ВВЕДЕНИЕ

[СЛАЙД 1] 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 моего 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.

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 многоугольника.

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 этих алгоритмов и упрощение процесса разработки. Видеоигра и ее исходный код доступны в моем репозитории на сайте github.com.

5. Предложить рекомендации для начинающих разработчиков видеоигр.

1.4. Архитектура двумерных видеоигр на языке JavaScript

Прежде чем перейти к рассмотрению проекта, стоит сначала разобраться в ключевых компонентах, 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, 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 проекта были созданы при помощи холста из JavaScript.

[СЛАЙД 2] При работе с холстом из JavaScript 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.

1.4.2. Игровой цикл и отрисовка

[СЛАЙД 3] 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: расчет данных кадра и отрисовка этих данных.

В первой фазе осуществляется пересчет данных игровой логики, например:

1. Обновление позиций объектов.

2. Обработка коллизий.

3. Генерация новых объектов.

4. Обновление игровых параметров.

5. Проверка условий победы и поражения.

6. 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.

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.

Рассмотрим 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 выравненный по осям прямоугольник, и поэтому часто приходится использовать алгоритм, проверяющий коллизию объектов таких форм.

[СЛАЙД 4] Для начала определим, что ориентированным по осям прямоугольником является прямоугольник, стороны которого параллельны осям координат.

[СЛАЙД 5] Чтобы понять, как работает этот алгоритм, проведем через стороны прямоугольника прямые, которые будут создавать по две полуплоскости, а затем через пересечение определенных полуплоскостей найдем область, занимаемую прямоугольником.

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. [СЛАЙД 6] При верхней прямой прямоугольник окажется в нижней полуплоскости.

2. [СЛАЙД 7] При правой прямой прямоугольник окажется в левой полуплоскости.

3. [СЛАЙД 8] При нижней прямой прямоугольник окажется в верхней полуплоскости.

4. [СЛАЙД 9] При левой прямой прямоугольник окажется в правой полуплоскости.

Пошаговое пересечение выбранных полуплоскостей будет постепенно формировать область, занимаемую прямоугольником [СЛАЙДЫ 10 - 12].

[СЛАЙД 13] 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 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.

[СЛАЙД 14] Для реализации этого алгоритма была создана функция «isPointInsideNotRotatedRectangle()», внутри которой указанное условие было перенесено на язык JavaScript.

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.

[СЛАЙД 15] 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.

[СЛАЙД 16] Этот алгоритм звучит достаточно просто и хорошо справляется с базовыми случаями. 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.

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.

Поскольку в проекте был выбран способ, использующий линейную интерполяцию, то про нее стоит кратко рассказать.

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

Общий порядок работы линейный интерполяции такой:

1. Вычисляем длину отрезка между двумя конечными точками отрезка.

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

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

[СЛАЙД 17] Поскольку для данной реализации алгоритма проверки нахождения точки внутри многоугольника требуется использование линейной интерполяции, то была создана вспомогательная функция, которая находит точки на отрезке при помощи линейной интерполяции.

[СЛАЙД 18] 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 специальная функция.

Для итоговой реализации алгоритма проверки нахождения точки внутри многоугольника была создана функция «isPointInsidePolygon()». Логика работы этой функции следующая:

1. [СЛАЙД 19] Проверяем не совпадает ли точка с одной из вершин многоугольника.

2. [СЛАЙД 20] Проверяем не находится ли точка снаружи «ограничивающей коробки» многоугольника.

3. [СЛАЙД 21] Находим точки на отрезке луча.

4. [СЛАЙД 22] Находим точки на каждой стороне многоугольника.

5. [СЛАЙД 23] Перебираем точки на отрезке луча и точки на сторонах многоугольника, сохраняя все совпадающие точки в массив.

6. [СЛАЙД 24] В этом массиве удаляем «соседние» точки, чтобы избежать случаев, когда какая-то сторона многоугольника совпадает с отрезком луча.

7. [СЛАЙД 25] Проверяем не получается ли так, что все точки в упомянутом массиве совпадают с вершинами многоугольника.

8. [СЛАЙД 26] Проверяем количество пересечений отрезка луча и сторон многоугольника на нечетность.

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.

[СЛАЙД 27] 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.

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.

[СЛАЙД 28] Векторное и скалярное произведения двух векторов, а также квадрат длины вектора находятся следующим образом.

[СЛАЙД 29] Для расчета векторного и скалярного произведений были созданы специальные функции.

[СЛАЙД 30] Для расчета квадрата длины вектора также была создана специальная функция.

[СЛАЙД 31] Геометрический смысл векторного произведения двух векторов, на примере точки P и прямой, содержащей отрезок AB, 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.

[СЛАЙД 32] Для определения положения точки относительно прямой, содержащей отрезок, используя геометрический смысл векторного произведения двух векторов, была создана специальная функция.

[СЛАЙД 33] Геометрический смысл скалярного произведения двух векторов, на примере векторов AB и AP, 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.

[СЛАЙД 34] Алгоритм проверки нахождения точки на отрезке 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.

[СЛАЙД 35] Для определения находится ли точка на отрезке была создана специальная функция. Логика работы этой функции 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.

[СЛАЙД 36] Алгоритм проверки пересечения двух отрезков сводится к общему случаю и частным случаям.

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.

Для определения пересекаются ли два отрезка была создана специальная функция. Логика работы этой функции следующая:

1. [СЛАЙД 37] Определяем положение конечных точек отрезков относительно прямых, содержащих противоположные отрезки.

2. [СЛАЙД 38] Проверяем общий случай пересечения двух отрезков.

3. [СЛАЙД 39] Проверяем частные случаи пересечения двух отрезков.

4. [СЛАЙД 40] Если ни одна проверка не прошла, то это означает, что отрезки не пересекаются.

Для итоговой реализации алгоритма проверки пересечения двух многоугольников была создана функция «doTwoPolygonsIntersect()». Логика работы этой функции следующая:

1. [СЛАЙД 41] Перебираем вершины многоугольников и ищем их совпадения.

2. [СЛАЙД 42] Перебираем пары соседних вершин каждого многоугольника, чтобы на их основе проверить не пересекаются ли какие-то стороны этих многоугольников.

3. [СЛАЙД 43] Если стороны не пересекаются, то проверяем не находится ли один многоугольник полностью внутри другого.

4. [СЛАЙД 44] Если ни одна проверка не прошла, то это означает, что многоугольники не пересекаются.

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.

Рассмотрим почему возникает указанная проблема и как ее решить при помощи нормализации вектора скорости.

1.2.1. Нормализация вектора скорости

[СЛАЙД 45] 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 вправо 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. 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. [СЛАЙД 46] 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.

Чтобы скорость диагонального движения была корректной необходимо использовать следующий 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.

[СЛАЙД 47] Для корректировки скорости диагонального движения объектов при помощи нормализации вектора скорости такого движения была создана функция «correctDiagonalMovementSpeed()».

1.3. Вспомогательные алгоритмы

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

1.3.1. Генерация координат вершин многоугольника

[СЛАЙД 48] Для реализации объектов произвольной формы был разработан алгоритм, создающий координаты вершин многоугольника в заданной области:

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.

Для того, чтобы можно было создавать многоугольники в видеоигре, была создана функция «preparePolygonIntVerticesData()», которая рассчитывает координаты вершин многоугольника в рамках указанной области. Логика работы этой функции следующая:

1. [СЛАЙД 49] Определяем сколько примерно вершин должно быть в каждой четверти области для генерации вершин.

2. [СЛАЙД 50] Рассчитываем первую координату вершины.

3. [СЛАЙД 51] Определяем в какой четверти производим расчет вершин:

1) [СЛАЙД 52] Если в первой, то рассчитываем координаты остальных вершин в первой четверти области.

2) [СЛАЙД 53] Если во второй, то рассчитываем координаты вершин во второй четверти области.

3) [СЛАЙД 54] Если в третьей, то рассчитываем координаты вершин в третьей четверти области.

4) [СЛАЙД 55] Если в четвертой, то рассчитываем координаты вершин в четвертой четверти области.

4. [СЛАЙД 56] Выравниваем рассчитанные координаты вершины по сетке, если необходимо.

5. [СЛАЙД 57] Корректируем рассчитанные координаты вершины, выходящие за пределы области.

6. [СЛАЙД 58] Рассчитав и откорректировав координаты вершины, добавляем ее в массив.

7. [СЛАЙД 59] Проверяем не нужно ли для расчета координат следующей вершины перейти в следующую четверть области и возвращаемся к шагу 3.

8. [СЛАЙД 60] Рассчитав все вершины, возвращаем массив объектов, содержащих рассчитанные координаты вершин.

2.4. Рекомендации для начинающих разработчиков двумерных видеоигр на языке JavaScript

В заключение можно предложить следующие 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

[СЛАЙД 61] На основе опыта, полученного при работе над данным проектом, можно предложить 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.

[СЛАЙД 62] 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

[СЛАЙД 63] Также можно предложить 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 файле или объекте, 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

Работа над этим проектом 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. [СЛАЙД 64]