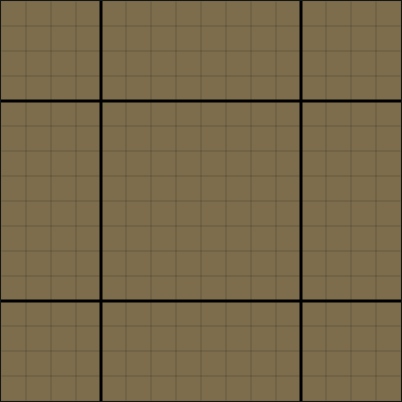


Рис. 2 - Через стороны прямоугольника проведены прямые

Начиная с верхней прямой берем по часовой стрелке каждую прямую, определяем в какой полуплоскости относительно взятой прямой находится прямоугольник, а в конце путем пересечения выбранных полуплоскостей определяем область, которую занимает прямоугольник.

В зависимости от взятой прямой прямоугольник будет находится в следующих полуплоскостях:

1. При верхней прямой прямоугольник окажется в нижней полуплоскости.

2. При правой прямой прямоугольник окажется в левой полуплоскости.

3. При нижней прямой прямоугольник окажется в верхней полуплоскости.

4. При левой прямой прямоугольник окажется в правой полуплоскости.

Пошаговое пересечение выбранных полуплоскостей будет постепенно формировать область, занимаемую прямоугольником (Рис. 3).

Поскольку область, занимаемая прямоугольником, получается из пересечения выбранных полуплоскостей, то и общее условие для того, чтобы точка была внутри этого прямоугольника, будет получаться из объединения всех отдельных условий для каждой полуплоскости.

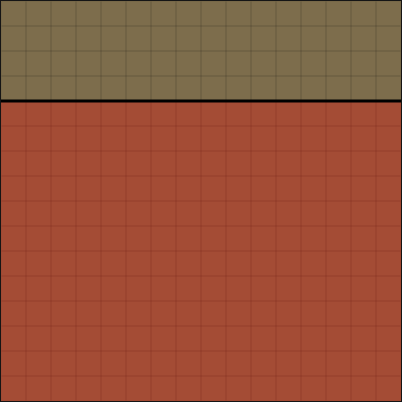
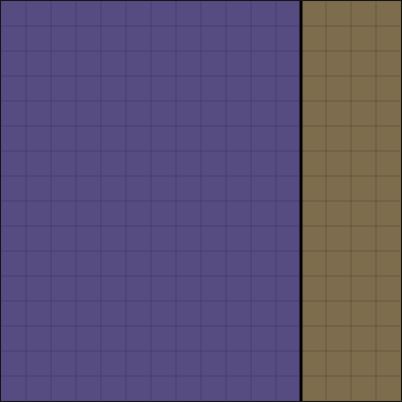
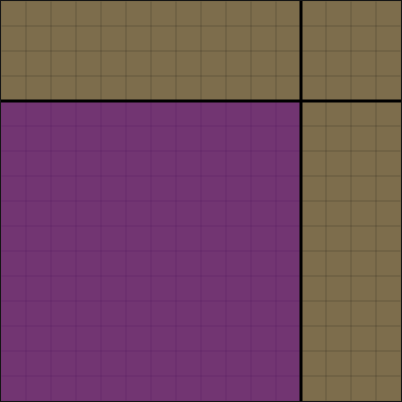
То есть для того, чтобы точка считалась внутри прямоугольника:

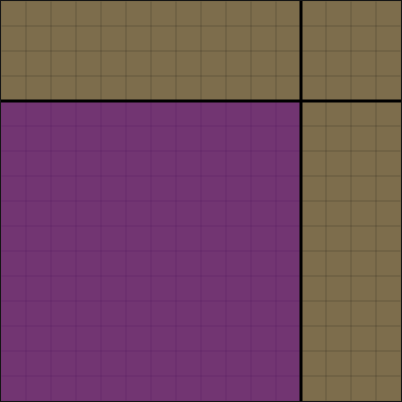
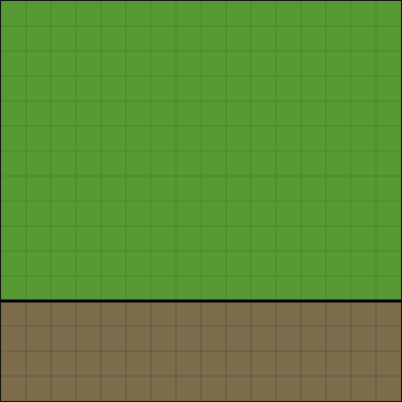
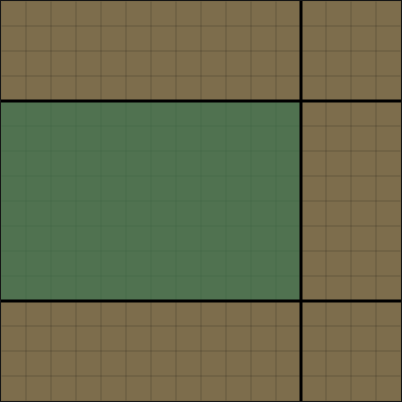
1) Первая полуплоскость будет говорить, что Y-координата точки должна быть меньше или равна максимальной Y-координате прямоугольника.

2) Вторая полуплоскость будет говорить, что X-координата точки должна быть меньше или равна максимальной X-координате прямоугольника.

3) Третья полуплоскость будет говорить, что Y-координата точки должна быть больше или равна минимальной Y-координате прямоугольника.

4) Четвертая полуплоскость будет говорить, что X-координата точки должна быть больше или равна минимальной X-координате прямоугольника.

 +  = 

 +  = 

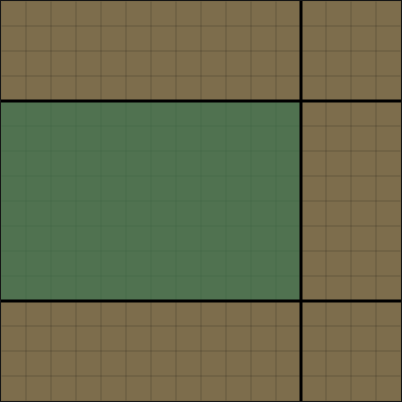
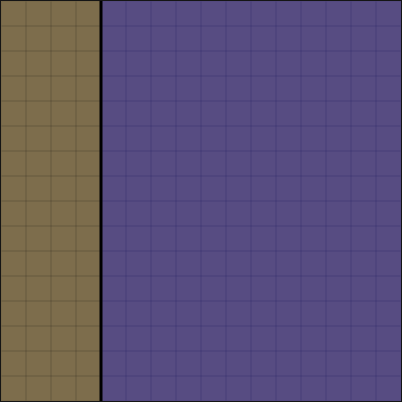
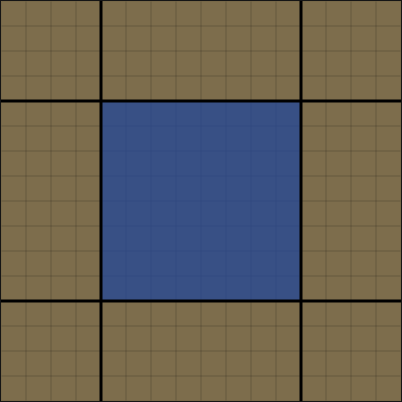
 +  = 

Рис. 3 - Процесс пересечения полуплоскостей

Весь процесс формирования области, занимаемой прямоугольником, продемонстрирован на Рис. 4.

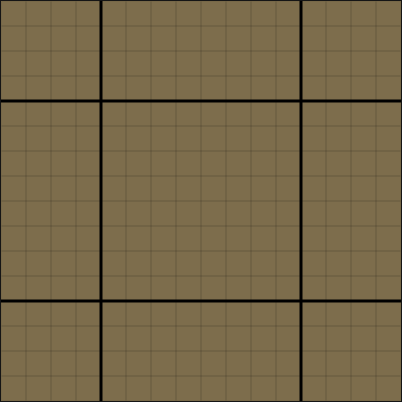


Рис. 4 - Весь процесс формирования области, занимаемой прямоугольником

В итоге алгоритм проверки нахождения точки внутри ориентированного по осям прямоугольника можно сформулировать следующим образом:

Точка A(xa, ya) находится внутри ориентированного по осям прямоугольника, если соблюдается условие:

где:

1. *xmax* - максимальная X-координата прямоугольника.

2. *xmin* - минимальная X- координата прямоугольника.

3. *ymax* - максимальная Y- координата прямоугольника.

4. *ymin* - минимальная Y- координата прямоугольника.

1.1.2 Проверка нахождения точки внутри многоугольника

Точка и прямоугольник позволяют создавать большое количество объектов в двумерной видеоигре, но когда требуется более сложная геометрия, например, для создания камней неправильной формы, то уже могут потребоваться многоугольники.

При использовании многоугольников в двумерной видеоигре часто приходится определять коллизию между объектами в форме точки и объектами в форме многоугольника. Например, задевает ли пуля камень. На такие случаи нужно иметь под рукой алгоритм проверки нахождения точки внутри многоугольника.

Рассмотрим алгоритм проверки нахождения точки внутри многоугольника, который основан на методе луча.

Метод луча работает по следующему принципу:

1. Из проверяемой точки выпускается луч, обычно горизонтально вправо.

2. Подсчитывается количество пересечений луча со сторонами многоугольника.

3. Если подсчитанное количество пересечений нечетное, то это означает, что точка находится внутри многоугольника, иначе - точка находится снаружи многоугольника.

Этот алгоритм звучит достаточно просто и хорошо справляется с базовыми случаями (Рис. 5). Но одна из сложностей использования этого алгоритма состоит в том, что при его реализации нужно не забывать учитывать нестандартные случаи пересечения луча и сторон многоугольника, которых может оказаться не так уж и мало (Рис. 6).

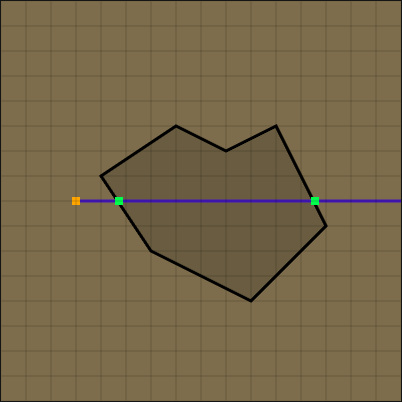
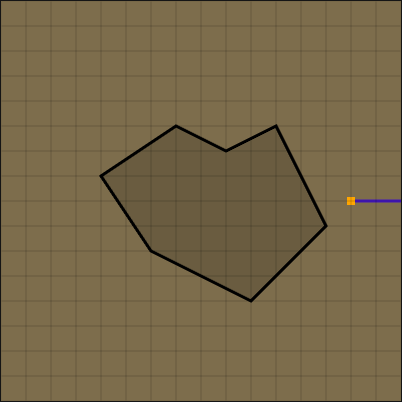
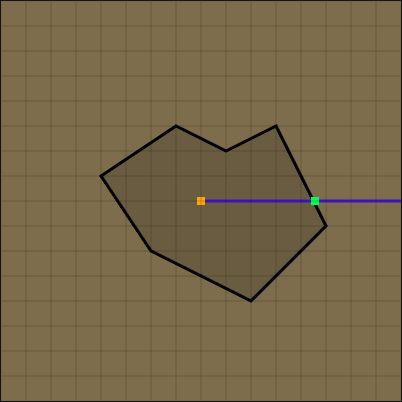
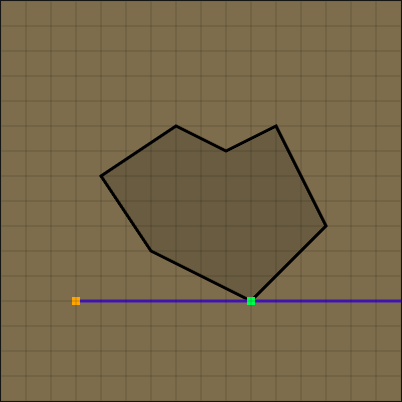
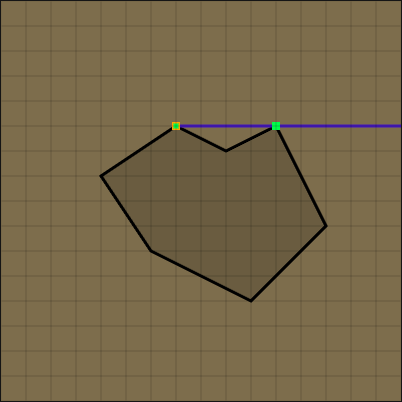
  

Рис. 5 - Базовые случаи пересечения луча и сторон многоугольника

Что если точка находится снаружи, а луч имеет одно пересечение с одной из вершин многоугольника? Что если точка находится на одной из вершин многоугольника, а луч имеет пересечение с двумя вершинами многоугольника? Что если точка находится снаружи, а луч имеет три пересечения с многоугольником - с вершиной и двумя сторонами многоугольника? И в конце концов что делать если луч совпадает с одной из сторон многоугольника, независимо, где находится точка? И это не полный список таких уникальных случаев пересечения луча и сторон многоугольника.

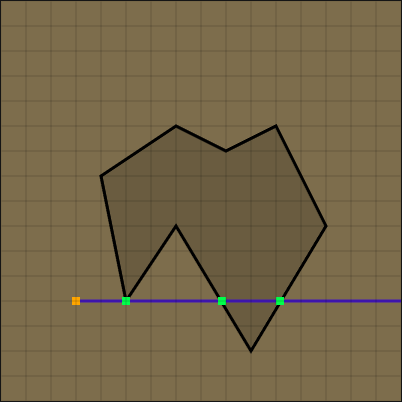
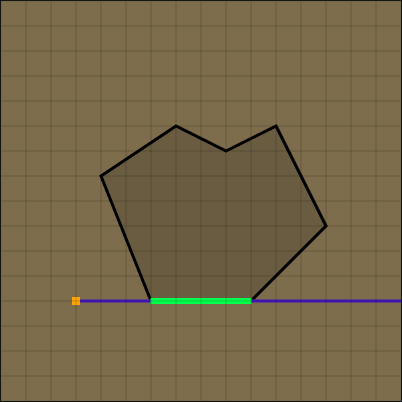
 

Рис. 6 - Нестандартные случаи пересечения луча и сторон многоугольника

Еще одна из сложностей использования этого алгоритма состоит в выборе способа регистрации пересечений луча и сторон многоугольника. Поскольку в проекте был выбран способ, использующий линейную интерполяцию, то про нее стоит кратко рассказать здесь.

При реальном использовании алгоритма нам нужен не весь луч, а только лишь отрезок луча, ограниченный экраном. То есть нам нужно знать все точки на отрезке луча и на каждой стороне многоугольника. Для этого можно использовать линейную интерполяцию.

Линейная интерполяция является методом нахождения промежуточных значений между двумя известными точками на прямой линии.

Общий порядок работы линейный интерполяции такой:

1. Вычисляется длина отрезка между двумя конечными точками отрезка.

2. Определяется количество шагов интерполяции, то есть количество точек на отрезке.

3. Проходя по шагам интерполяции, вычисляются координаты точек на отрезке.

Про конкретную реализацию выбранного способа регистрации пересечений луча и сторон многоугольника, и всего алгоритма будет рассказано в практической части проекта.

1.1.3. Проверка пересечения двух многоугольников

Еще одним распространенным видом коллизий в двумерных видеоиграх является коллизия объектов в форме многоугольников, например, когда нужно проверить, что персонаж касается врага.

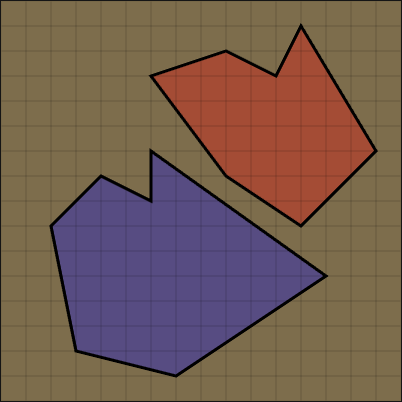
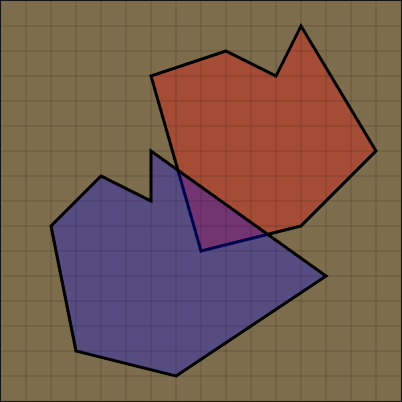
Алгоритм проверки пересечения двух многоугольников заключается в следующем: если хотя бы одна сторона одного многоугольника пересекается со стороной другого, то это означает, что эти многоугольники пересекаются.

Используя этот алгоритм, нужно всегда учитывать пару нестандартных случаев пересечения двух многоугольников:

1) Один многоугольник полностью находится внутри другого.

2) Два многоугольника полностью совпадают.

В итоге этот алгоритм сводится к четырем основным случаям (Рис. 7).

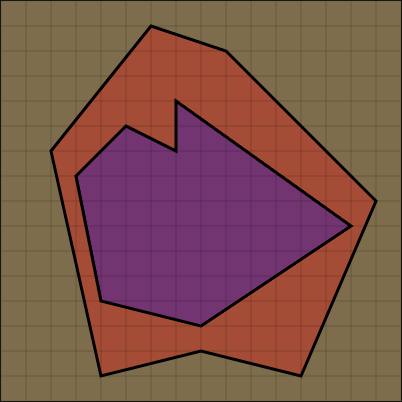
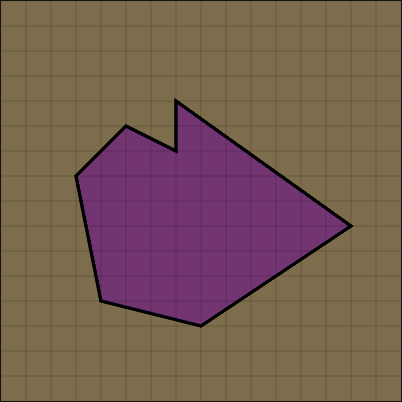
 

Рис. 7 - Основные случаи пересечения многоугольников

Сложность этого алгоритма заключается в его реализации, так как она требует следующих знаний из математики и алгоритмов:

1. Расчет векторного произведения двух векторов.

2. Расчет скалярного произведения двух векторов.

3. Расчет квадрата длины вектора.

4. Геометрический смысл векторного произведения двух векторов.

5. Геометрический смысл скалярного произведения двух векторов.

6. Алгоритм проверки нахождения точки на отрезке.

7. Алгоритм проверки пересечения двух отрезков.

Разберем каждый пункт отдельно, чтобы не было проблем при рассмотрении конкретной реализации этого алгоритма в практической части проекта.

Для двух векторов AB и CD векторное произведение вычисляется по формуле:

Для двух векторов AB и CD скалярное произведение вычисляется по формуле:

 Квадрат длины вектора AB равен скалярному произведению двух таких векторов AB:

Геометрический смысл векторного произведения двух векторов, на примере точки P и прямой, содержащей отрезок AB (Рис. 8), звучит так:

1. Если векторное произведение двух векторов AB и AP положительное, то это означает, что:

а) Точка P находится слева от прямой, содержащей отрезок AB.

б) Вектор AP «повернут» относительно вектора AB против часовой стрелки.

2. Если векторное произведение двух векторов AB и AP отрицательное, то это означает, что:

а) Точка P находится справа от прямой, содержащей отрезок AB.

б) Вектор AP «повернут» относительно вектора AB по часовой стрелке.

3. Если векторное произведение двух векторов AB и AP равно нулю, то это означает, что:

а) Точка P лежит на прямой, содержащей отрезок AB.

б) Векторы AB и AP коллинеарные, то есть лежат на одной прямой.

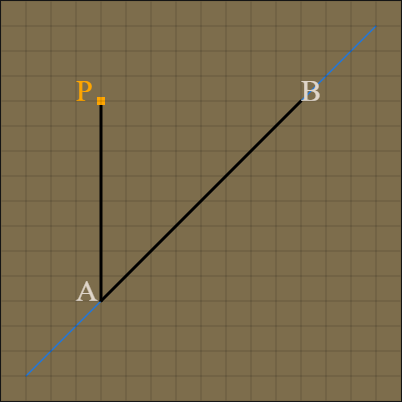
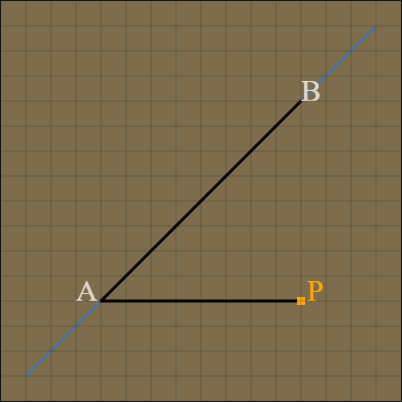
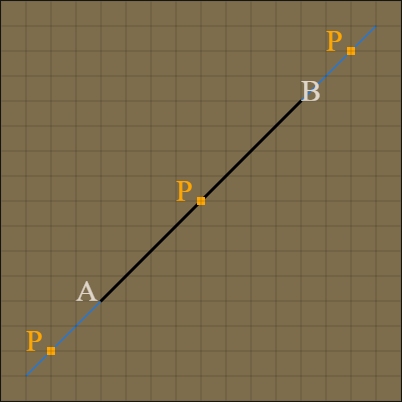
  

Рис. 8 - Геометрический смысл векторного произведения двух векторов

Геометрический смысл скалярного произведения двух векторов, на примере векторов AB и AP (Рис. 9), звучит так:

1. Если скалярное произведение двух векторов AB и AP положительное, то это означает, что угол между векторами острый, то есть векторы направлены «в целом в одну сторону».

2. Если скалярное произведение двух векторов AB и AP отрицательное, то это означает, что угол между векторами тупой, то есть векторы направлены «в целом в противоположные стороны».

3. Если скалярное произведение двух векторов AB и AP равно нулю, то это означает, что векторы перпендикулярны.

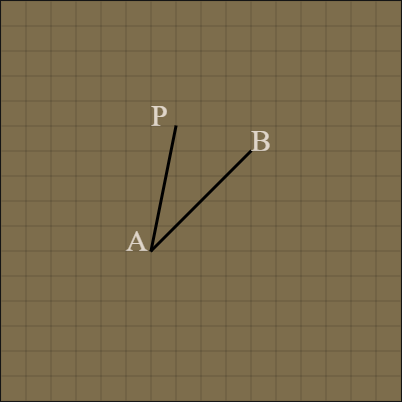
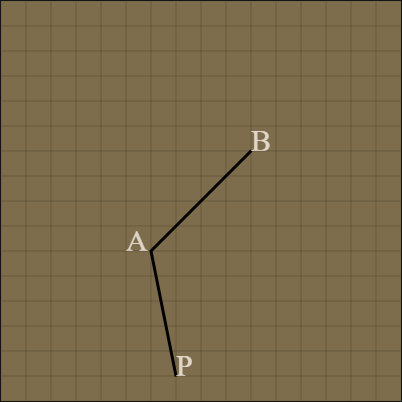
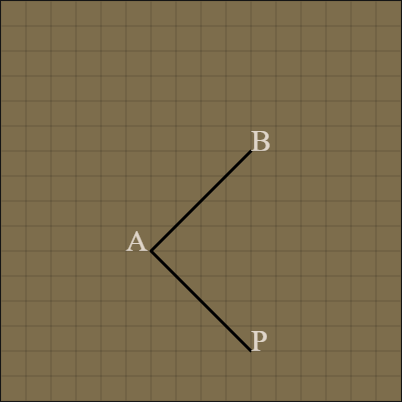
  

Рис. 9 - Геометрический смысл скалярного произведения двух векторов

Алгоритм проверки нахождения точки на отрезке (Рис. 10) заключается в следующем:

1. При помощи геометрического смысла векторного произведения двух векторов проверяется лежит ли точка на прямой, содержащей отрезок.

2. При помощи геометрического смысла скалярного произведения двух векторов проверяется не находится ли точка до «первой» конечной точки отрезка, то есть за пределами отрезка.

3. При помощи геометрического смысла скалярного произведения двух векторов и квадрата длины вектора проверяется не находится ли точка дальше «второй» конечной точки отрезка, то есть за пределами отрезка.

4. Если точка оказалась на прямой содержащей отрезок и находится между его конечными точками, то это означает, что точка находится на отрезке.

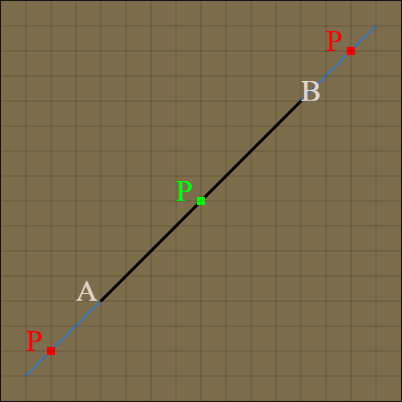


Рис. 10 - Процесс нахождения точки на отрезке

Алгоритм проверки пересечения двух отрезков сводится к общему случаю и частным случаям (Рис. 11).

Общий случай пересечения двух отрезков звучит так: два отрезка пересекаются, если конечные точки первого отрезка лежат по разные стороны от прямой, содержащей второй отрезок, а конечные точки второго отрезка лежат по разные стороны от прямой, содержащей первый отрезок. Для реализации проверки общего случая используется геометрический смысл векторного произведения двух векторов.

Частные случаи пересечения двух отрезков являются случаями, когда одна из точек одного отрезка лежит на другом отрезке. Для реализации проверки частных случаев используется как геометрический смысл векторного произведения двух векторов, так и алгоритм проверки нахождения точки на отрезке.

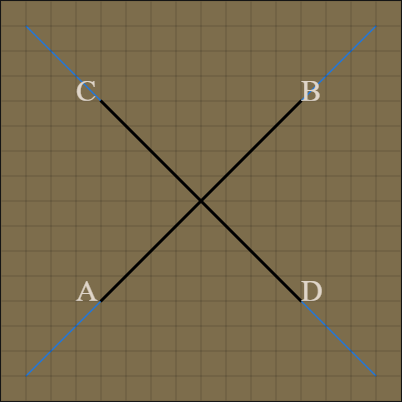
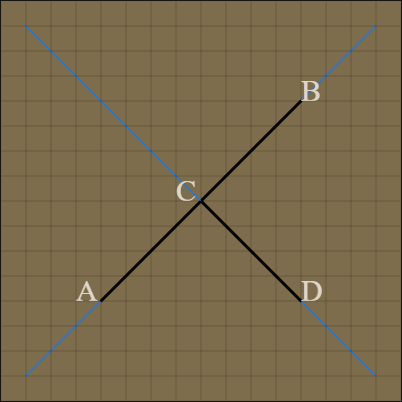
 

Рис. 11 - Общий случай и пример частного случая пересечения двух отрезков

Как именно используется указанные знания из математики и алгоритмы будет рассмотрено в конкретной реализации этого алгоритма в практической части проекта.

1.2. Алгоритм корректировки диагонального движения в двумерных видеоиграх

Не редкий случай, когда в двумерной видеоигре можно обнаружить, что персонаж двигается по диагонали быстрее, чем должен. Такое происходит из-за недосмотра разработчиков, которые забывают корректировать скорость при диагональном движении.

В этом параграфе будет исследовано почему возникает указанная проблема и как ее решить при помощи нормализации вектора скорости.

1.2.1. Нормализация вектора скорости

Самым простым вариантом реализации движения объектов в двумерной видеоигре является отслеживание нажатых кнопок перемещения и изменение координат объекта. Например, если скорость объекта равна 5 и была нажата кнопка вправо, то X-координата объекта увеличится на 5, а если была нажата кнопка влево - уменьшится на 5. При такой системе координаты объекта меняются поступательно, то есть, например, при нажатии кнопок влево и вверх объект сначала сдвигается по оси X, а после по оси Y.

Но при такой системе скорость диагонального движения оказывается больше, чем должна быть (Рис. 12). Дело в том, что в данном случае вектор скорости при перемещении из точки A в точку B будет рассчитываться по теореме Пифагора. Например, если скорость объекта равна 5, то его скорость диагонального движения будет равна , а должна быть равна 5. Если построить окружность с центром в точке A с радиусом равному скорости объекта, то будет видно, что объект при диагональном движении не должен выходить за пределы этой окружности.

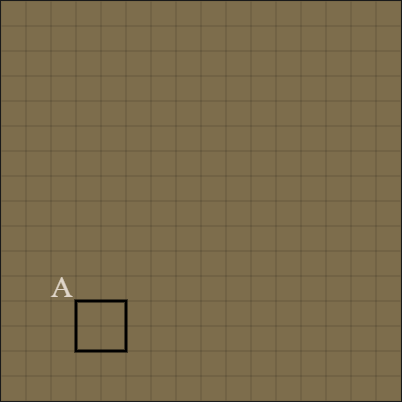
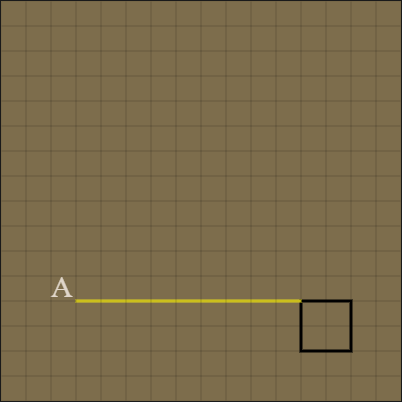
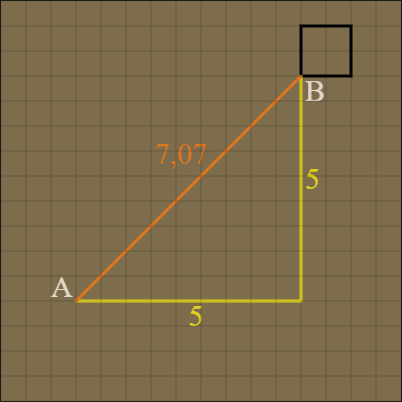
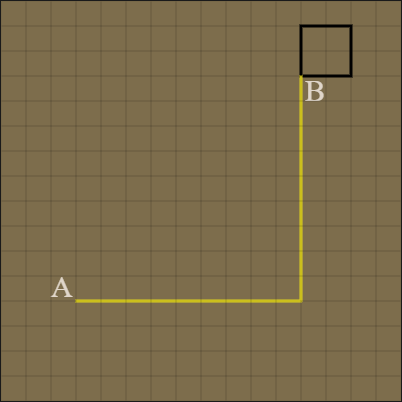
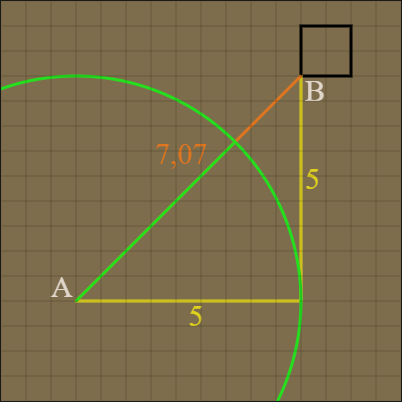
   

Рис. 12 - Диагональное движение с некорректным вектором скорости

Чтобы скорость диагонального движения была корректной необходимо использовать алгоритм, основанный на нормализации вектора скорости, то есть приведении вектора к единичной длине, сохраняя его направление:

1. Найти текущую (некорректную) величину скорости диагонального движения путем нахождения длины вектора этой скорости по теореме Пифагора.

2. Найти компоненты нормализованного вектора скорости. Это делается путем деления каждой длины компоненты некорректного вектора, то есть осевых проекций вектора, на длину некорректного вектора.

3. Компоненты нормализованного вектора скорости умножить на требуемую скорость, чтобы получить компоненты корректного вектора скорости.

4. Полученные компоненты будут сообщать на сколько нужно пошагово сдвинуть объект по осям, чтобы объект прошел корректное расстояние.

Такой алгоритм нормализации вектора скорости часто используется, например, при обработке движения игрока или врагов.

1.3. Вспомогательные алгоритмы

При разработке видеоигр часто возникают специфические задачи, требующие создания специализированных математических алгоритмов. В данном параграфе рассматривается алгоритм генерации вершин многоугольника в заданной области, который был разработан для решения конкретной задачи в проекте.

1.3.1. Генерация координат вершин многоугольника

Для реализации объектов произвольной формы был разработан алгоритм, создающий координаты вершин многоугольника в заданной области (Рис. 13):

1. Заданная область условно делится на четверти. Первая четверть находится справа и сверху, остальные три четверти идут по часовой стрелке.

2. Заданное количество вершин примерно делится поровну на четыре порции, где каждая порция вершин относится к каждой четверти.

3. Первая вершина генерируется в верхней центральной части первой четверти.

4. Остальные вершины генерируются по часовой стрелке путем случайных сдвигов предыдущей вершины. Как именно происходит сдвиг зависит от заданного размера сдвига и от того, к какой четверти относятся вершины:

1) В первой четверти сдвиг идет вправо и вниз.

2) Во второй четверти сдвиг идет влево и вниз.

3) В третьей четверти сдвиг идет влево и вверх.

4) В четвертой четверти сдвиг идет вправо и вверх.

При работе с этим алгоритмом нужно всегда помнить, что качество его работы во многом зависит от адекватно подобранных параметров. Например, если указать маленькую область и большое количество вершин, то в итоге, скорее всего, всегда будет получаться прямоугольник.

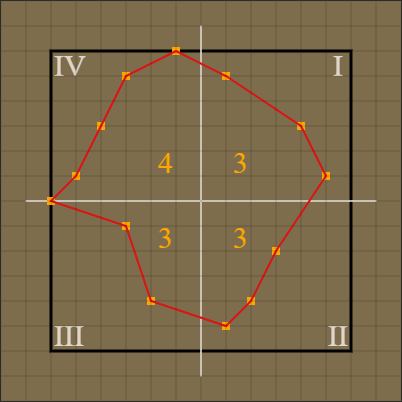


Рис. 13 - Образный пример генерации вершин многоугольника из 13 вершин

1.4. Архитектура двумерных видеоигр на языке JavaScript

Прежде чем перейти к практической части проекта, нужно рассмотреть ключевые компоненты, из которых состоит видеоигра на языке JavaScript.

1.4.1. Холст и система координат

Для работы с графикой язык JavaScript предоставляет специальный элемент - холст (canvas). Вот неполный перечень возможностей, которые предоставляет этот элемент:

1. Отрисовка примитивов, например, отрезков, дуг, прямоугольников, окружностей или текста.

2. Заливка и обводка примитивов.

3. Создание градиентов.

Полный список возможностей гораздо больше. Инструментарий холста позволяет создавать анимации, видеоигры и любые другие приложения. Например, все рисунки в теоретической части данного проекта были созданы при помощи холста из JavaScript.

При работе с холстом из JavaScript (Рис. 14) нужно всегда помнить, что начало координат (0, 0) находится в верхнем левом углу, а ось Y идет вниз, а не вверх. Хоть холст и позволяет трансформировать систему координат, в проекте используется стандартная система координат холста из JavaScript.

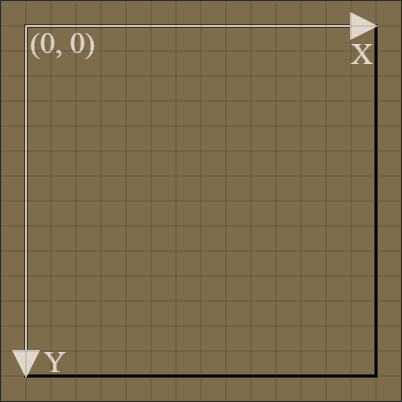
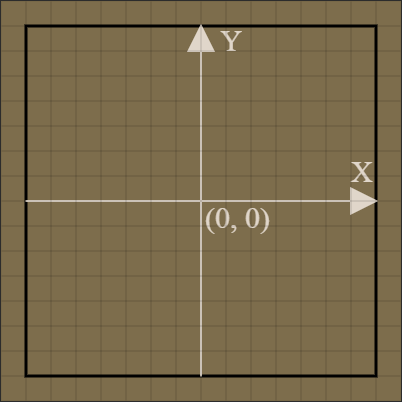
 

Рис. 14 - Отличие системы координат холста из JavaScript от «классической»

1.4.2. Игровой цикл и отрисовка

Видеоигра это не просто одно статичное изображение, а постоянная смена кадров. Чтобы понять, как работает такой процесс нужно знать, что такое игровой цикл. Когда один кадр сменяет предыдущий, то это означает, что прошел ровно один игровой цикл. Каждый игровой цикл состоит из двух основных фаз: подготовка кадра и отрисовка кадра.

В первой фазе осуществляется пересчет данных игровой логики, например, таких как:

1. Обновление позиций объектов.

2. Обработка коллизий: удаление объектов при их столкновении.

3. Генерация новых объектов: создание новых врагов.

4. Обновление игровых параметров: очки здоровья, номер уровня.

5. Проверка условий победы и поражения.

6. Многие другие игровые данные, которые изменились с предыдущего кадра.

Во второй фазе данные обновленной игровой логики «отправляются» холсту, чтобы он понимал что, где и как необходимо отрисовать.

После этого начинается следующий игровой цикл и так пока не сработает какой-то триггер, прерывающий создание новых игровых циклов, или пока пользователь не закроет приложение.

Самыми распространенными способами генерации игровых циклов на языке JavaScript является использование функций «setInterval()» и «setTimeout()», или использование функции «requestAnimationFrame()». Последнее является более предпочтительным вариантом, так как функция «requestAnimationFrame()» гораздо более оптимизирована и синхронизирует игровые циклы с частотой обновления экрана.

ГЛАВА 2. ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1. Реализация алгоритмов обработки коллизий в двумерных видеоиграх на языке JavaScript

В этом параграфе будут рассмотрены функции на языке JavaScript, созданные с целью реализации алгоритмов обработки коллизий, исследованных в теоретической части проекта. В параграфе будут показаны примеры кода с комментариями, но для более детального понимания можно обратиться к коду всей видеоигры.

2.1.1. Реализация проверки нахождения точки внутри ориентированного по осям прямоугольника

Для реализации алгоритма проверки нахождения точки внутри ориентированного по осям прямоугольника была создана функция «isPointInsideNotRotatedRectangle()». Поскольку этот алгоритм заключается в проверке условия, указанного в конце пункта 1.1.1., то главной целью этой функции является перенесение этого условия на язык JavaScript (Рис. 15).

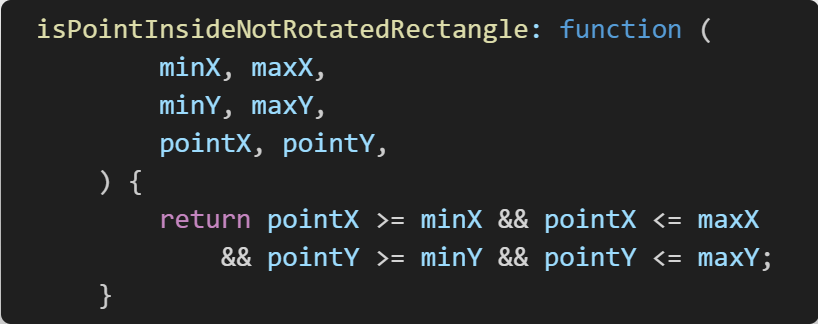


Рис. 15 - Код функции «isPointInsideNotRotatedRectangle()»

Функция «isPointInsideNotRotatedRectangle()» принимает следующие параметры:

1. «minX» - минимальная X-координата прямоугольника.

2. «maxX» - максимальная X-координата прямоугольника.

3. «minY» - минимальная Y-координата прямоугольника.

4. «maxY» - максимальная Y-координата прямоугольника.

5. «pointX» - X-координата точки.

6. «pointY» - Y-координата точки.

Функция «isPointInsideNotRotatedRectangle()» возвращает true, если точка находится внутри ориентированного по осям прямоугольника, иначе false.

2.1.2. Реализация проверки нахождения точки внутри многоугольника

Поскольку для данной реализации алгоритма проверки нахождения точки внутри многоугольника требуется использование линейной интерполяции, то была создана вспомогательная функция «findIntPointsOnLineSegment()», которая находит точки на отрезке при помощи линейной интерполяции (Рис. 16).

Функция «findIntPointOnLineSegment()» принимает следующие параметры:

1. «x1» - X-координата первой конечной точки отрезка.

2. «y1» - Y-координата первой конечной точки отрезка.

3. «x2» - X-координата второй конечной точки отрезка.

4. «y2» - Y-координата второй конечной точки отрезка.

Функция «findIntPointOnLineSegment()» возвращает массив объектов, содержащих целочисленные координаты найденных точек на отрезке.



Рис. 16 - Код функции «findIntPointOnLineSegment()»

Для регистрации пересечений отрезка луча и сторон многоугольника, нужно знать точки на отрезке луча и на сторонах многоугольника. Для того, чтобы найти точки на сторонах многоугольника была создана функция «findIntPointsOnPolygonEdges()» (Рис. 17).

Функция «findIntPointsOnPolygonEdges()» принимает параметр «vertices», который является массивом объектов, содержащих целочисленные координаты последовательно указанных вершин многоугольника.

Функция «findIntPointsOnPolygonEdges()» возвращает массив объектов, содержащих целочисленные координаты точек на каждой стороне многоугольника.



Рис. 17 - Код функции «findIntPointsOnPolygonEdges()»

Для итоговой реализации алгоритма проверки нахождения точки внутри многоугольника была создана функция «isPointInsidePolygon()».

Функция «isPointInsidePolygon()» принимает следующие параметры:

1. «point» - объект, содержащий целочисленные координаты точки.

2. «vertices» - массив объектов, содержащих целочисленные координаты последовательно указанных вершин многоугольника.

Функция «isPointInsidePolygon()» возвращает true, как знак того, что точка находится внутри многоугольника, иначе false.

Логика работы функции «isPointInsidePolygon()» следующая:

1. Проверяем не совпадает ли точка с одной из вершин многоугольника (Рис. 18).

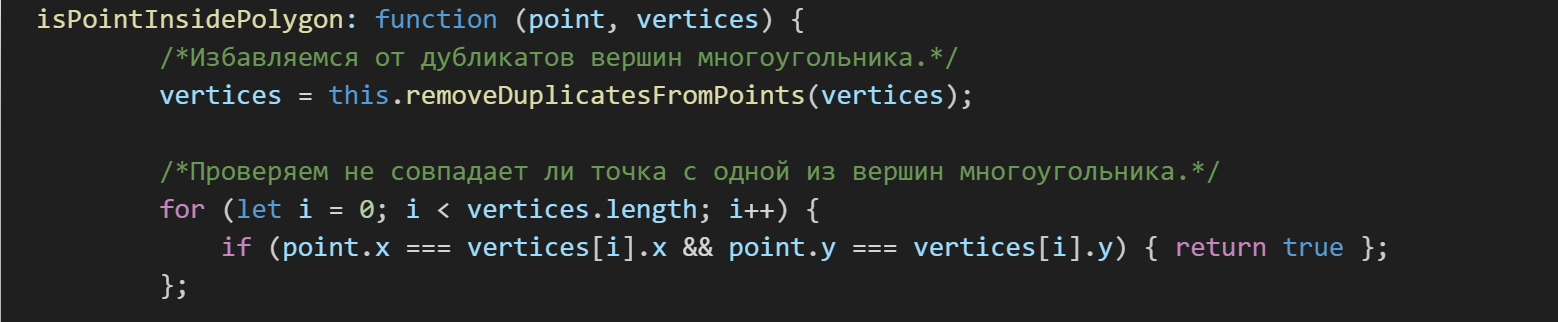


Рис. 18 - Шаг 1 функции «isPointInsidePolygon()»

2. Проверяем не находится ли точка снаружи «ограничивающей коробки» многоугольника (Рис. 19).

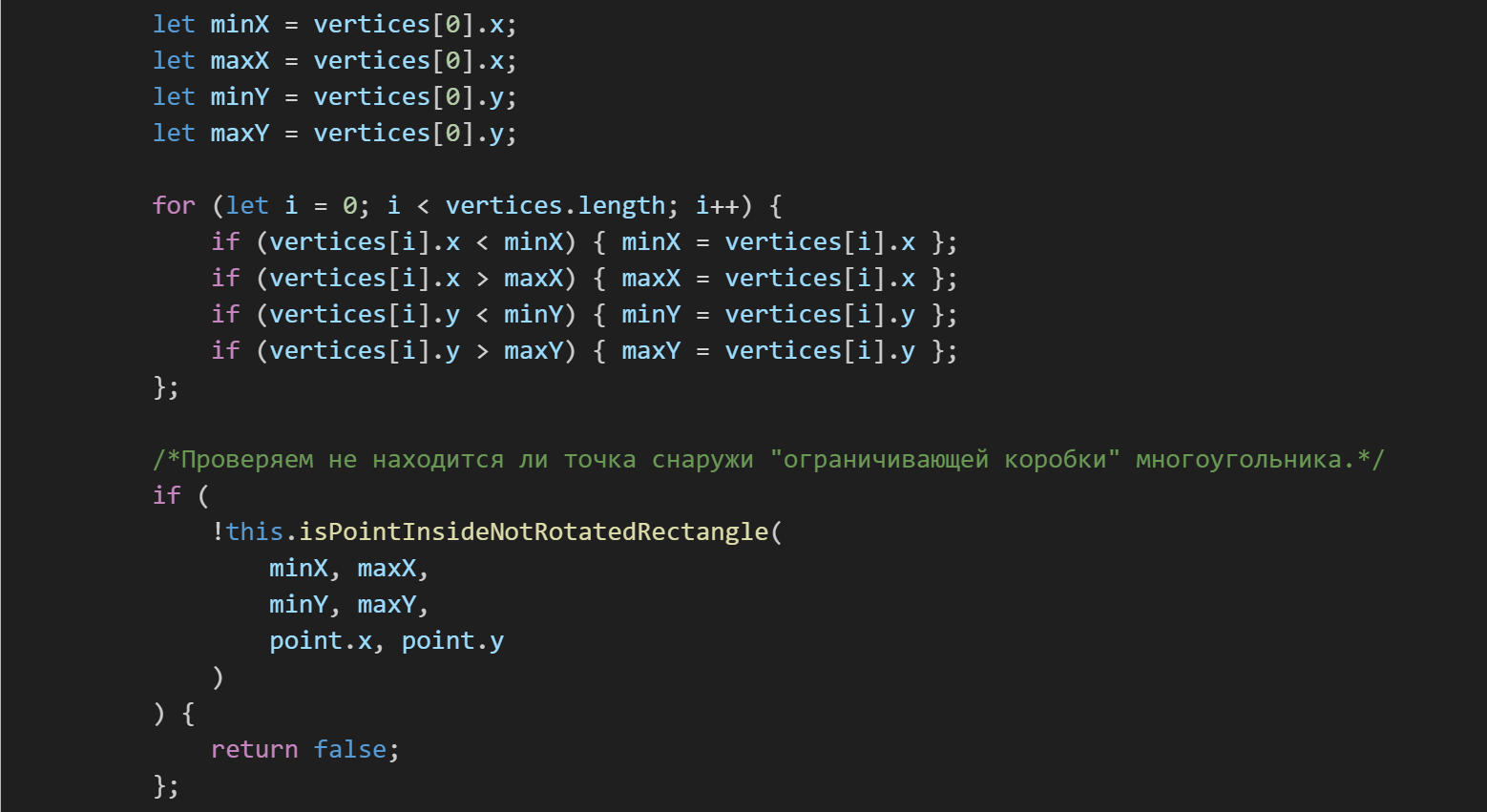


Рис. 19 - Шаг 2 функции «isPointInsidePolygon()»

3. Находим точки на отрезке луча (Рис. 20).

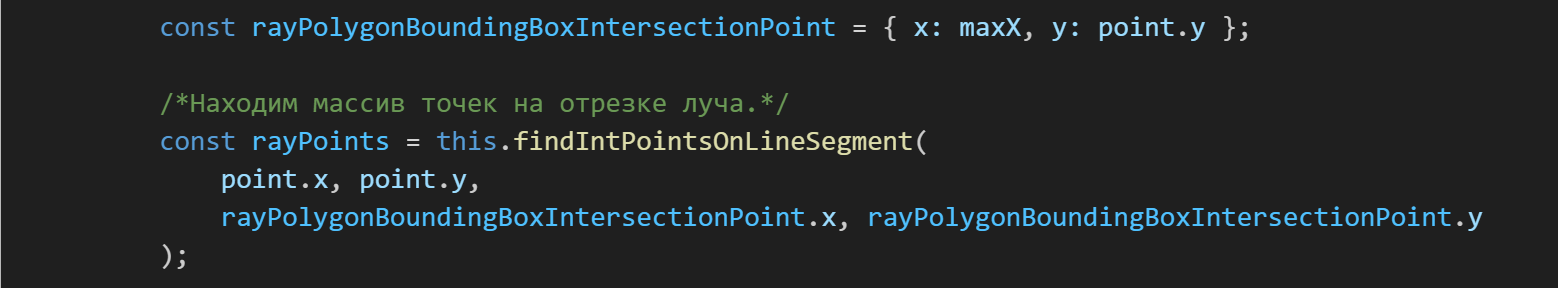


Рис. 20 - Шаг 3 функции «isPointInsidePolygon()»

4. Находим точки на каждой стороне многоугольника (Рис. 21).

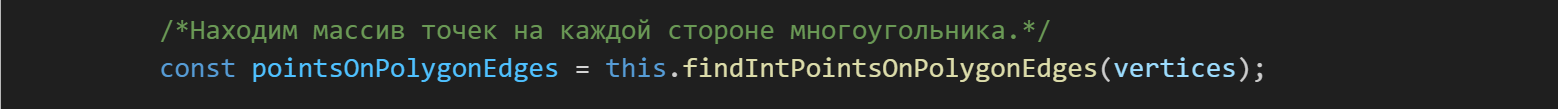


Рис. 21 - Шаг 4 функции «isPointInsidePolygon()»

5. Перебираем точки на отрезке луча и точки на сторонах многоугольника, сохраняя все совпадающие точки в массив (Рис. 22).

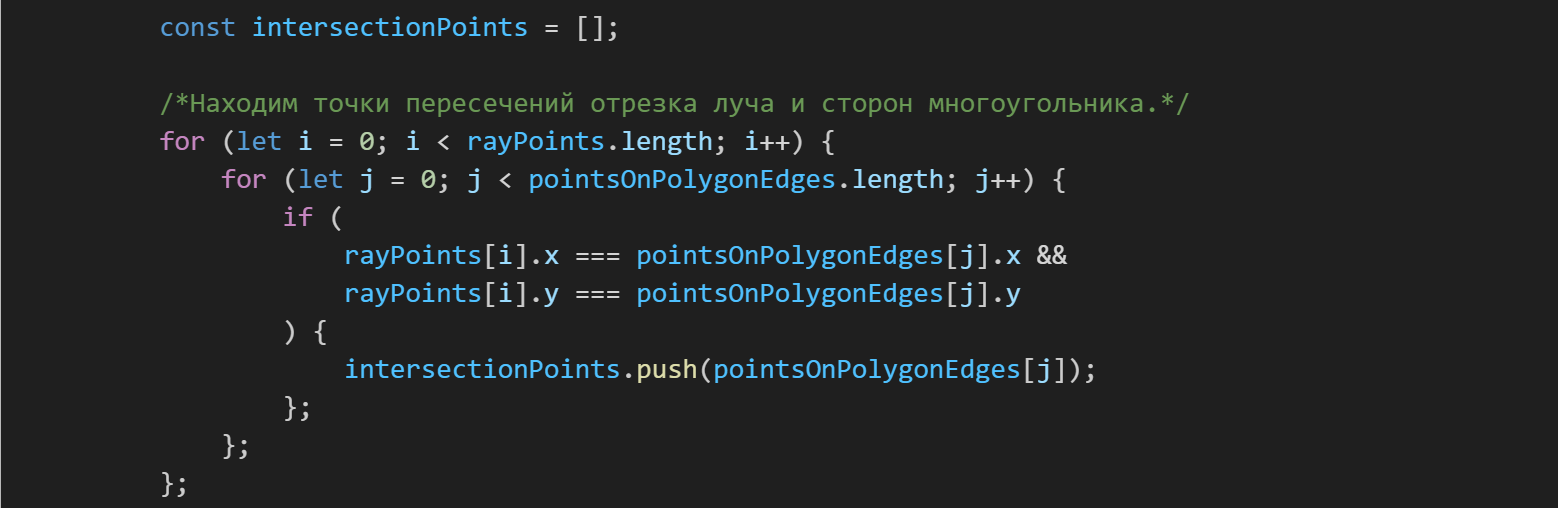


Рис. 22 - Шаг 5 функции «isPointInsidePolygon()»

6. В этом массиве удаляем «соседние» точки, чтобы избежать случаев, когда какая-то сторона многоугольника совпадает с отрезком луча (Рис. 23).

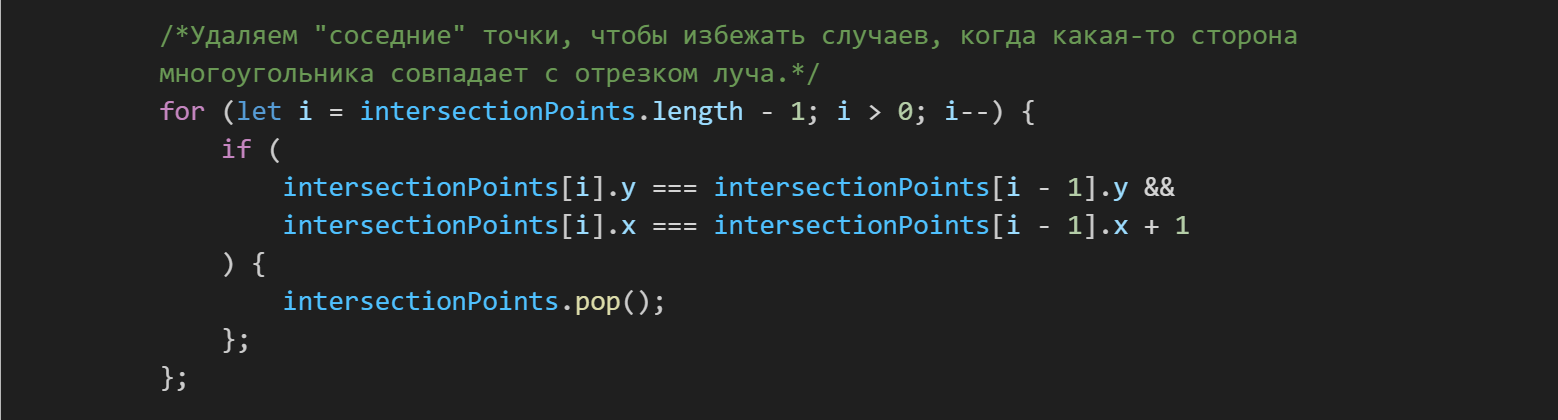


Рис. 23 - Шаг 6 функции «isPointInsidePolygon()»

7. Проверяем не получается ли так, что все точки в упомянутом массиве совпадают с вершинами многоугольника (Рис. 24).

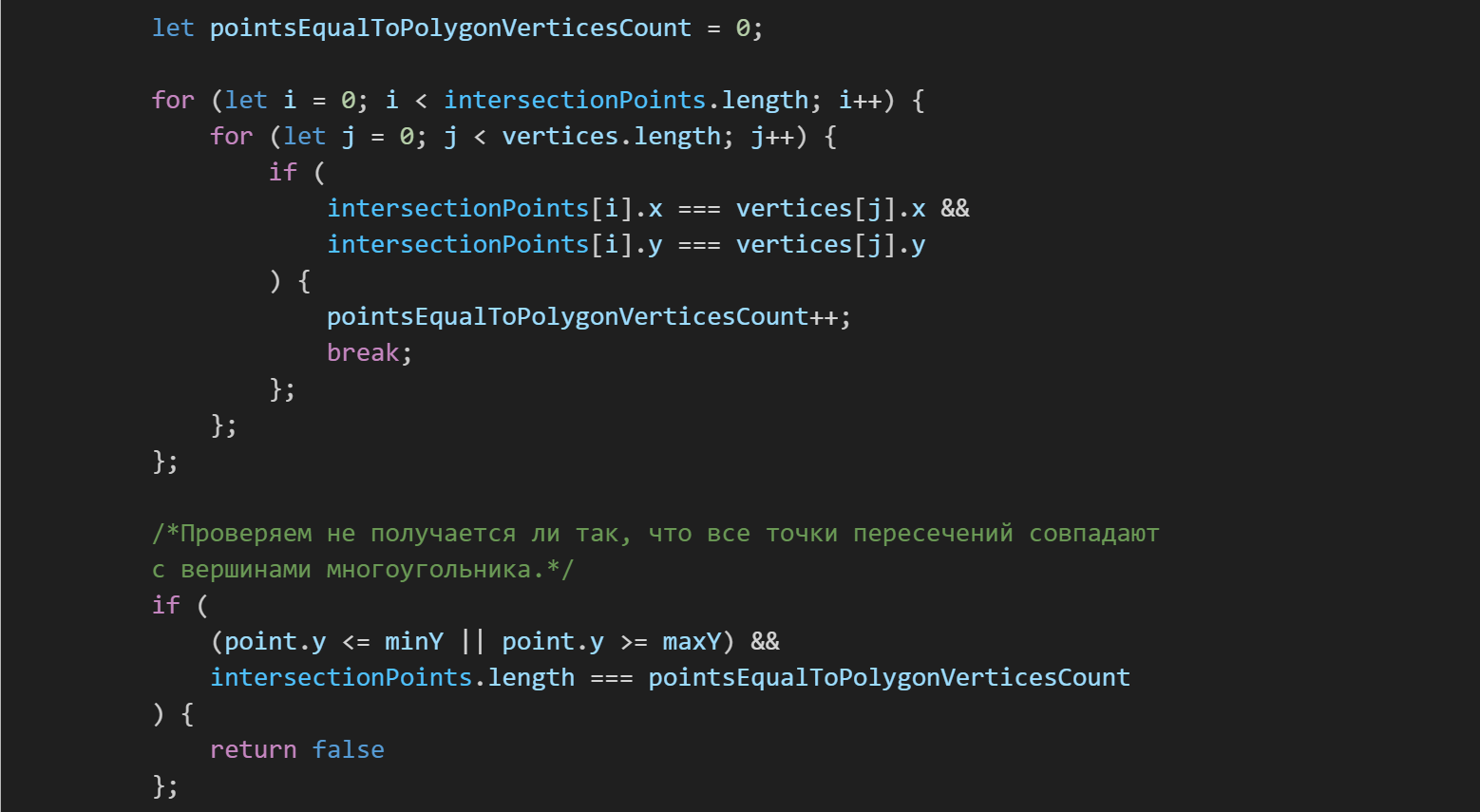


Рис. 24 - Шаг 7 функции «isPointInsidePolygon()»

8. Проверяем количество пересечений отрезка луча и сторон многоугольника на нечетность (Рис. 25).

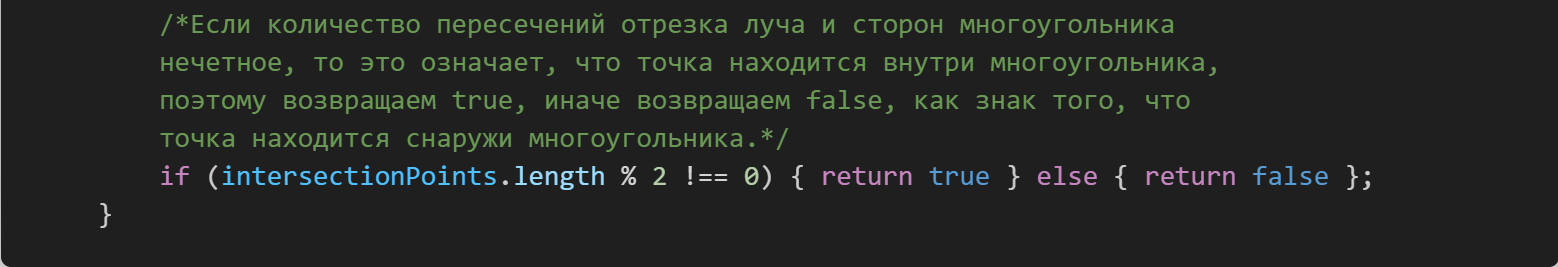


Рис. 25 - Шаг 8 функции «isPointInsidePolygon()»

2.1.3. Реализация проверки пересечения двух многоугольников

С целью упрощения реализации алгоритма проверки пересечения двух многоугольников, было создано несколько вспомогательных функций.

Для расчета векторного и скалярного произведений были созданы функции «getCrossProduct()» и «getDotProduct()» (Рис. 26, 27).

Обе функции принимает следующие параметры:

1. «v1StartPointX» - X-координата начальной точки первого вектора.

2. «v1StartPointY» - Y-координата начальной точки первого вектора.

3. «v1EndPointX» - X-координата конечной точки первого вектора.

4. «v1EndPointY» - Y-координата конечной точки первого вектора.

5. «v2StartPointX» - X-координата начальной точки второго вектора.

6. «v2StartPointY» - Y-координата начальной точки второго вектора.

7. «v2EndPointX» - X-координата конечной точки второго вектора.

8. «v2EndPointY» - Y-координата конечной точки второго вектора.

Функции «getCrossProduct()» и «getDotProduct()» возвращают величины векторного и скалярного произведений двух векторов соответственно.

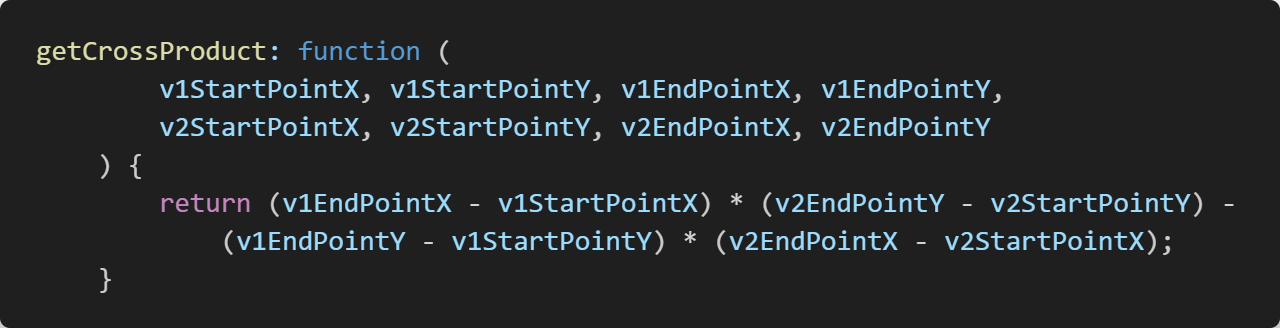


Рис. 26 - Код функции «getCrossProduct()»

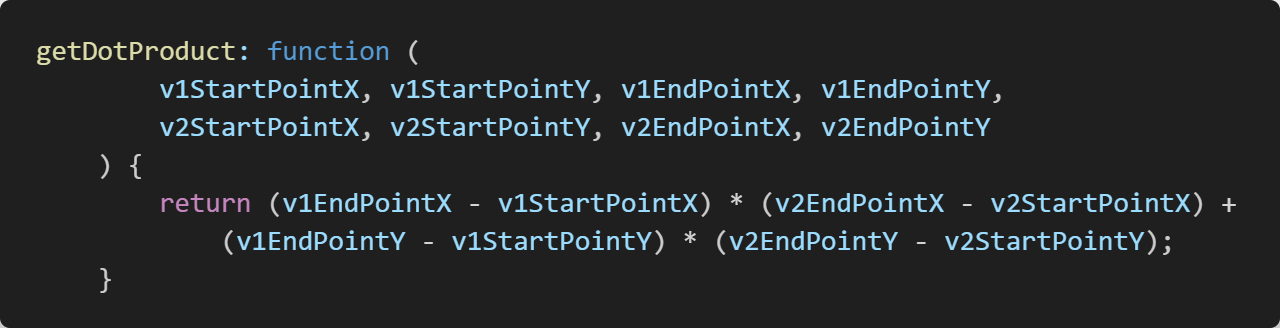


Рис. 27 - Код функции «getDotProduct()»

Для расчета квадрата длины вектора была создана функция «getSquaredVectorLength()» (Рис. 28).

Функция «getSquaredVectorLength()» принимает следующие параметры:

1. «v1StartPointX» - X-координата начальной точки вектора.

2. «v1StartPointY» - Y-координата начальной точки вектора.

3. «v1EndPointX» - X-координата конечной точки вектора.

4. «v1EndPointY» - Y-координата конечной точки вектора.

Функция «getSquaredVectorLength()» возвращает величину квадрата длины вектора.

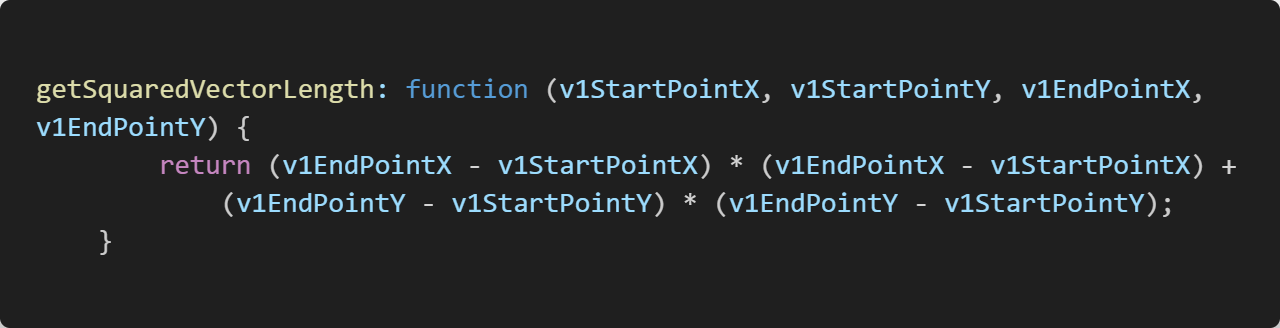


Рис. 28 - Код функции «getSquaredVectorLength()»

Для определения положения точки относительно прямой, содержащей отрезок, используя геометрический смысл векторного произведения двух векторов, была создана функция «getPointPositionRelativeToLineSegment()» (Рис. 29).

Функция «getPointPositionRelativeToLineSegment()» принимает следующие параметры:

1. «startPointX» - X-координата начальной точки вектора.

2. «startPointY» - Y-координата начальной точки вектора.

3. «endPointX» - X-координата конечной точки вектора.

4. «endPointY» - Y-координата конечной точки вектора.

5. «pointX» - X-координата точки.

6. «pointY» - Y-координата точки.

Функция «getPointPositionRelativeToLineSegment()» возвращает:

1. 0, если точка находится на прямой, содержащей отрезок.

2. 1, если точка находится «слева» от прямой, содержащей отрезок.

3. -1, если точка находится «справа» от прямой, содержащей отрезок.

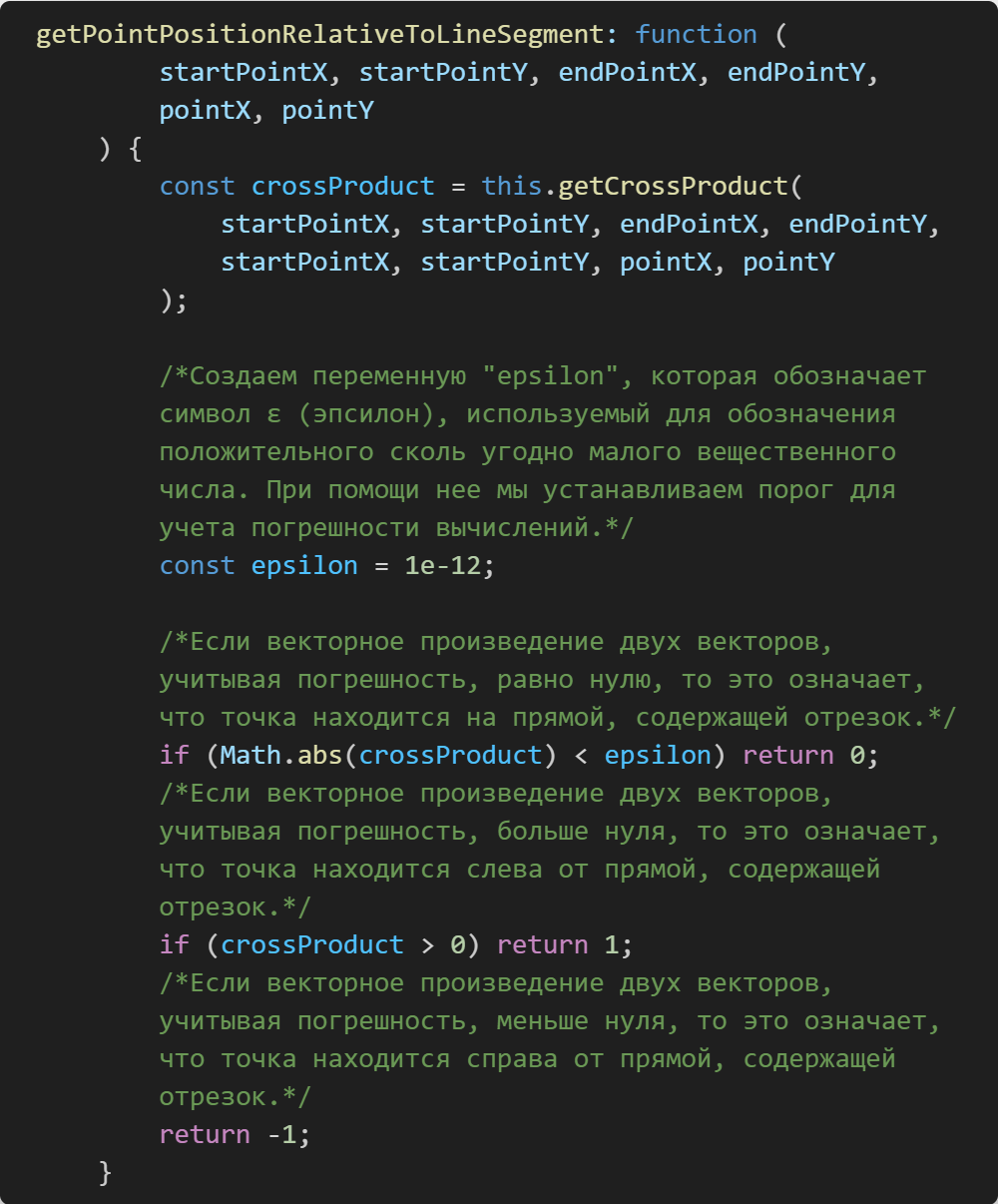


Рис. 29 - Код функции «getPointPositionRelativeToLineSegment()»

Для определения находится ли точка на отрезке была создана функция «isPointOnLineSegment()» (Рис. 30).

Функция «isPointOnLineSegment()» принимает следующие параметры:

1. «startPointX» - X-координата первой точки отрезка.

2. «startPointY» - Y-координата первой точки отрезка.

3. «endPointX» - X-координата второй точки отрезка.

4. «endPointY» - Y-координата второй точки отрезка.

5. «pointX» - X-координата точки.

6. «pointY» - Y-координата точки.

Функция «isPointOnLineSegment()» возвращает true, как знак того, что точка находится на отрезке, иначе false.

Логика работы функции «isPointOnLineSegment()» следующая:

1. Проверяем не лежит ли точка за пределами прямой, содержащей отрезок.

2. Рассчитываем скалярное произведение двух векторов, где первый вектор строится на основе точек отрезка, а второй вектор строится из начальной точки первого вектора в проверяемую точку.

3. Проверяем не лежит ли точка до первой точки отрезка.

4. Проверяем не лежит ли точка после второй точки отрезка.

5. Если все проверки пройдены, то это означает, что точка лежит на отрезке.



Рис. 30 - Код функции «isPointOnLineSegment()»

Для определения пересекаются ли два отрезка была создана функция «doTwoLineSegmentsIntersect()».

Функция «doTwoLineSegmentsIntersect()» принимает следующие параметры:

1. «point01» - объект, содержащий координаты первой конечной точки первого отрезка.

2. «point02» - объект, содержащий координаты второй конечной точки первого отрезка.

3. «point03» - объект, содержащий координаты первой конечной точки второго отрезка.

4. «point04» - объект, содержащий координаты второй конечной точки второго отрезка.

Функция «doTwoLineSegmentsIntersect()» возвращает true, как знак того, что два отрезка пересекаются, иначе false.

Логика работы функции «doTwoLineSegmentsIntersect()» следующая:

1. Определяем положение конечных точек отрезков относительно прямых, содержащих противоположные отрезки (Рис. 31).

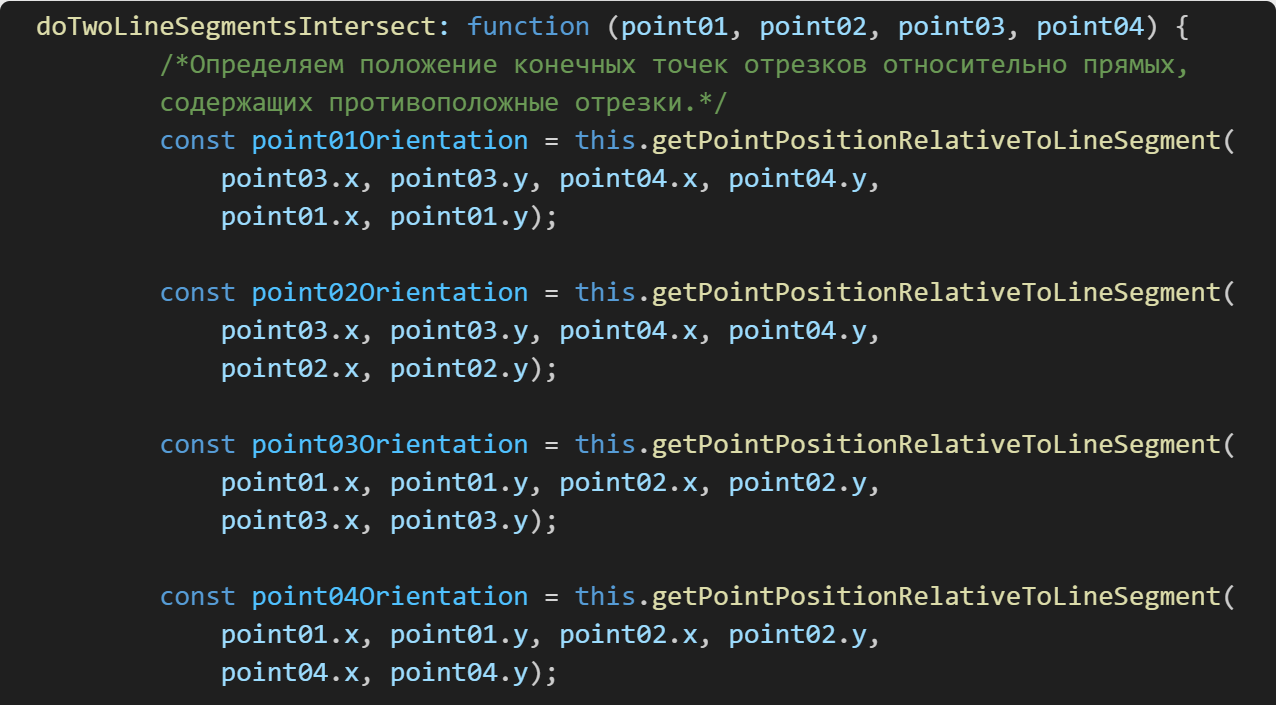


Рис. 31 - Шаг 1 функции «doTwoLineSegmentsIntersect()»

2. Проверяем общий случай пересечения двух отрезков (Рис. 32).

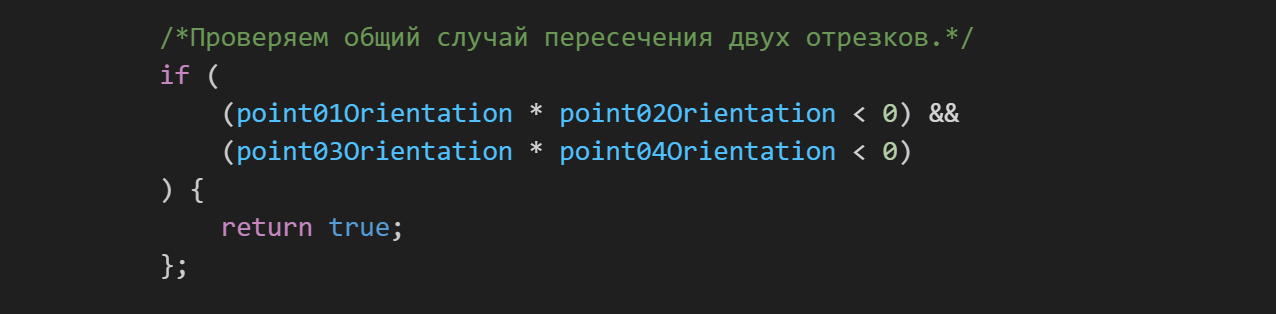


Рис. 32 - Шаг 2 функции «doTwoLineSegmentsIntersect()»

3. Проверяем частные случаи пересечения двух отрезков (Рис. 33).

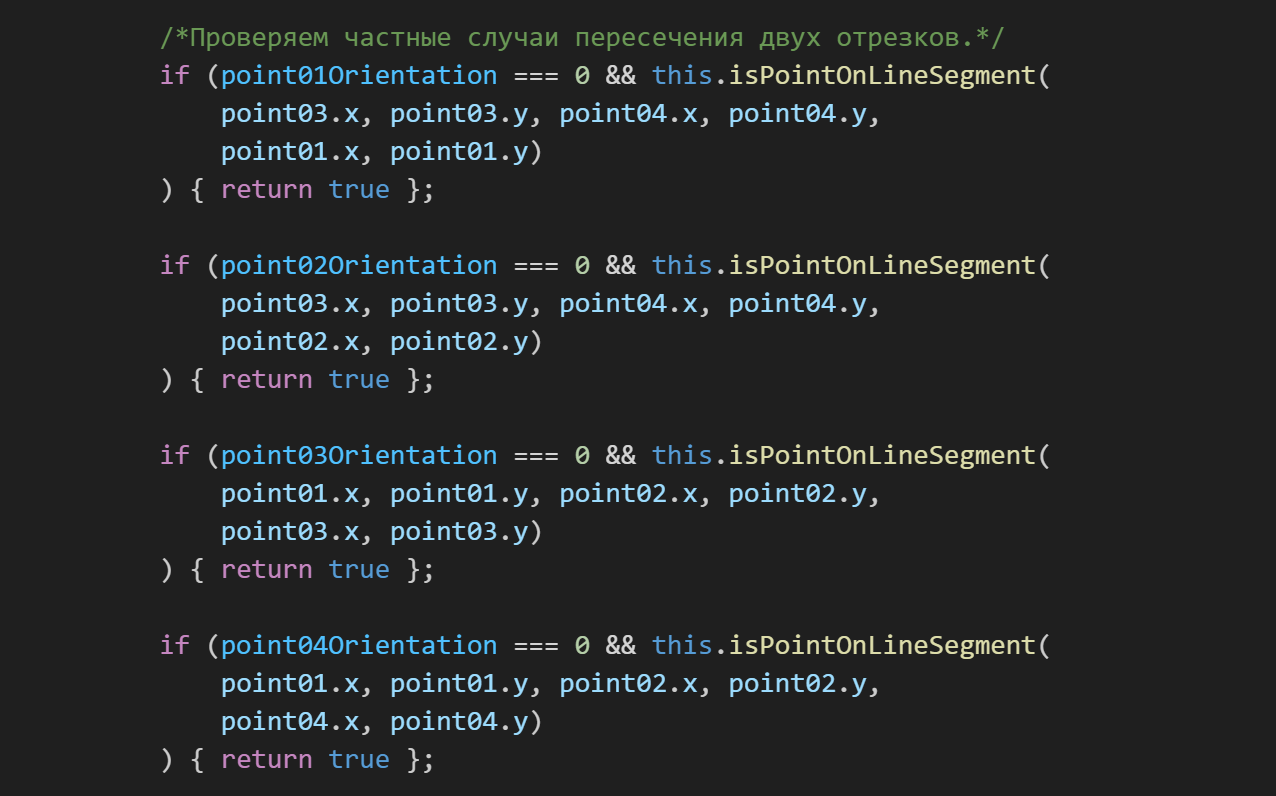


Рис. 33 - Шаг 3 функции «doTwoLineSegmentsIntersect()»

4. Если ни одна проверка не прошла, то это означает, что отрезки не пересекаются (Рис. 34).

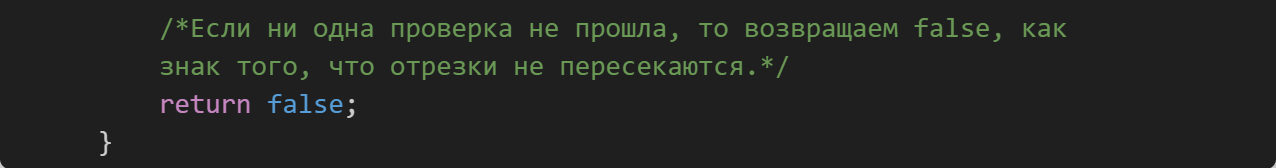


Рис. 34 - Шаг 4 функции «doTwoLineSegmentsIntersect()»

Для итоговой реализации алгоритма проверки пересечения была создана функция «doTwoPolygonsIntersect()».

Функция «doTwoPolygonsIntersect()» принимает следующие параметры:

1. «vertices01» - объект, содержащий координаты вершин первого многоугольника.

2. «vertices02» - объект, содержащий координаты вершин второго многоугольника.

Функция «doTwoPolygonsIntersect()» возвращает true, как знак того, что многоугольники пересекаются, иначе false.

Логика работы функции «doTwoPolygonsIntersect()» следующая:

1. Перебираем вершины многоугольников и ищем их совпадения (Рис. 35).



Рис. 35 - Шаг 1 функции «doTwoPolygonsIntersect()»

2. Перебираем пары соседних вершин каждого многоугольника, чтобы на их основе проверить не пересекаются ли какие-то стороны этих многоугольников (Рис. 36).

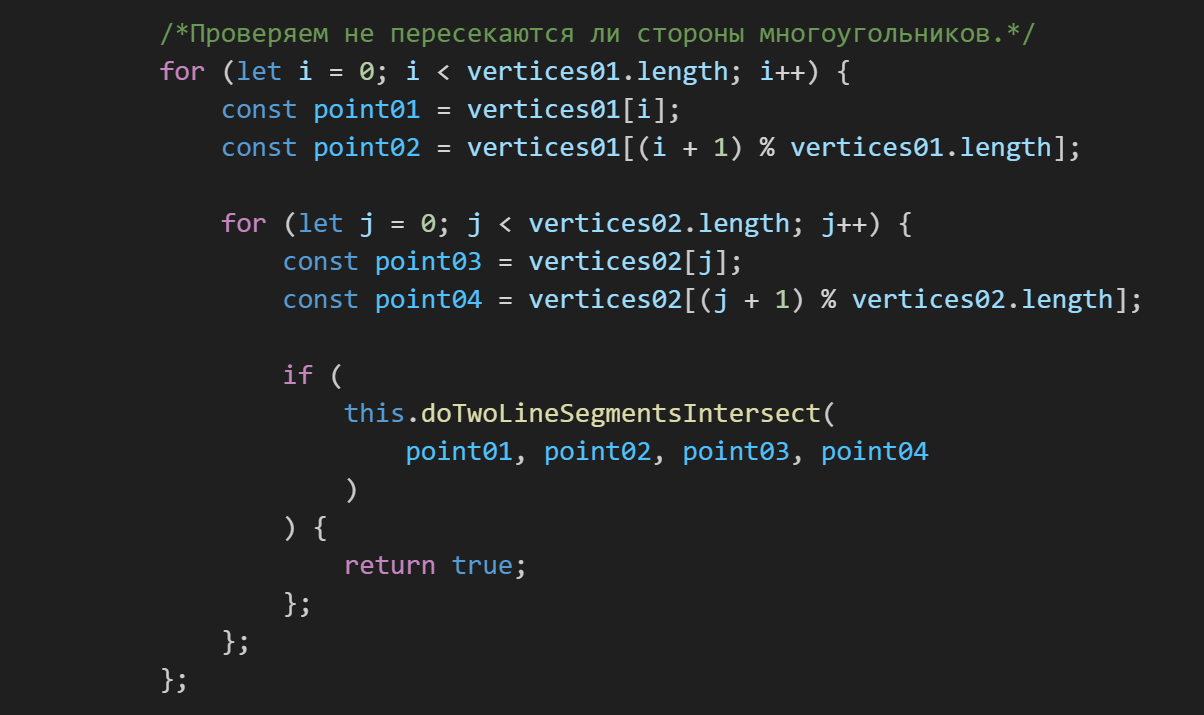


Рис. 36 - Шаг 2 функции «doTwoPolygonsIntersect()»

3. Если стороны не пересекаются, то проверяем не находится ли один многоугольник полностью внутри другого (Рис. 37).

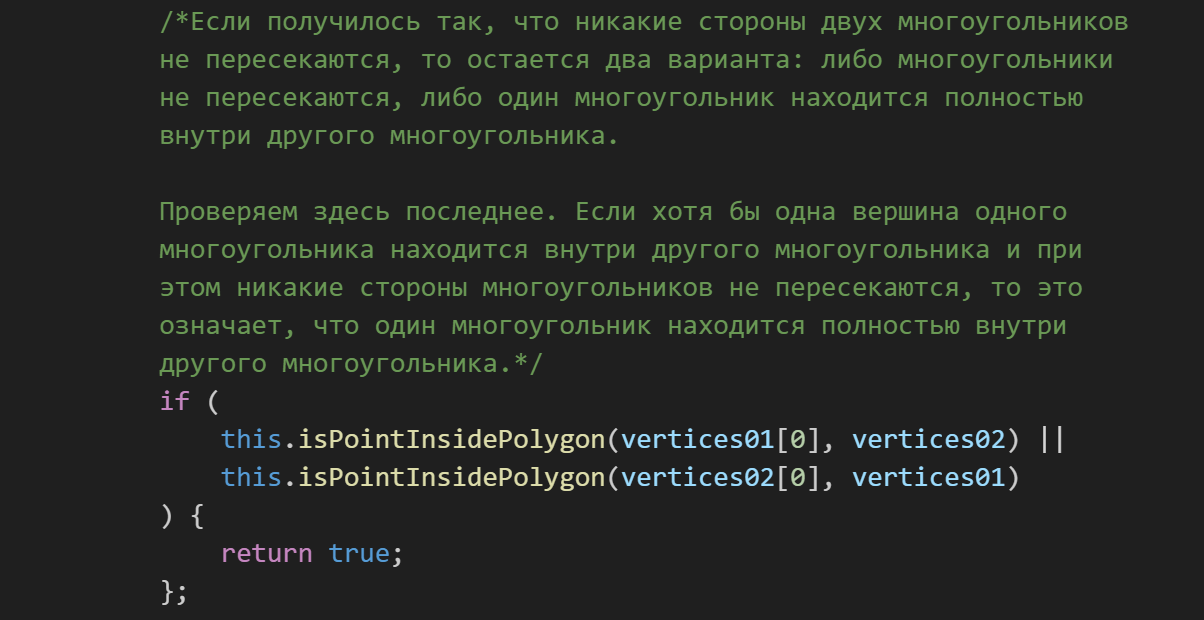


Рис. 37 - Шаг 3 функции «doTwoPolygonsIntersect()»

4. Если ни одна проверка не прошла, то это означает, что многоугольники не пересекаются (Рис. 38).

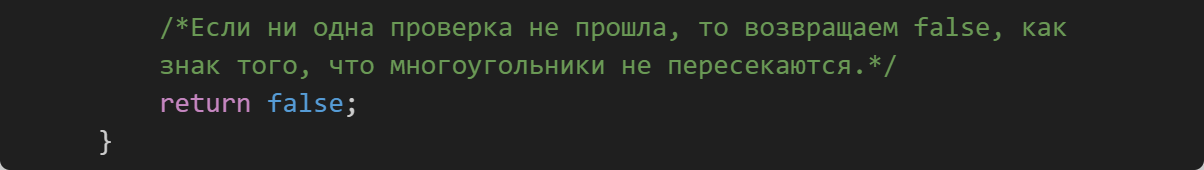


Рис. 38 - Шаг 4 функции «doTwoPolygonsIntersect()»

2.2. Реализация алгоритмов корректировки диагонального движения в двумерных видеоиграх на языке JavaScript

В этом параграфе будет рассмотрена реализация на языке JavaScript решения проблемы, когда скорость объекта при диагональном движении больше, чем она должна быть.

2.2.1. Реализация нормализации вектора скорости

Для корректировки скорости диагонального движения объектов при помощи нормализации вектора скорости такого движения была создана функция «correctDiagonalMovementSpeed()» (Рис. 39).

Функция «correctDiagonalMovementSpeed()» принимает следующие параметры:

1. «currentSpeedX» - текущая скорость передвижения объекта по оси X.

2. «currentSpeedY» - текущая скорость передвижения объекта по оси Y.

Функция «correctDiagonalMovementSpeed()» возвращает объект, содержащий скорости по осям X и Y для корректной скорости объекта при диагональном движении.

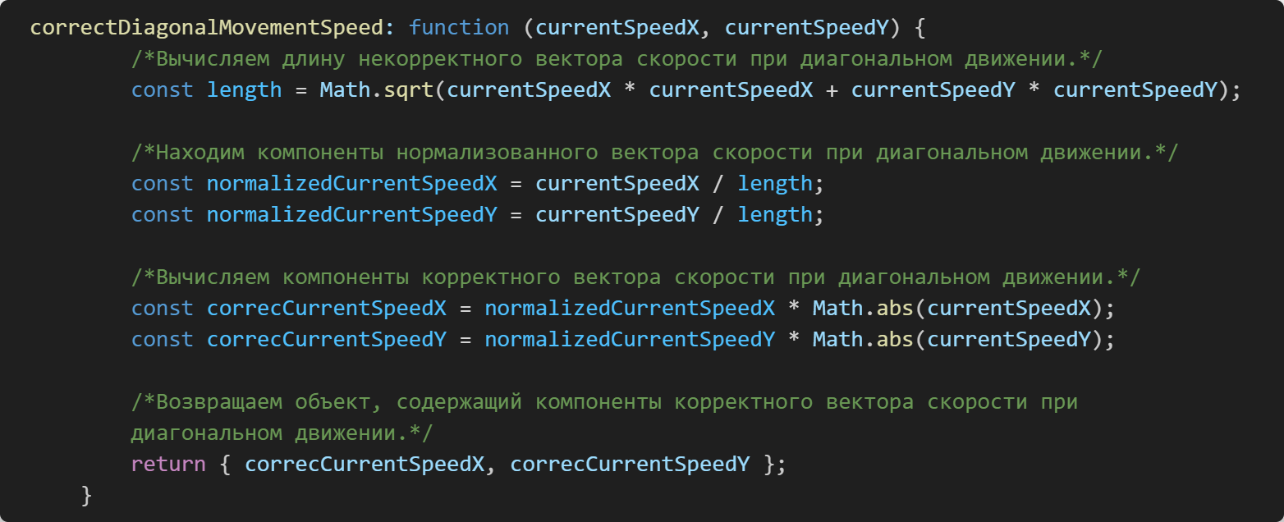


Рис. 39 - Код функции «correctDiagonalMovementSpeed()»

2.3. Реализация вспомогательных алгоритмов на языке JavaScript

Этот параграф будет посвящен разбору алгоритма, который позволяет генерировать вершины многоугольника. Этот алгоритм был специально разработан под задачи данного проекта.

2.3.1. Реализация генерации координат вершин многоугольника

Для того, чтобы можно было создавать многоугольники в видеоигре, был создана функция «preparePolygonIntVerticesData()», которая рассчитывает координаты вершин многоугольника в рамках указанной области.

Функция «preparePolygonIntVerticesData()» принимает следующие параметры:

1. «numberOfVertices» - количество вершин многоугольника.

2. «x1» - минимальная X-координата области для генерации вершин.

3. «x2» - максимальная X-координата области для генерации вершин.

4. «y1» - минимальная Y-координата области для генерации вершин.

5. «y2» - максимальная Y-координата области для генерации вершин.

6. «clockwiseStepX» - максимальный сдвиг по оси X вершин друг от друга.

7. «clockwiseStepY» - максимальный сдвиг по оси Y вершин друг от друга.

8. «cellWidth» - ширина одной клетки в сетке холста.

9. «cellHeight» - высота одной клетки в сетке холста.

10. «gridAligned» - булев параметр, указывающий нужно ли выравнивать координаты вершин по сетке.

Функция «preparePolygonIntVerticesData()» возвращает массив объектов, содержащих рассчитанные координаты вершин многоугольника.

Логика работы функции «preparePolygonIntVerticesData()» следующая:

1. Определяем сколько примерно вершин должно быть в каждой четверти области для генерации вершин (Рис. 40).

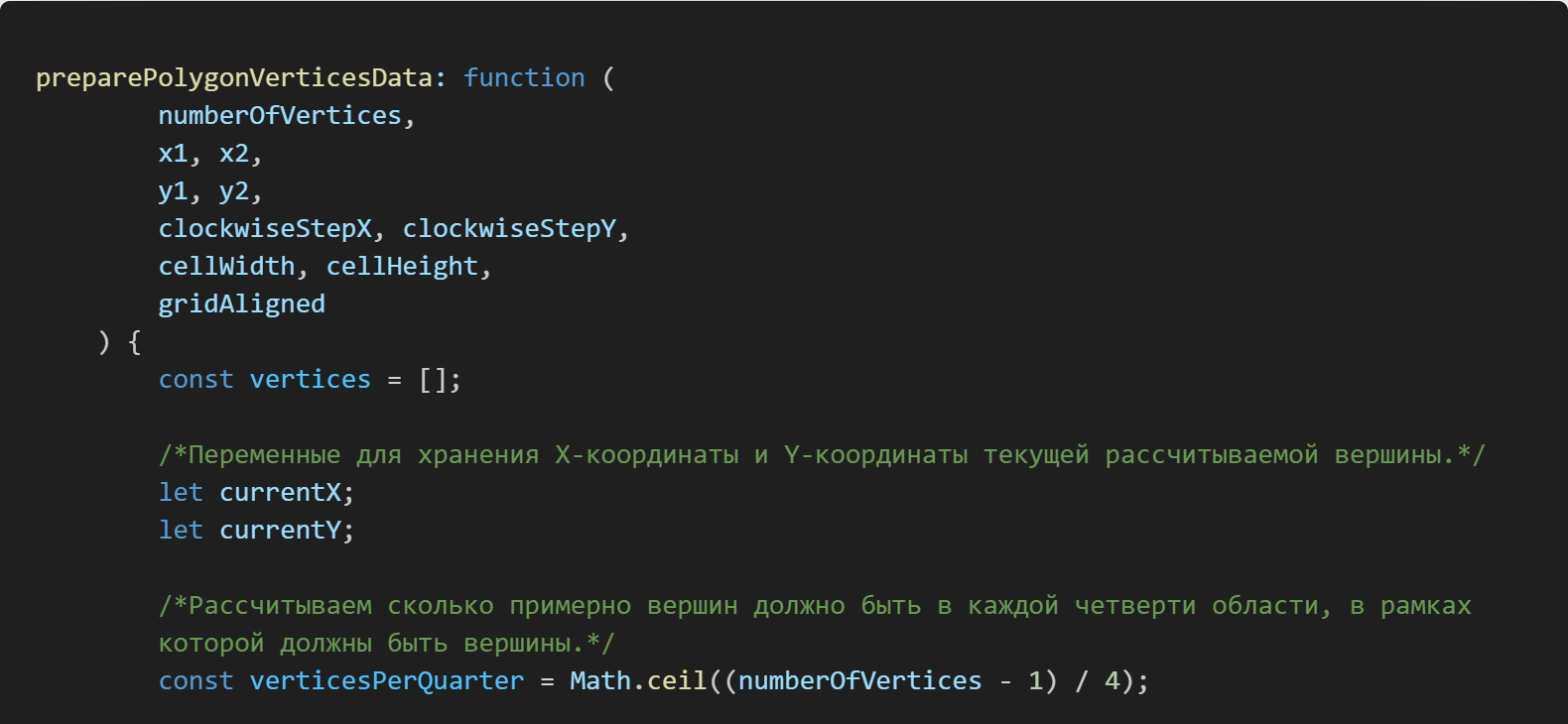


Рис. 40 - Шаг 1 функции «preparePolygonIntVerticesData()»

2. Рассчитываем первую координату вершины (Рис. 41).

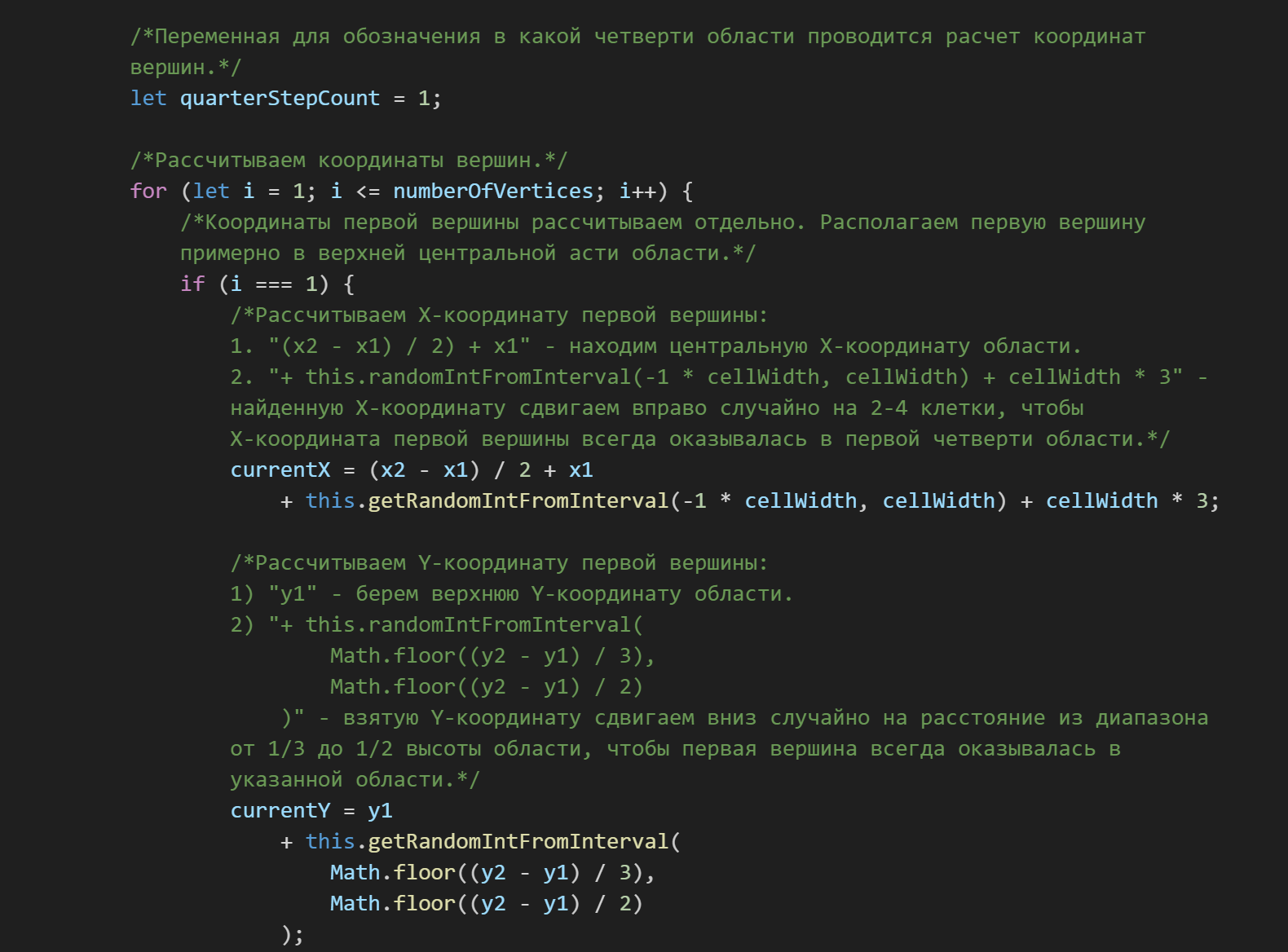


Рис. 41 - Шаг 2 функции «preparePolygonIntVerticesData()»

3. Определяем в какой четверти производим расчет вершин (Рис. 42):

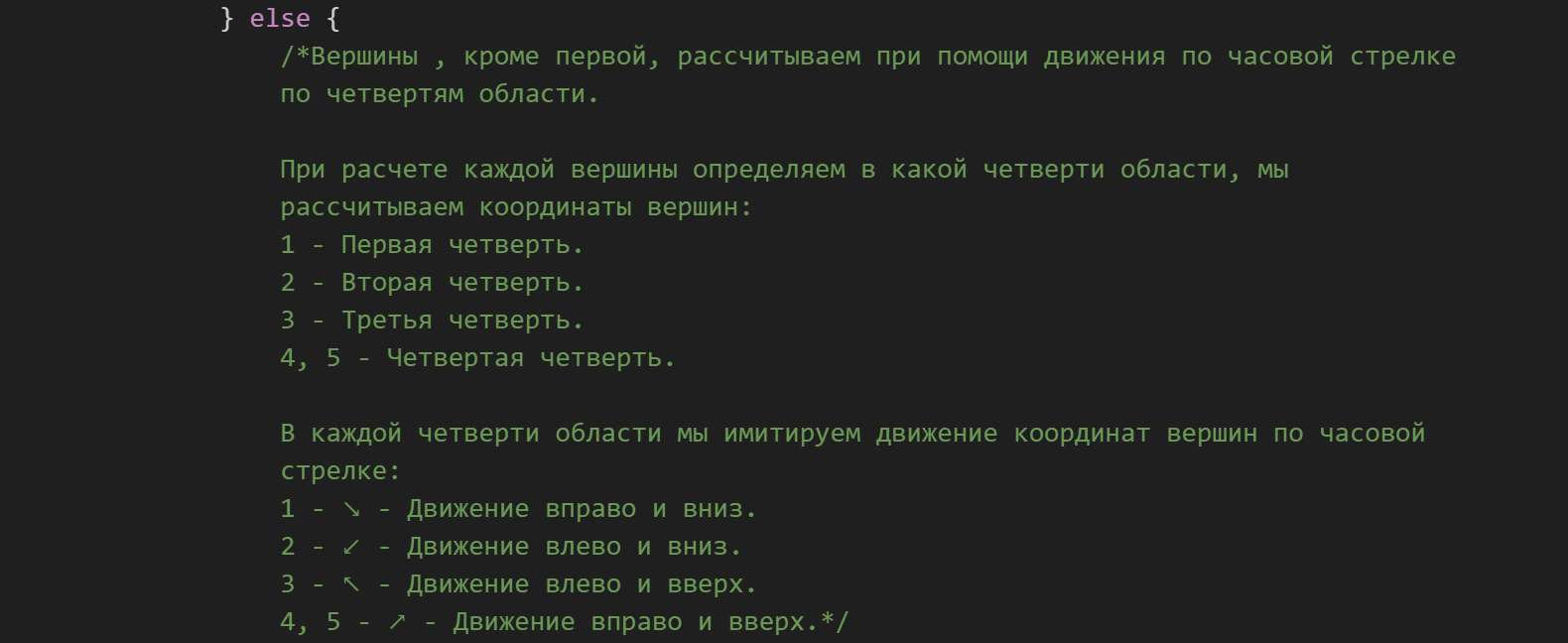


Рис. 42 - Шаг 3 функции «preparePolygonIntVerticesData()»

1) Если в первой, то рассчитываем координаты остальных вершин в первой четверти области для генерации вершин (Рис. 43).

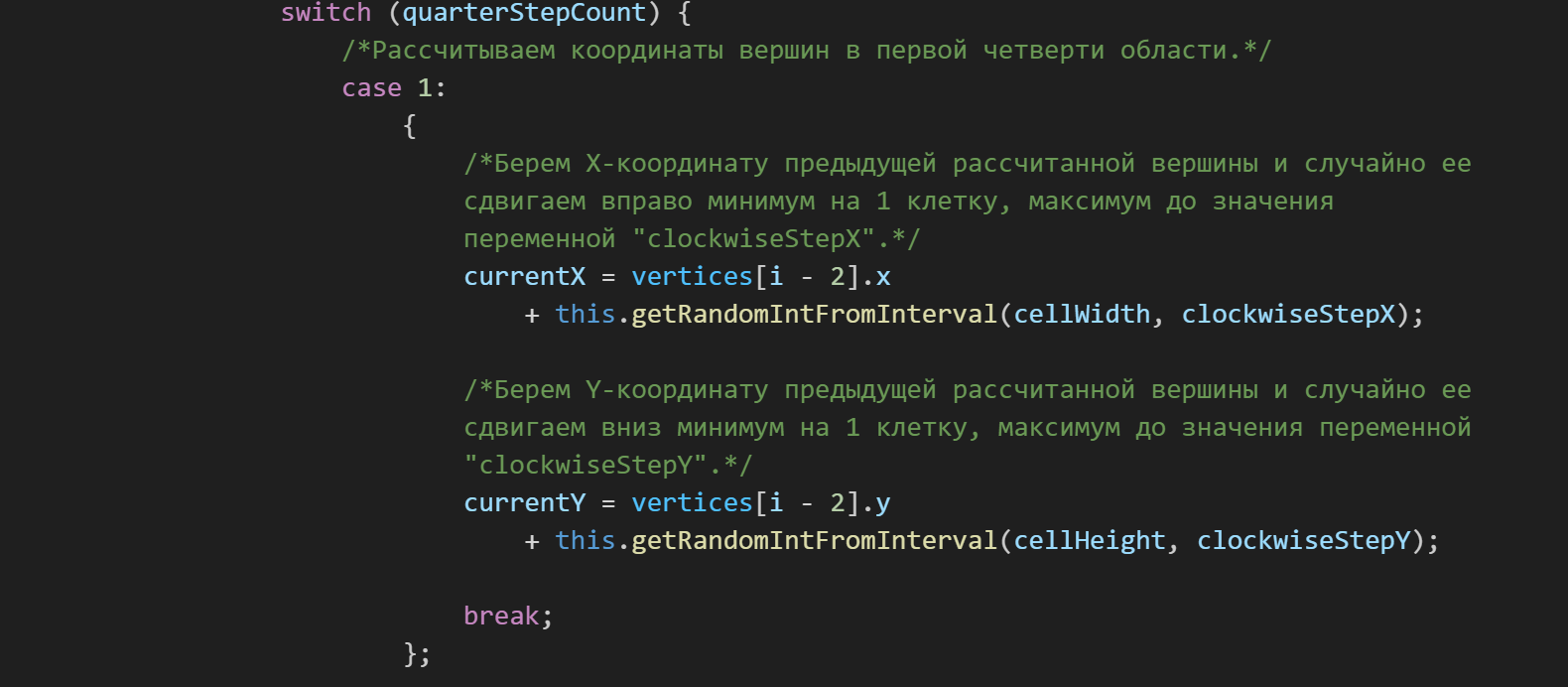


Рис. 43 - Шаг 3.1 функции «preparePolygonIntVerticesData()»

2) Если во второй, то рассчитываем координаты вершин во второй четверти области для генерации вершин (Рис. 44).

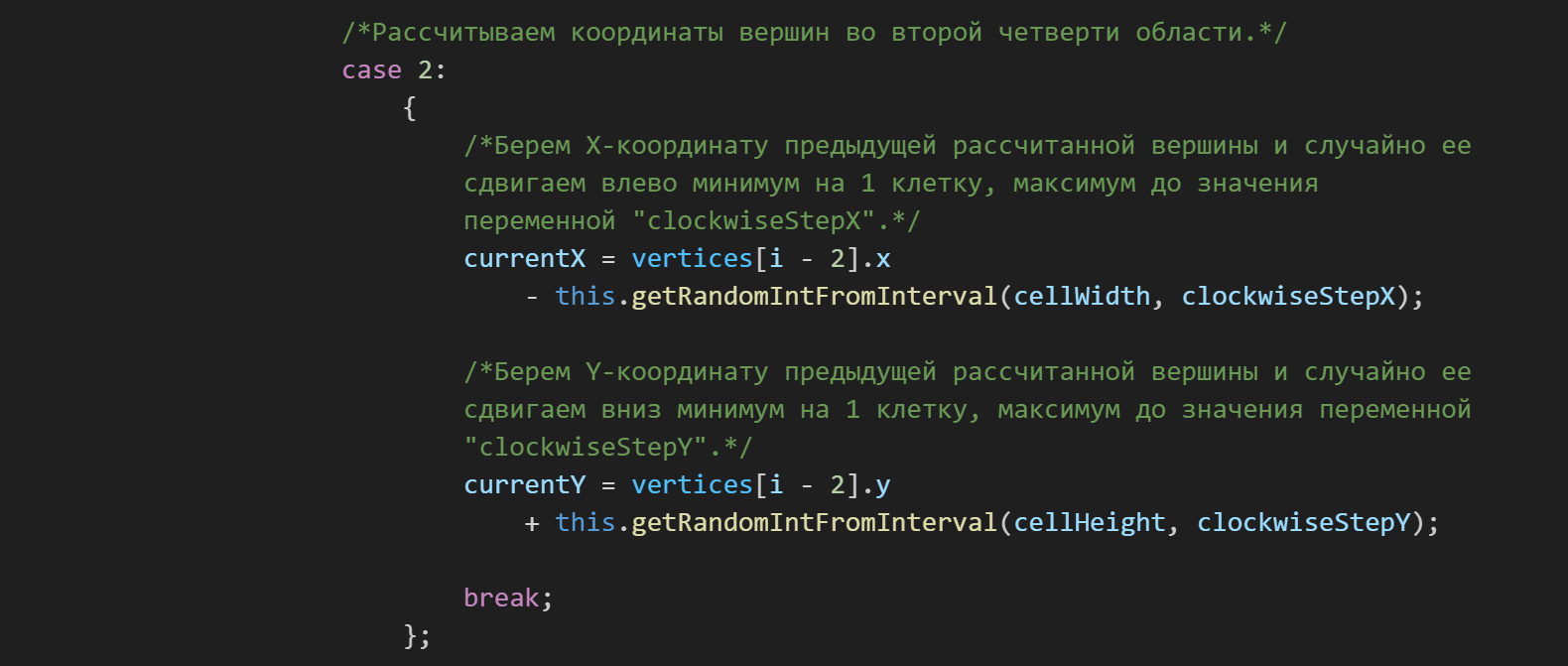


Рис. 44 - Шаг 3.2 функции «preparePolygonIntVerticesData()»

3) Если в третьей, то рассчитываем координаты вершин в третьей четверти области для генерации вершин (Рис. 45).

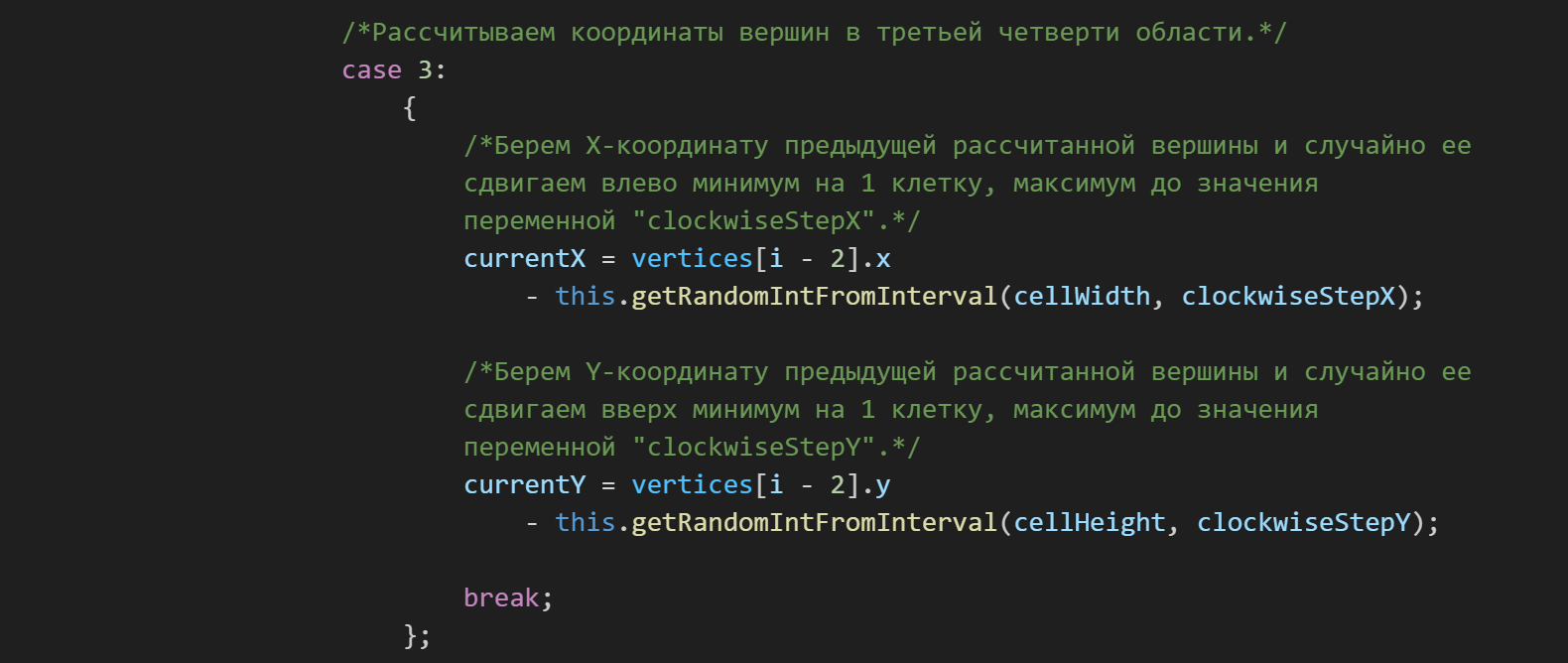


Рис. 45 - Шаг 3.3 функции «preparePolygonIntVerticesData()»

4) Если в четвертой, то рассчитываем координаты вершин в четвертой четверти области для генерации вершин (Рис. 46).

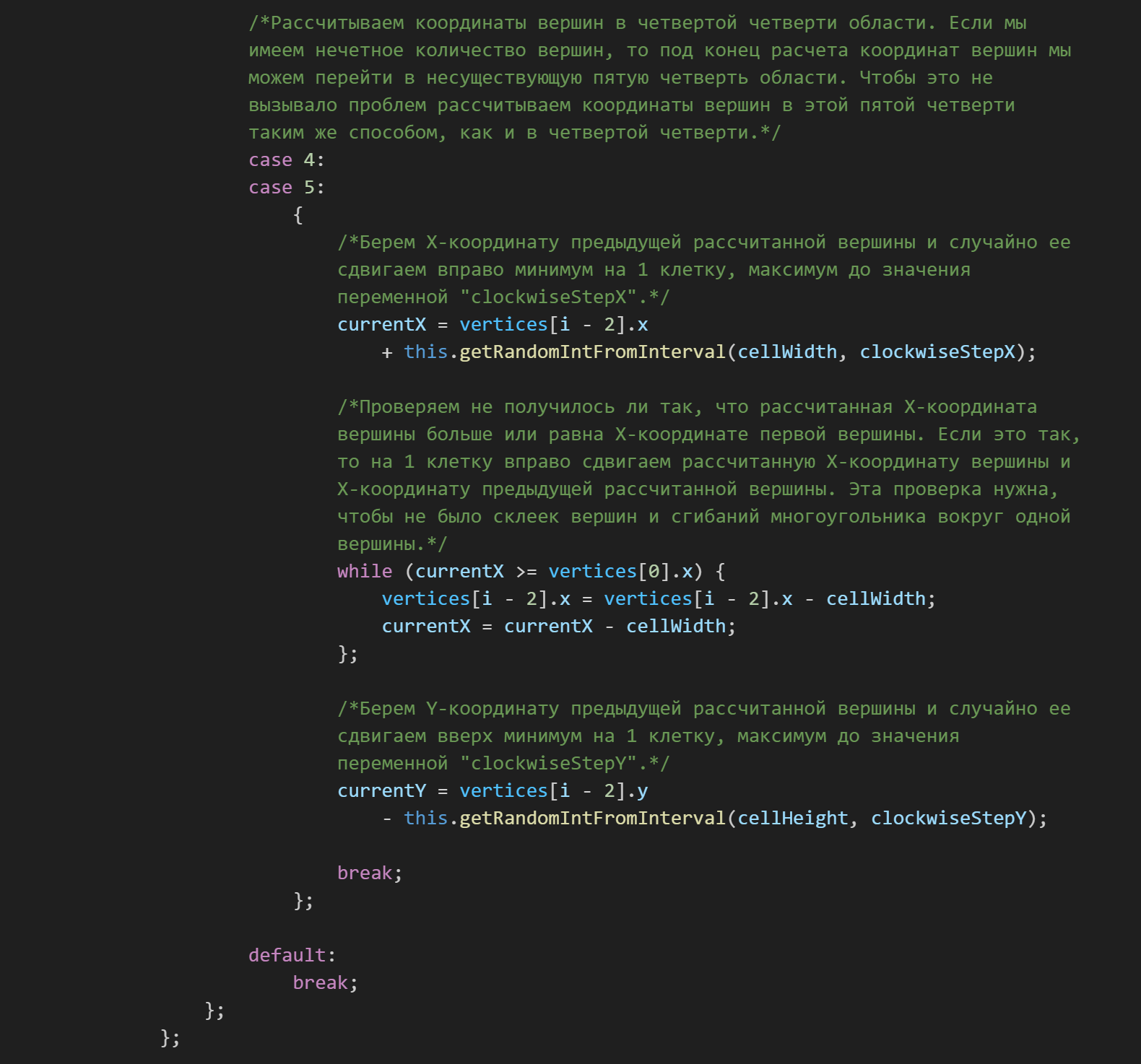


Рис. 46 - Шаг 3.4 функции «preparePolygonIntVerticesData()»

4. Выравниваем рассчитанные координаты вершины по сетке, если необходимо (Рис. 47).

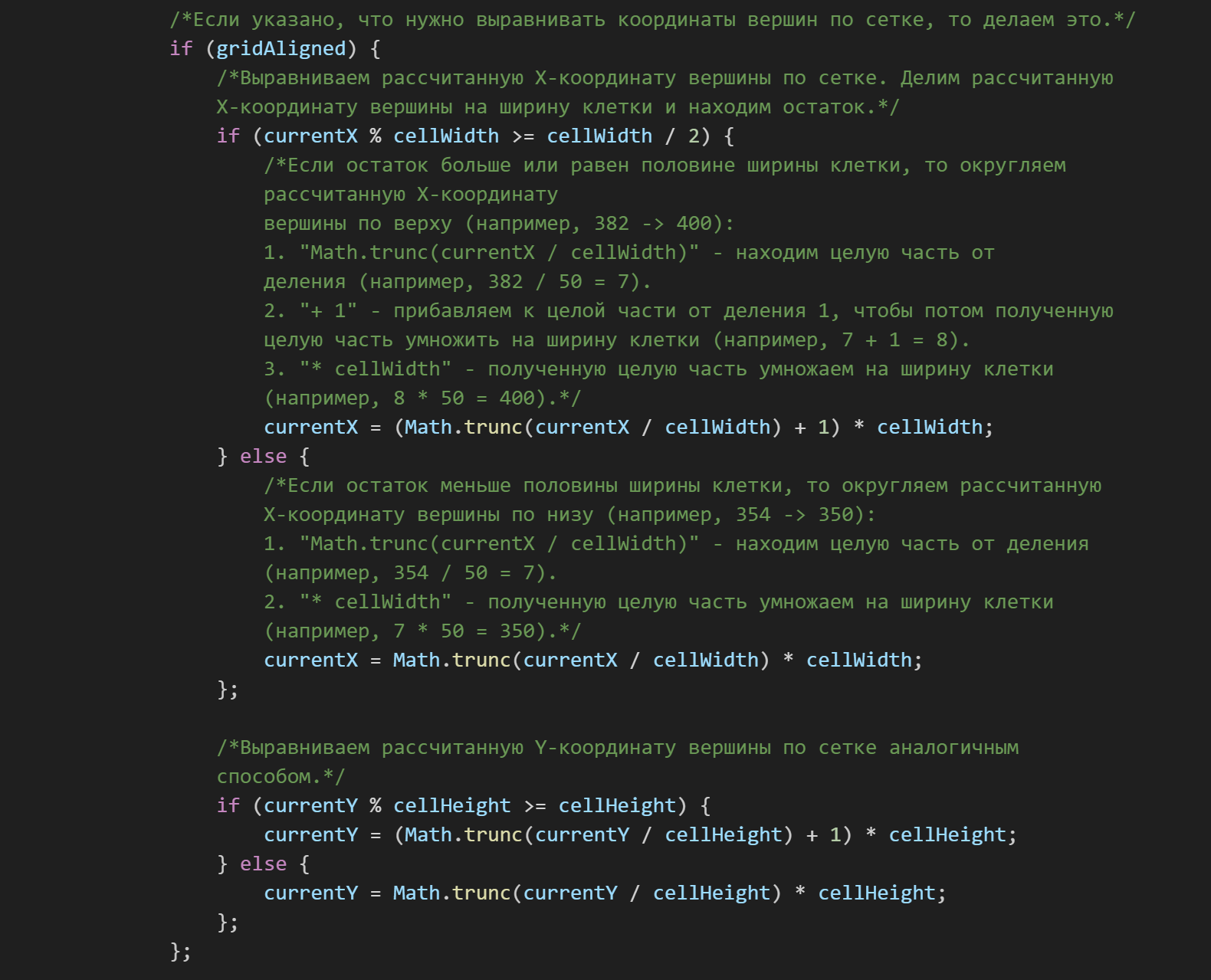


Рис. 47 - Шаг 4 функции «preparePolygonIntVerticesData()»

5. Корректируем рассчитанные координаты вершины, выходящие за пределы области для генерации вершин (Рис. 48).

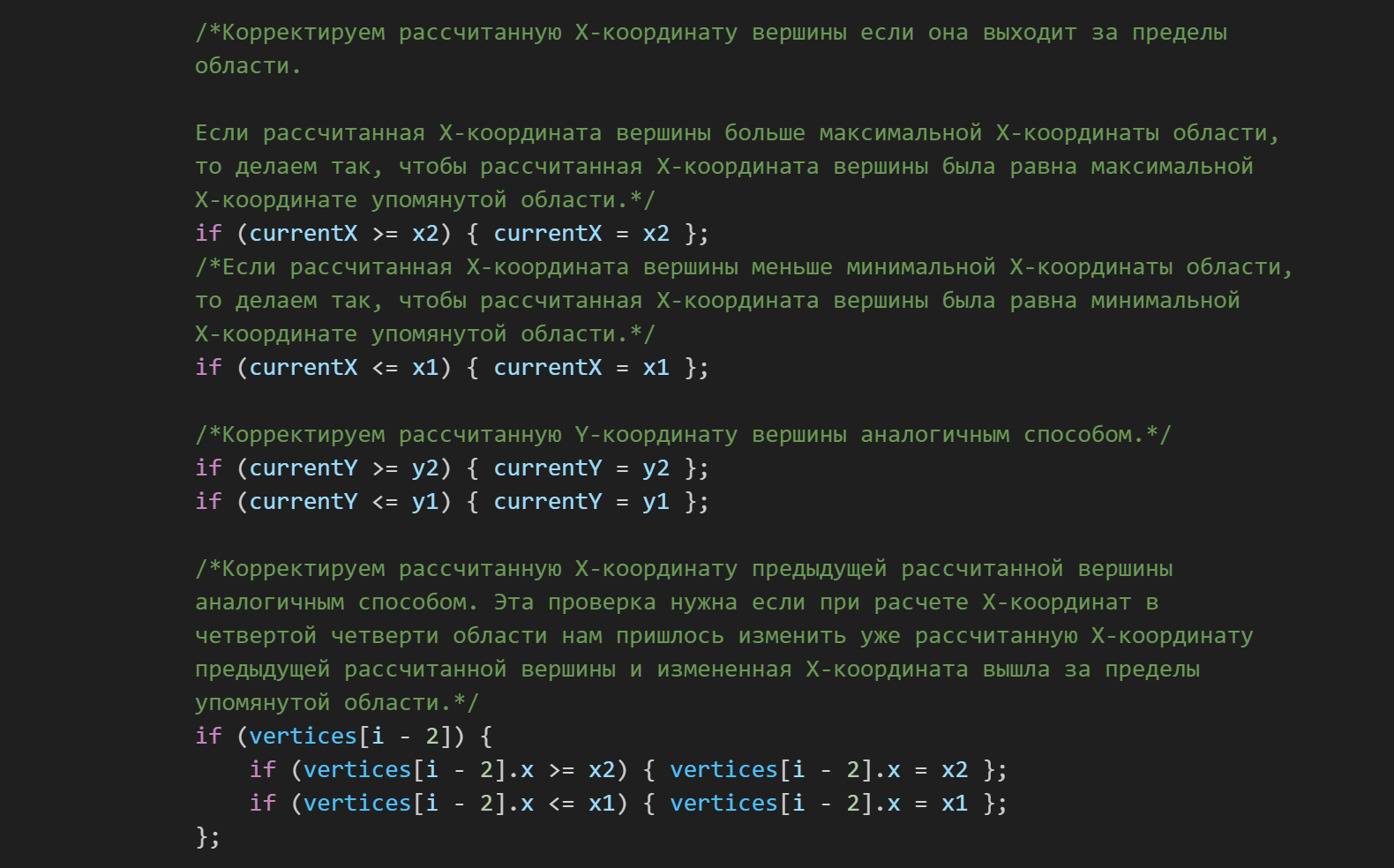


Рис. 48 - Шаг 5 функции «preparePolygonIntVerticesData()»

6. Рассчитав и откорректировав координаты вершины, добавляем ее в массив (Рис. 49).

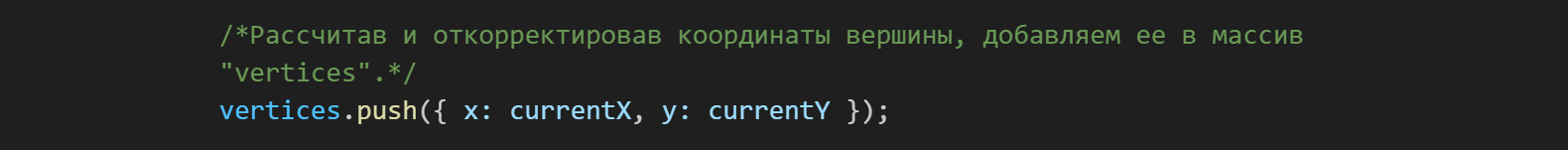


Рис. 49 - Шаг 6 функции «preparePolygonIntVerticesData()»

7. Проверяем не нужно ли для расчета координат следующей вершины перейти в следующую четверть области для генерации вершин и возвращаемся к шагу 3 (Рис. 50).

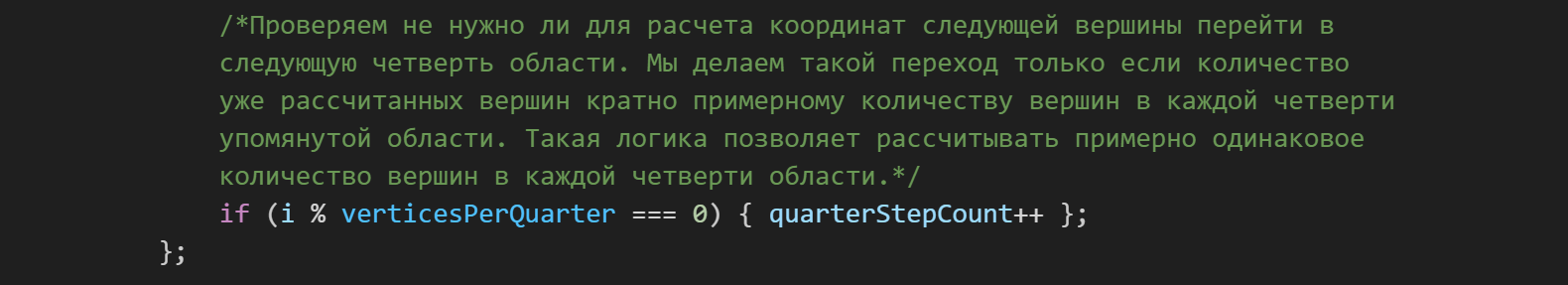


Рис. 50 - Шаг 7 функции «preparePolygonIntVerticesData()»

8. Рассчитав все вершины, возвращаем массив объектов, содержащих рассчитанные координаты вершин (Рис. 51).

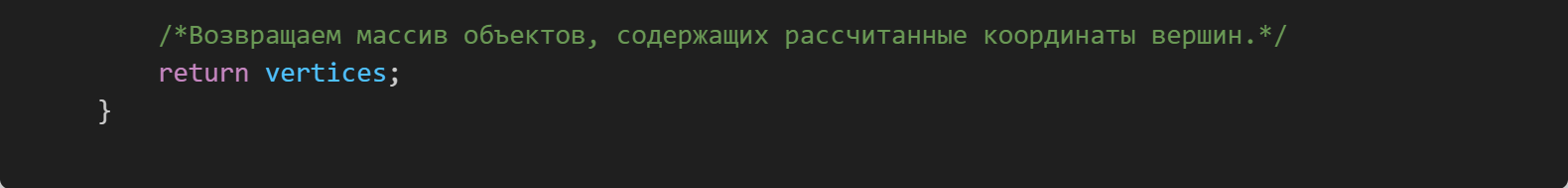


Рис. 51 - Шаг 7 функции «preparePolygonIntVerticesData()»

2.4. Рекомендации для начинающих разработчиков двумерных видеоигр на языке JavaScript

В заключительном параграфе практической части проекта будут предложены рекомендации для начинающих разработчиков как по обработке коллизий и движения объектов, так и по разработке двумерных видеоигр в целом.

2.4.1. Рекомендации по обработке коллизий и движения объектов

На основе опыта, полученного при работе над данным проектом, предлагаются следующие рекомендации касательно обработки коллизий объектов в двумерной видеоигре:

1. Заранее продумывайте какие геометрические формы будут принимать объекты в видеоигре. Даже если используются готовые изображения, то для обработки коллизий игровые объекты должны иметь хитбокс, то есть некую упрощенную геометрическую форму, например, точка, прямоугольник или многоугольник.

2. Заранее определяйте какие типы коллизий будут в видеоигре в зависимости от форм объектов, например, точка и прямоугольник, точка и многоугольник, или два многоугольника. Это поможет понять какие математические алгоритмы понадобятся для обработки коллизий.

3. При реализации алгоритмов обработки коллизий не забывайте про уникальные частные случаи, которые не покрываются общей логикой алгоритмов.

4. Поскольку алгоритмы, обрабатывающие коллизии, иногда требуют достаточно больших вычислений (например, линейная интерполяция), то в целях улучшения производительности стоит делать следующее:

1) Добавляйте условия, уточняющие, когда именно стоит запускать обработку коллизии. Например, не нужно каждый кадр проверять задевает ли пуля (точка) камень (многоугольник), если пуля не находится, в рамках «ограничивающей коробки» этого камня.

2) Более простые типы коллизий обрабатывайте более простыми алгоритмами. Например, хоть алгоритм проверки нахождения точки внутри многоугольника и покрывает случай, когда точка находится внутри прямоугольника, но в такой ситуации лучше используйте отдельный алгоритм для проверки нахождения точки внутри прямоугольника, так как он проще и требует гораздо меньше вычислительных ресурсов.

Касательно обработки движения объектов в двумерной видеоигре предлагаются следующие рекомендации:

1. Заранее продумывайте какие объекты могут двигаться, а также отдельные характеристики их движения, например, замедление.

2. Для корректировки скорости диагонального движения объектов используйте нормализацию вектора скорости.

3. Поскольку движение объектов в видеоигре осуществляется не как в реальном мире, а путем «телепортации» объектов из одной точки координат в другую, то отдельно обрабатывайте случаи, когда у объектов «слишком большая» скорость, чтобы они не проходили сквозь другие объекты.

2.4.2. Общие рекомендации по разработке

На основе опыта работы над данным проектом предлагаются следующие практические рекомендации по разработке двумерных видеоигр на языке JavaScript:

1. Перед тем как реализовывать какой-либо алгоритм, исследуйте его теоретическую часть, чтобы не потратить время на изобретение алгоритма, который уже существует.

2. Для каждого алгоритма создавайте отдельную функцию, чтобы всегда иметь возможность переиспользовать этот алгоритм в любом месте приложения без прибегания к дублированию кода.

3. Реализованные алгоритмы сохраняйте в отдельном файлах или объектах, чтобы можно было их переиспользовать в других проектах.

4. Заранее продумывайте архитектуру и файловую структуру видеоигры. Под каждую игровую сущность (например, враг, пуля или камень) или под каждую составную часть видеоигры (например, игровой цикл, управление или звук) следует выделять отдельный класс или объект. Выделенные классы или объекты следует располагать по отдельным файлам и папкам. Понятность общей структуры приложения будет упрощать работу над кодовой базой видеоигры.

5. Поскольку язык JavaScript не очень хорошо работает с числами, у которых много знаков, особенно после запятой, то для работы с такими числами используйте следующее:

1) Округление.

2) Специальные типы данных для больших чисел, например, BigInt для работы с большими целыми числами, или Float32Array для хранения чисел с большим количеством знаков после запятой.

3) Сторонние библиотеки, предоставляющие функционал для корректной работы с такими числами.

6. Для генерации игрового цикла используйте функцию «requestAnimationFrame()», а не функции «setTimeout()» или «setInterval()». Последние плохо влияют на производительность приложения и имеют проблемы при слишком частом вызове. Также функции «setTimeout()» или «setInterval()» не позволяют отделить расчет данных игровой логики от их отрисовки, что приводит к тому, что на медленных компьютерах видеоигра будет казаться слишком медленной, а на мощных - слишком быстрой.

7. Тестируйте реализованные алгоритмы, не забывая проверять уникальные частные случаи.

ЗАКЛЮЧЕНИЕ

В данном проекте были исследованы и разработаны на языке JavaScript алгоритмы для обработки коллизий, корректировки диагонального движения, и других ключевых аспектов разработки двумерных видеоигр. Работа проводилась в два этапа: сначала анализировалась математическая теория, лежащая в основе алгоритмов, затем осуществлялся ее перенос на язык JavaScript. Такой подход продемонстрировал упрощение процесса разработки, подтвердив первоначальную гипотезу о том, что освоение конкретных математических концепций делает процесс разработки более доступным.

Практическая реализация алгоритмов показала, что базовые знания векторной алгебры и вычислительной геометрии позволяют решать многие типичные проблемы, с которыми сталкиваются начинающие разработчики.

В рамках проекта был создан продукт в виде двумерной видеоигры, которая является подтверждением работоспособности разработанных решений. Публикация исходного кода и рекомендаций предоставляет начинающим разработчикам готовый инструментарий и методику для самостоятельного применения.

В результате работы над проектом были сделаны следующие выводы:

1. Реализованные алгоритмы подтвердили, что без базовых знаний векторной алгебры и вычислительной геометрии процесс разработки даже простой двумерной видеоигры становится непредсказуемым и бессистемным.

2. Созданная библиотека решений на языке JavaScript доказывает, что использование готовых математических методов экономит время и уменьшает количество ошибок.

3. Разработанная интерактивная видеоигра эффективно выполняет образовательную функцию, наглядно демонстрируя применение математических концепций в реальных игровых сценариях, что особенно ценно для визуального обучения начинающих разработчиков.

Таким образом, проект достиг своей цели, показав, что целенаправленное освоение математических основ игровой разработки не только помогает преодолеть типичные трудности начинающих разработчиков, но и открывает возможности для создания более сложных и стабильных игровых систем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Современный учебник JavaScript. – Текст: электронный: [Электронный ресурс]. – Режим доступа: [https://learn.javascript.ru](https://learn.javascript.ru/)
2. Томпсон Дж. Collision Detection: Polygon/Polygon. – Текст: электронный: [Электронный ресурс]. – Режим доступа: <https://www.jeffreythompson.org/collision-detection/poly-poly.php>
3. Уитни Д. Программирование для детей. Пять самых крутых игр на HTML и JavaScript. – СПб.: Питер, 2020. – 224 с.
4. Уитни Д. Программирование для детей. Учимся создавать сайты, приложения и игры. HTML, CSS и JavaScript. – СПб.: Питер, 2021. – 208 с.
5. 2D collision detection. – Текст: электронный // MDN Web Docs: [Электронный ресурс]. – Режим доступа: <https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection>
6. How to Fix Diagonal Movement in 2D Top-Down Games. – Текст: электронный // JSLegendDev’s Substack: [Электронный ресурс]. – Режим доступа: <https://jslegenddev.substack.com/p/how-to-fix-diagonal-movement-in-2d>
7. JavaScript. – Текст: электронный // MDN Web Docs: [Электронный ресурс]. – Режим доступа: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
8. Point in polygon. – Текст: электронный // Wikipedia: [Электронный ресурс]. – Режим доступа: <https://en.wikipedia.org/wiki/Point_in_polygon>