Goerging 5 мая 2024 в 01:57

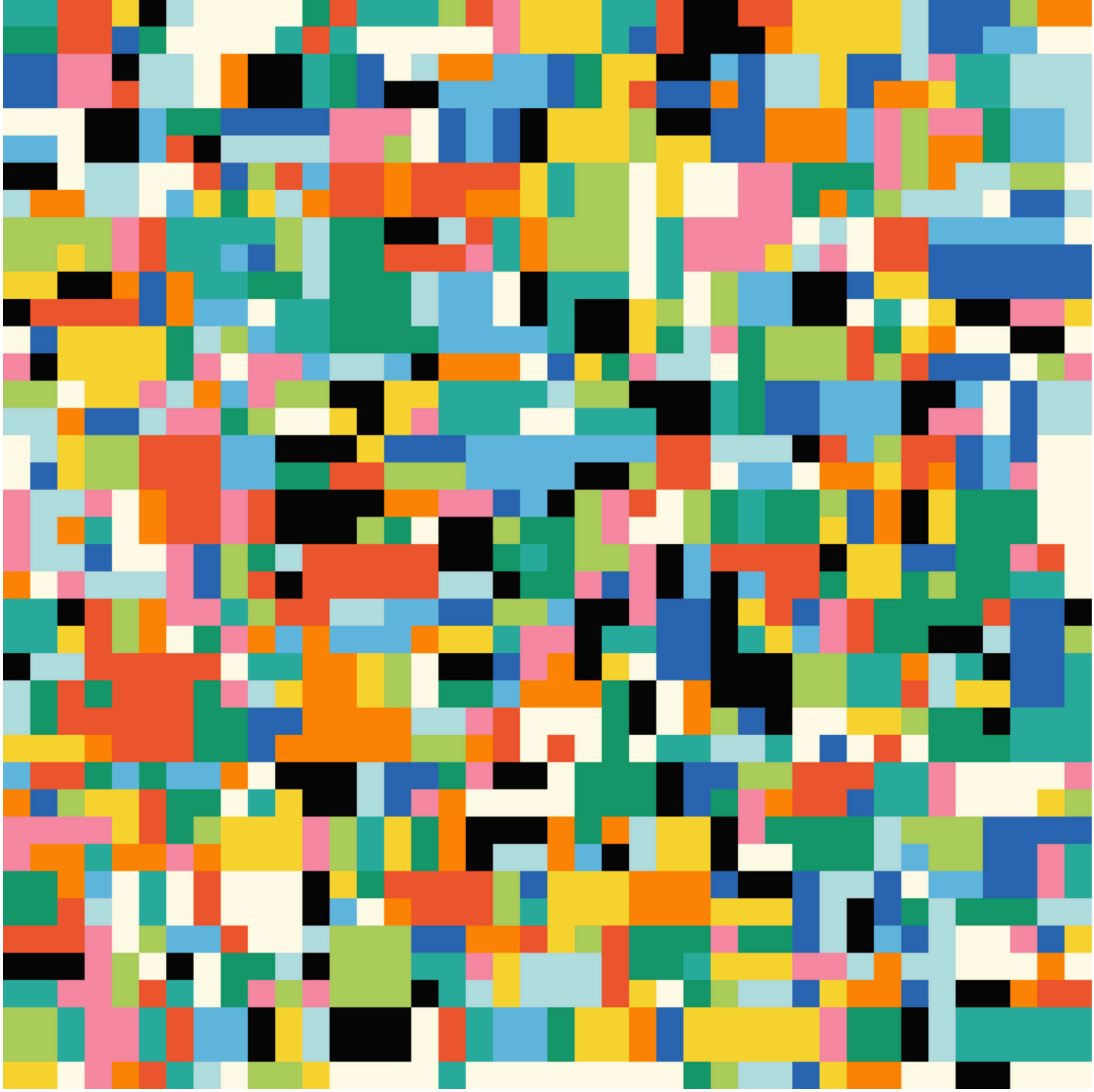
Алгоритм пересечения полигонов

20 мин **11K**

Алгоритмы*, Математика*

Туториал Перевод

Автор оригинала: Ahmad Moussa



В этом посте мы разработаем алгоритм, позволяющий вычислять пересечение выпуклых полигонов, а так же на ряду с проверкой точки на принадлежность полигону мы рассмотрим метод пересечения выровненных по осям прямоугольников и функцию пересечения отрезков.

Огромная благодарность Дэйву за помощь в обсуждении вопроса через дискорд!

Изначально эта статья задумывалась как небольшой пост на тему методов, способных определять пересечение прямоугольников - как выровненных по осям так и произвольно повернутых. Однако работая над недавним проектом, я заметил, что проверка на пересечение прямоугольников может быть доработана так, чтобы возвращать не только образуемый пересечением двух прямоугольников полигон, но и таковой, образуемый пересечением двух выпуклых многоугольников. Так что я расширил пост и включил в него и эту тему.

Вместе с методом определения того, находится ли точка внутри полигона (проверка точки на принадлежность полигону), мы рассмотрим вычисление пересечения двух отрезков. Все это

является важной частью способа определения пересечения полигонов. Получается, этот пост охватывает немалое количество материала, но для полноты картины я бы не хотел что-либо упускать.

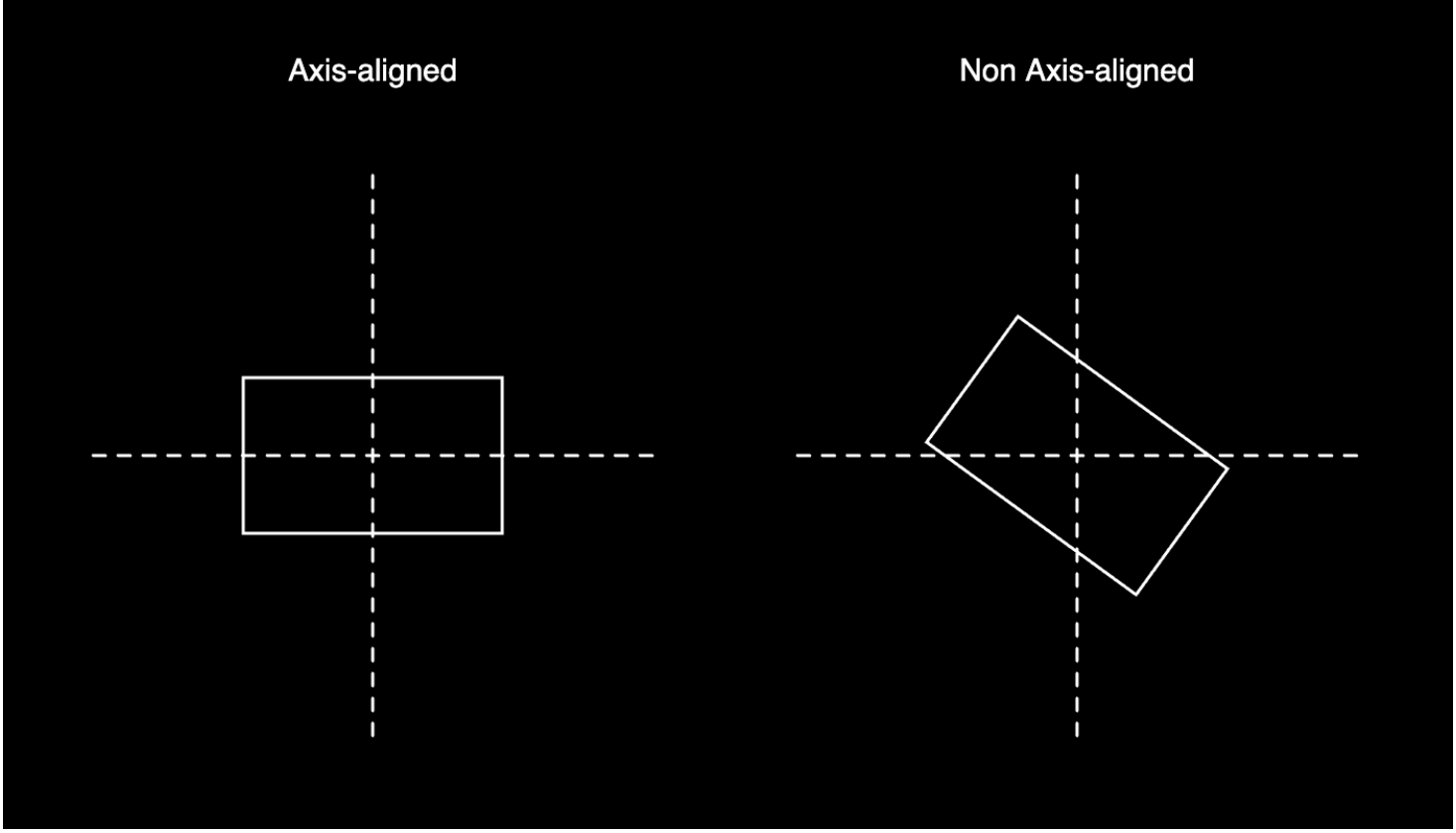
Перед тем как начать, немного о терминологии. По большей части я буду использовать слова **"пересечение"** и **"коллизия"** взаимозаменяемо, так как по существу они означают одно и то же: два прямоугольника касаются или накладываются друг на друга каким-либо образом, однако слово **"коллизия"** подразумевает движение, и, вероятно, лучше вписывается в контекст.

Описанный в этом посте метод может и не самый эффективный для большого количества прямоугольников, зато он компенсирует это своей простотой. Ниже указатель на разные разделы этой статьи:

Содержание

Что значит "выровненный по осям"?

Термин "выровненный по осям" означает, что ориентация фигуры соответствует направлению осей. Например, у выровненного по осям прямоугольника стороны будут параллельны осям x и y. Что-то не выровненное может иметь другую ориентацию, например, быть повернутым относительно координатных осей. Вроде такого:



Работа с выровненными по осям прямоугольниками сильно упрощает некоторые задачи, как, например, определение пересечения между ними. Я уже упоминал об этом в разделе предыдущей статьи: A Simple Solution for Shape Packing in 2D, однако также выровненные по осям прямоугольники невероятно полезны, когда речь заходит о быстром определении коллизии того или иного рода. Выявление коллизий скорее типичная проблема для игр, где, обычно, скорость алгоритма для определения этих самых коллизий имеет решающее значение. Популярный подход в играх - разделение проблемы на два этапа: "грубый", на котором мы определяем потенциально пересекающиеся объекты, и "точный", в который затем переходят эти объекты, где уже и происходит проверка на коллизию. Выровненные по осям ограничительные рамки (прямоугольники) *[Axis-aligned bounding boxes]* часто используются в качестве "грубого" этапа алгоритма, так как вычисление их коллизии относительно дешево. В следующем разделе мы посмотрим, как это работает!

Коллизия выровненных по осям прямоугольников

Прямоугольник может быть представлен четырьмя числами, два из которых означают его **x** и **y** координату и обычно указывают на левый верхний угол, и два других, означающих его длину и высоту, начиная с координаты. Получается, в пересечении двух выровненных по осям прямоугольников можно быть уверенным тогда, когда выполнены четыре следующих условия:

$$x_{R1} < x_{R2} + width_{R2}$$

Левая граница R1 меньше, чем **правая** граница R2

$$y_{R1} < y_{R2} + height_{R2}$$

Верхняя граница R1 меньше, чем **нижняя** граница R2

$$x_{R2} < x_{R1} + width_{R1}$$

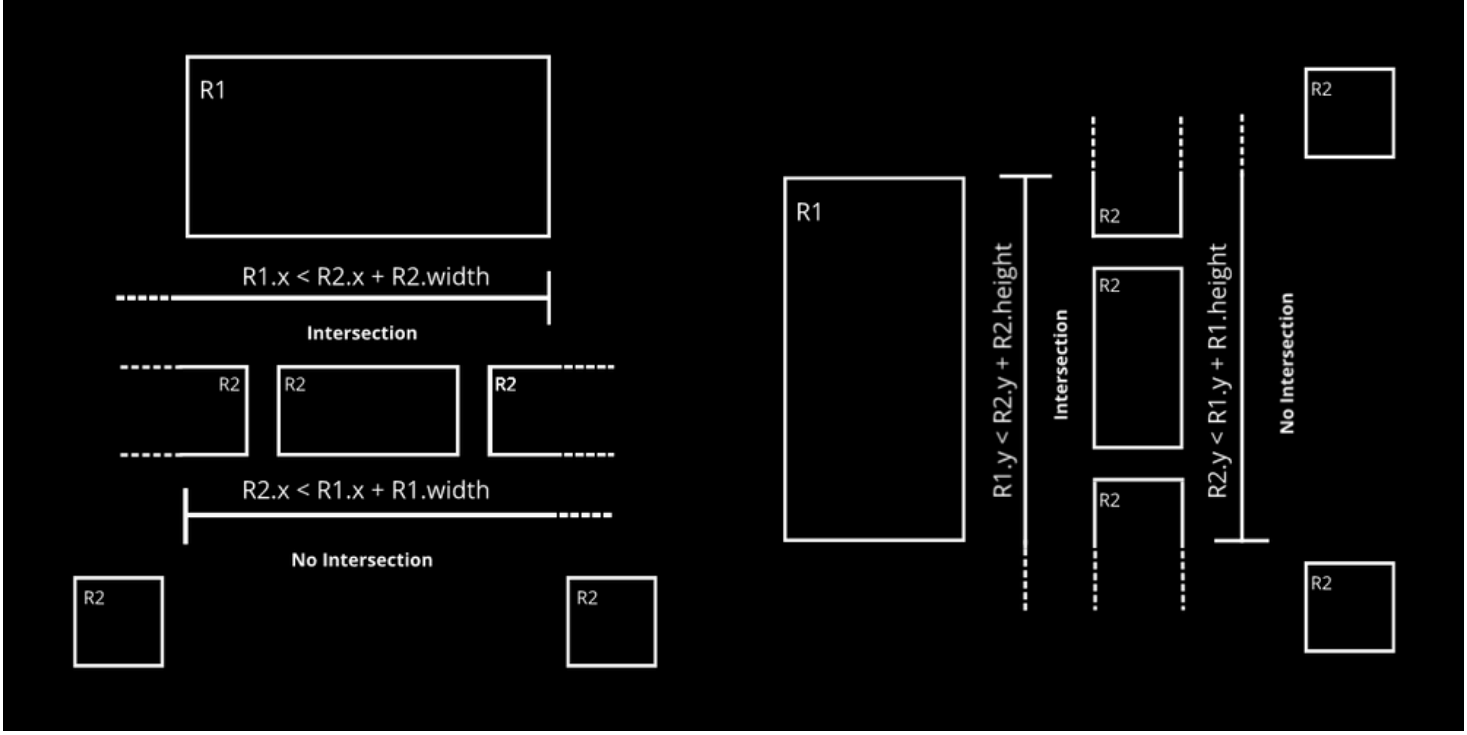
Левая граница R2 меньше, чем **правая** граница R1

$$y_{R2} < y_{R1} + height_{R1}$$

Верхняя граница R2 меньше, чем **нижняя** граница R1

Увидев это впервые можно немного опешить, но по существу эти проверки определяют, как границы двух прямоугольников расположены друг относительно друга. В таком виде пересечение вдоль оси может быть определено двумя условиями: вдоль оси x например, проверяется, что левая граница R1 где-то левее правой границы R2, и левая граница R2 где то левее правой границы R1. Объединяя оба условия, мы, по факту, проверяем, что хотя бы одна граница одного прямоугольника находится между границами другого прямоугольника - в таком случае прямоугольники в каком-то роде накладываются. В более общем смысле это означает, что два прямоугольника одновременно охватывают общий диапазон оси x.

Если рассматривать одно из этих условий отдельно от другого, то это мало что скажет о положении прямоугольников, но если оба условия выполняются, то можно сказать, что они пересекаются хотя бы вдоль оси x. Дальше мы должны сделать то же самое вдоль оси y, чтобы точно определить, пересекаются ли прямоугольники на двумерной плоскости. Если это было непонятно, возможно, картинка ниже поможет:



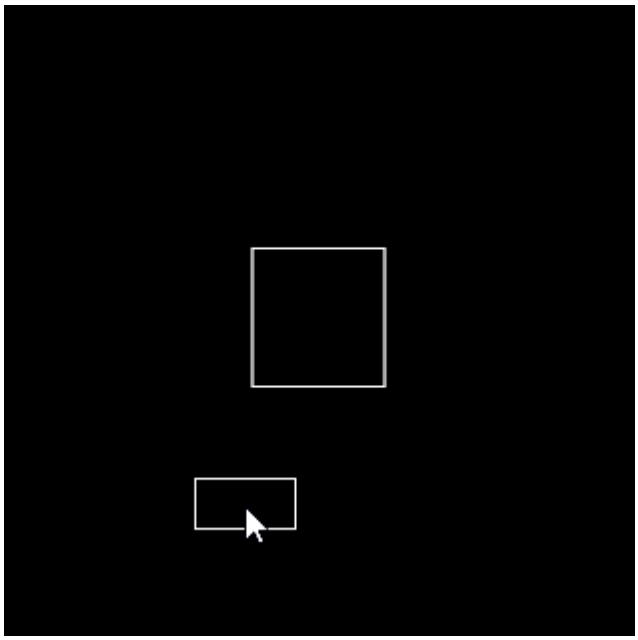
А теперь пора бы написать код для этого всего! Чтобы сделать все немного понятнее мы создадим наш собственный класс прямоугольника, которому потом дадим возможность определять коллизии с другими прямоугольниками. Для представления простого прямоугольника нам не требуется больше того, что было упомянуто ранее: просто две координаты для хранения позиции в дополнение к размерам. В случае выровненных по осям прямоугольников имеет смысл представлять координатами левый верхний угол, так как это просто на просто делает вычисления короче. Также обычно это стандартный **rectMode()** в p5js:

```
// Пример минимального класса прямоугольника
function makeRect(posx, posy, wid, hei){
  this.posx = posx
  this.posy = posy

  this.wid = wid
  this.hei = hei

  // Функция рисования прямоугольника на холсте
  this.display = function(){
    rect(this.posx, this.posy, this.wid, this.hei)
  }
}
```

Теперь о функции, которая позволит вычислять пересечение нашего прямоугольника с другим. Она принимает на вход объект другого прямоугольника, относительно которого нам надо понять положение нашего:

```
this.checkCollision = function(otherRect){
  if( this.posx < otherRect.posx + otherRect.wid &&
      this.posx + this.wid > otherRect.posx &&
      this.posy < otherRect.posy + otherRect.hei &&
      this.posy + this.hei > otherRect.posy
    ){
    return true
  }
  return false
}
```

Метод использует в точности ту логику, которую мы обсуждали немного ранее. Функция возвращает `true` если прямоугольники пересекаются, и `false` в ином случае. Давайте проверим все это на простом примере: наведите свой указатель на окошко и установите его на покоящийся прямоугольник, тогда если все в порядке, то перетаскиваемый прямоугольник загорится красным, когда те двое пересекаются:



[в оригинале это интерактивная вставка p5js]

▸ Код для p5js

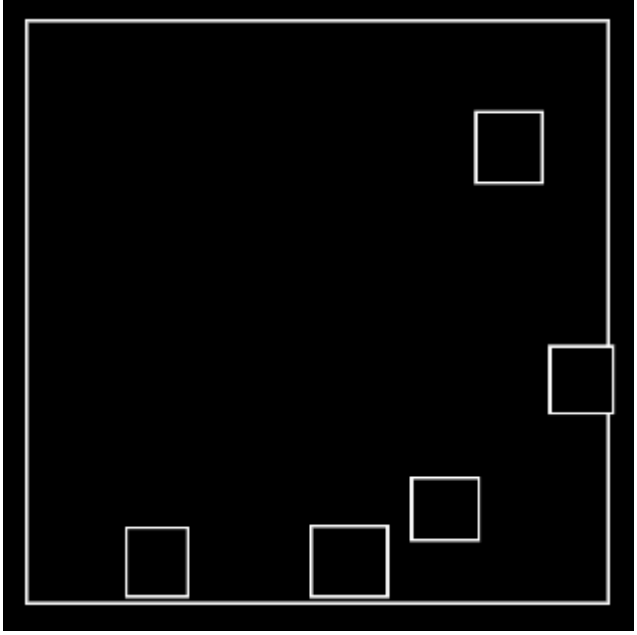
На этом мы могли бы остановится, но что если мы захотим создать несколько прямоугольников? Тогда есть смысл создать обработчик, который следит за разными экземплярами, отвечает за их движение и проверку коллизий между ними. Обработчик будет выглядеть примерно как в этом кусочке кода, где прямоугольники хранятся в его переменной-члене:

```
function makeCollisionHandler(){
  this.rectangles = []
```

```
  this.createRectangles = function(num){
    for(let n = 0; n < num; n++){
      let wid = random(100,200)
      let hei = random(100,200)
      this.rectangles.push(new makeRect(random(pad,wx-pad-wid), random(pad, wy-pad-hei), wid
    }
  }

  this.checkCollisions = function(){
    for(let n = 0; n < this.rectangles.length; n++){
      let currRect = cR = this.rectangles[n]

      for(let k = 0; k < this.rectangles.length; k++){
        let otherRect = oR = this.rectangles[k]

        if(cR.id != oR.id){
          if(cR.checkCollision(oR)){
            cR.col = color(255,0,0)
            oR.col = color(255,0,0)
          }
        }
      }
    }
  }

  this.displayRectangles = function(){
    for(let n = 0; n < this.rectangles.length; n++){
      this.rectangles[n].display()
    }
  }
}
```

Для проверки коллизий между прямоугольниками нам нужен вложенный цикл, который проходит по каждому прямоугольнику и сопоставляет его с другими прямоугольниками из массива.

Однако для гораздо большего количества прямоугольников, вероятно, лучше реализовать поиск по сетке, вроде того, что обсуждался в статье "A Simple Solution for Shape Packing in 2D". Теперь мы можем собрать все в кучу, чтобы получить что-то на подобие такого:



[в оригинале это интерактивная вставка p5js]

- Код для p5js

Также здесь я дал прямоугольникам возможность прыгать по экрану прям как у известного символа DVD:

```
this.move = function(){
  this.posx = this.posx + this.xspeed
  this.posy = this.posy + this.yspeed

  if(this.posx + this.wid >= wx -pad || this.posx <= 0+pad){
    this.xspeed = -this.xspeed
  }

  if(this.posy + this.hei >= wy-pad || this.posy <= 0+pad){
    this.yspeed = -this.yspeed
  }
}
```

Ладно, в этом заключается первая часть статьи, и хочется верить, что с помощью рассмотренного можно сделать многое! В следующем разделе мы посмотрим на прямоугольники, которые не выровнены по осям, и подготовим основу для расчета их пересечений.

Отрисовка произвольных прямоугольников

Давайте для начала напишем немного кода, который позволит правильно позиционировать и рисовать на холсте повернутые прямоугольники. Позже, для вычисления пересечения между

ними, нам будет нужен легкий доступ к координатам вершин, формирующих четыре угла прямоугольника (подробнее об этом в следующем разделе). Опять таки мы сделаем это в объектно-ориентированном стиле и создадим класс для наших повернутых прямоугольников:

```
function makeRect(posx, posy, wid, hei, angle) {
  this.posx = posx
  this.posy = posy

  this.wid = wid
  this.hei = hei

  this.angle = angle
  this.vertices = []
}
```

Для повернутых прямоугольников имеет больший смысл указывать координатой позиции на центр прямоугольника, нежели на левый верхний угол, что позже сильно упростит вращение его вершин. В дополнение к этому мы можем предположить, что прямоугольник располагается своим центром в начале координат, тогда затем нам просто надо будет передвинуть координаты в фактическую позицию прямоугольника. Теперь давайте заполним массив **vertices** координатами четырех углов, не подвергавшихся вращению:

```
// создает ровные координаты вокруг центра
this.makeCoordinates = function() {
  this.vertices = []

  // знали ли вы, что вы можете передавать несколько значений в метод push() объекта array?
  this.vertices.push(
    { x: this.wid/2, y: this.hei/2 },
    { x: - this.wid/2, y: this.hei/2 },
    { x: - this.wid/2, y: - this.hei/2 },
    { x: this.wid/2, y: - this.hei/2 }
  )
}
this.makeCoordinates()
```

Возможно это выглядит не лучшим образом, но видимо это простейший способ. Менее читабельная альтернатива метода создания координат - это использование параметрических уравнений, так как прямоугольник - это, по сути, окружность с четырьмя точками:

```
for(let a = 0; a <= TAU; a+=TAU/4){this.vertices.push({x: this.wid * Math.sign(cos(a)), y: t
```

Используйте тот способ, что вам больше по душе. Обратите внимание, что этот параметрический метод не учитывает угол вращения, он будет фиксирован на углах для вершин. Для обоих методов нам все еще нужен дополнительный шаг, чтобы учесть угол вращения. Как вариант, это можно сделать умножением координаты на матрицу вращения, где тета - это угол поворота:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{aligned} x_{rotated} &= x_{orig}.cos(\theta) - y_{orig}.sin(\theta) \\ y_{rotated} &= x_{orig}.sin(\theta) + y_{orig}.cos(\theta) \end{aligned}$$

В виде функции на javascript это будет выглядеть следующим образом:

```
// провращать координаты после поворота
this.computeRotation = function() {
  // новый массив для хранения повернутых координат
  let rotatedCoordinates = []

  for (let n = 0; n < this.vertices.length; n++) {

    // получим неповернутую координату
    let vert = this.vertices[n]

    // посчитаем значения из матрицы
    var c = cos(this.angle);
    var s = sin(this.angle);

    // применим матрицу поворота
    var xr = c * tempX - s * tempY;
    var yr = s * tempX + c * tempY;

    // переместим координаты в фактическую позицию прямоугольника
    var xr = xr + this.posx;
    var yr = yr + this.posy;

    rotatedCoordinates.push({ x: xr, y: yr })
  }
  return rotatedCoordinates
}
```

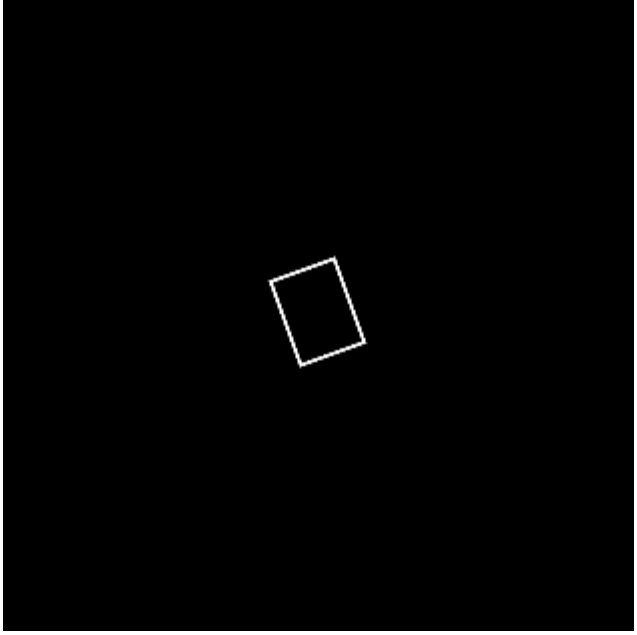
Теперь массив **rotatedCoordinates** содержит фактические координаты повернутых вершин прямоугольника в системе отсчета холста. В качестве альтернативы использованию матрицы поворота вы так же можете перевести координаты в полярные, прибавить угол вращения и затем преобразовать обратно в Декартовы, однако тут слишком много лишних шагов. Пусть и немного менее читабельно, но мы можем переписать все в более компактный вид следующим образом:

```
function rotateRectangle(posx, posy, wid, hei, angle){
  vertices = []

  let [c, s] = [cos(angle), sin(angle)]
  for(let a = 0; a <= TAU; a+=TAU/4){
    let [x, y] = [wid * Math.sign(cos(a)), hei * Math.sign(sin(a))]
    vertices.push(
      {
        x: x * c - y * s + posx,
        y: x * s - y * c + posy
      }
    )
  }
  return vertices
}
```

Последняя альтернатива, которую я могу предложить, - это параметрические уравнения для квадрокруга. Параметрические уравнения квадрокруга можно найти в четвертом разделе этой статьи. Хотя для нашей задачи это, конечно, лишнее. И теперь, наконец, мы можем нарисовать повернутый прямоугольник, итерируя ранее вычисленные координаты:

```
this.display = function(){
  let rotatedCoords = this.computeRotation()
  beginShape()
  for(let n = 0; n < 4; n++){
    let v = rotatedCoords[n]
    vertex({x: v.x, y: v.y})
  }
  endShape(CLOSE)
}
```

Получилось неплохо, потому что если мы захотим поменять некоторые параметры прямоугольника, например его длину или высоту, можно просто изменить соответствующую переменную-член и вызвать две функции: **makeCoordinates()** и **computeRotation(),** и получить новый прямоугольник. Вот пример:



[в оригинале это интерактивная вставка p5js]

- Код для p5js

Далее нас ждет пересечения между двумя подобными прямоугольниками!

Пересечение двух отрезков

Зачем нам надо знать, как вычислять пересечение отрезков? На самом деле прямоугольник - это не более чем множество из 4 отрезков. Это сводит задачу пересечения прямоугольников к проверке, не пересекаются ли два отрезка, принадлежащих разным прямоугольникам. В этом разделе перед нами будет стоять задача реализовать функцию, определяющую пересечение отрезков.

На самом деле мы одолжим функцию пересечения отрезков, которую я изначально нашел в этом ответе на stackoverflow. Математические расчеты, стоящие за этой функцией, были проделаны Paul Bourke в его заметках. Реализована же в JavaScript она была Leo Bottaro здесь. Этот отрывок пригодился в бесчисленном количестве случаев. Он не только определяет, пересекаются ли два отрезка, но и вычисляет точку пересечения отрезков, что пригодится позже, когда мы будем вычислять полигон, образованный пересечением двух прямоугольников. Сам код:

```
/*
  Line intercept math by Paul Bourke http://paulbourke.net/geometry/pointlineplane/

  - Returns the coordinate of the intersection point
  - Returns FALSE if the lines don't intersect

  Coordinates x1, y1, x2 and y2 designate the start and end point of the first line
  Coordinates x3, y3, x4 and y4 designate the start and end point of the second line
```

```
*/
function intersect(x1, y1, x2, y2, x3, y3, x4, y4) {

  // Check if none of the lines are of length 0
  if ((x1 === x2 && y1 === y2) || (x3 === x4 && y3 === y4)) {
      return false
  }

  denominator = ((y4 - y3) * (x2 - x1) - (x4 - x3) * (y2 - y1))

  // Lines are parallel
  if (denominator === 0) {
      return false
  }

  let ua = ((x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3)) / denominator
  let ub = ((x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3)) / denominator

  // is the intersection along the segments
  if (ua < 0 || ua > 1 || ub < 0 || ub > 1) {
      return false
  }

  // Return a object with the x and y coordinates of the intersection
  let x = x1 + ua * (x2 - x1)
  let y = y1 + ua * (y2 - y1)

  return {x, y}
}
```

Давайте немного пройдемся по тому, что здесь происходит! Наша начальная точка (ага, каламбур) - это описать прямые как кривые Безье первого порядка. По факту это означает, что мы представляем их как функции линейной интерполяции. Линейная интерполяция - это метод, позволяющий находить новые точки на прямой посредством перемещения при помощи параметра от начальной точки к конечной. Таким образом, любая точка на этой прямой может быть представлена как значение функции от двух точек, которая образует отрезок прямой, умноженное на параметр интерполяции. Этот параметр определяет, где на отрезке лежит итоговая точка (как далеко от каждой точки). Давайте напишем это:

$$P_a = P_1 * u_a + P_2 * (1 - u_a)$$

По сути, чем ближе параметр к 1, тем итоговая точка ближе к P1, и наоборот. Это также можно переписать в виде:

$$P_a = P_1 + u_a * (P_2 - P_1)$$

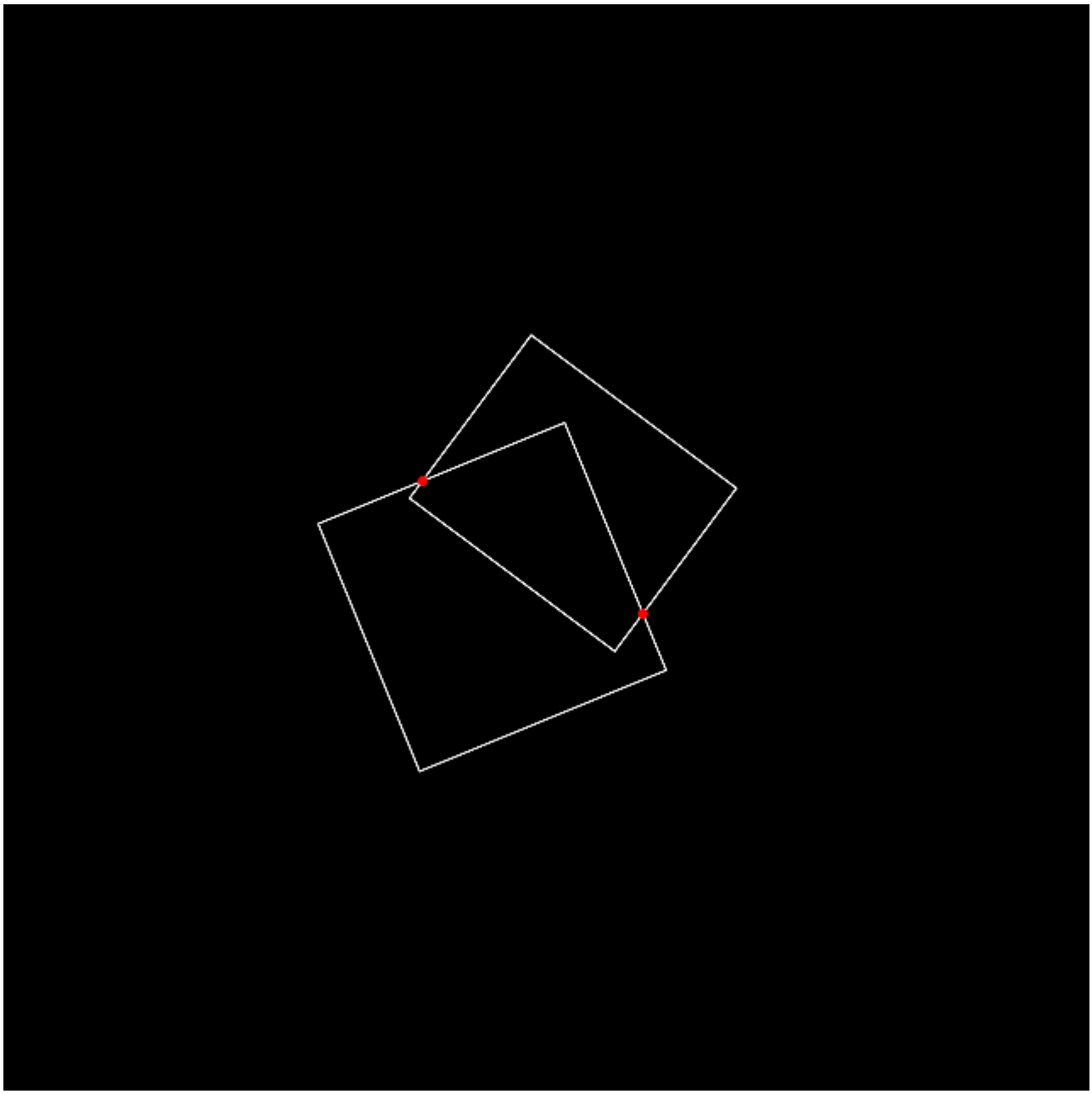
С этими формулами и понимаем, что точка пересечения - это точка, принадлежащая обоим отрезкам, нам осталось решить уравнение, в котором обе интерполяции указывают на одну точку. Это вытекает в задачу с двумя уравнениями и двумя неизвестными, в качестве неизвестных выступают параметры интерполяции. Здесь все сводится к манипуляциям с уравнениями и использованию в одном другого, это слегка длинновато и видно в коде. Тут также есть дополнительный код, проверяющий прямые на параллельность или равенство нулю их длины, ведь в обоих случаях точки пересечения не существует.

Теперь мы можем пройтись по обоим множествам ребер, образующих наши прямоугольники, и попарно проверить, не пересекаются ли какие-либо из них. Так же во время этого мы будем сохранять найденные точки пересечения и возвращать их в виде массива, они пригодятся позже. Эта функция-член принимает в качестве параметра объект другого прямоугольника:

```
/*
  Эта функция возвращает массив, содержащий координаты точек пересечения.
  Пустой массив означает, что пересечения нет.
*/
this.collisionCheck = function(otherRect){
  // складываем все точки пересечения в этот массив
  let intersectionPoints = []

  if(this.id == otherRect.id){
    return intersectionPoints
  }

  let intersection = false
  for(let n = 0; n < 4; n++){
    let currLine = this.edges[n]

    for(let k = 0; k < 4; k++){
      let otherLine = otherRect.edges[k]

      let check = currLine.intersects(otherLine)
      if(check){
        intersectionPoints.push(check)
      }
    }
  }
  return intersectionPoints
}
```

Еще одна вещь о которой мы до сих пор не упоминали, состоит в том, что также полезно присвоить id каждому объекту прямоугольника. Это пригодится во время проверки на

пересечения, так как позволит избежать бесполезных проверок по типу пересечения прямоугольника самим с собой. Должна ли эта проверка осуществляться внутри функции или где-то за ее пределами - это другой вопрос. Совместив коды выше, мы можем написать следующий скетч, способный визуализировать места пересечения прямоугольников:

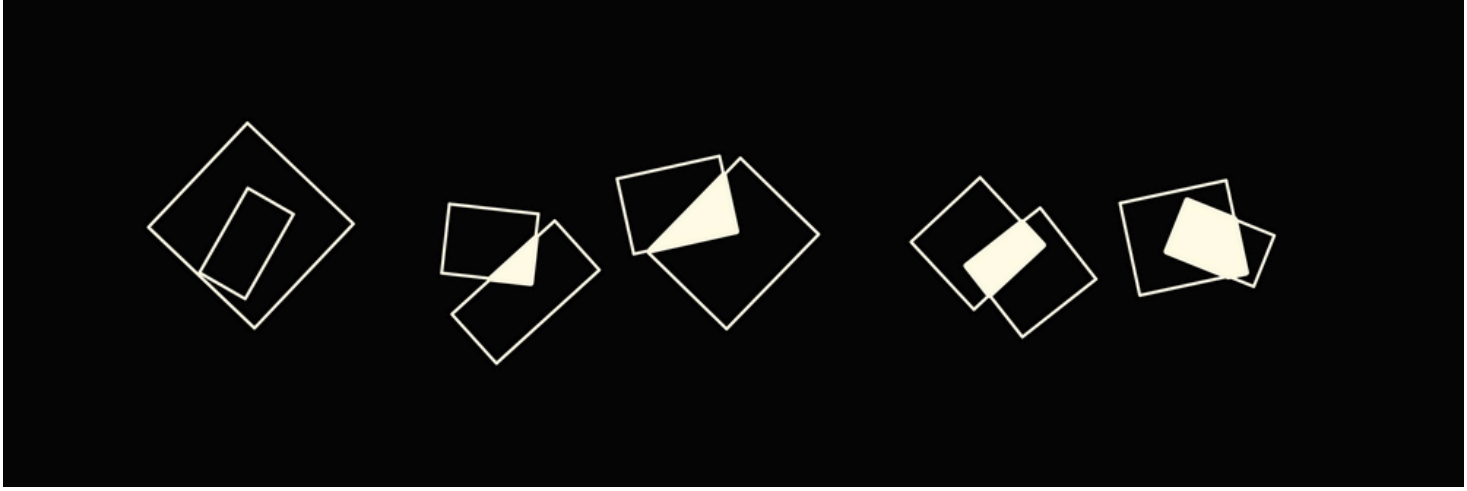


Заимствуйте этот код на openprocessing.

Теперь можно приступить к поиску полигона, образованного пересечением двух прямоугольников. Если вам нужно только определить, пересекаются ли прямоугольники, вы можете на этом остановится. Если же вам надо каким-то образом получить полигон, образуемый двумя прямоугольниками, тут есть еще несколько шагов, которые мы должны сделать.

Проверка точки на принадлежность полигону

Теперь когда у нас есть точки пересечения, как нам определить полигон пересечения? Другими словами, как нам определить вершины, образующие этот полигон?



Если вы посмотрите на фигуры выше или нарисуете подобные на бумаге, вы заметите, что вершины полигона пересечения всегда состоят из точек пересечения и вершин прямоугольников, которые находятся внутри другого. Получается, нам нужно некоторым образом определять, какие вершины прямоугольника #1 находятся в прямоугольнике #2, и наоборот. Также можно заметить, что особый случай - это когда один прямоугольник полностью находится в другом, и разработанный нами к этому моменту метод не сможет определить пересечение. Нам надо будет это учесть.

Получается, что нам надо две вещи, одна из которых это код пересечения отрезков, уже рассмотренный нами в предыдущем разделе, и другая - это функция, позволяющая определять, находится точка внутри прямоугольника или нет. Последняя поможет нам в определении вершин одного прямоугольника, лежащих внутри другого, и наоборот. Обе из них сделают свой вклад в формирование итогового полигона. Для проверки точки на принадлежность прямоугольнику или, на самом деле, любому многоугольнику, мы можем использовать алгоритм, известный как ray-casting. Daniel Schiffman обеспечит неплохое начало в понимании ray-casting'а:

Если вы посмотрите видео то заметите, что наш код пересечения отрезков появляется и тут! Вообще ray-casting полезен в огромном количестве иных ситуаций, и особенно в играх. Для наших задач нам не нужно разрабатывать полноценный пример ray-casting'a как в видео Coding Train, нам нужно только проверить, находится ли точка внутри полигона, с помощью относительно маленькой, но очень хитрой функции.

Подобно функции пересечения отрезков, мы можем найти функцию проверки точки на принадлежность полигону в этом ответе на stackoverflow. Функция была переписана с предыдущей версии на C, описанной W. Randolph Franklin-ом на его странице PNPOLY - Point Inclusion in Polygon Test. Он объясняет все в мельчайших подробностях, как работает функция и как он вообще додумался до этого метода. Сама функция:

```
// vs - это массив вершин, например [[1,0],[1,1],[0,1],[0,0]]
function inside(point, vs) {
    /*
      алгоитм ray-casting'a на основе
      https://wrf.ecse.rpi.edu/Research/Short_Notes/pnpoly.html
    */

    var x = point[0], y = point[1];

    var inside = false;
    for (var i = 0, j = vs.length - 1; i < vs.length; j = i++) {
        var xi = vs[i][0], yi = vs[i][1];
        var xj = vs[j][0], yj = vs[j][1];

        var intersect = ((yi > y) != (yj > y))
            && (x < (xj - xi) * (y - yi) / (yj - yi) + xi);
        if (intersect) inside = !inside;
    }

    return inside;
};
```

Когда я впервые это увидел, то подумал, что эта функция выглядит как непойми что, но она работала идеально! Получав координату и множество верши, она могла определять, находится ли эта координата внутри полигона, образованного этими вершинами. Объяснения Franklin'a дали ключ к понимаю, как это работает:

> Я выпускаю горизонтальный луч (x увеличивается, y фиксирован) из начальной точки и считаю, сколько сторон он пересекает. Каждое пересечение луч переключается между внутренней областью и наружной. Это называется Теоремой Жордана.

Также этот алгоритм известен как "The crossing number algorithm" или "Правило четный-нечетный". По сути, если мы возьмем точку и нарисуем луч в строну, начиная с нашей точки, мы

сможем увидеть, сколько раз этот луч пересекает стороны полигона. Если это число нечетно, то точка лежит внутри полигона, если четно, то снаружи. Мне нравится, как такая достаточно сложная проблема решилась такими простыми наблюдениями.

Основная логика в методе Franklin'a состоит из двух условий, которые сначала проверяют, что у координата точки находится в диапазоне у координат ребра, которое мы рассматриваем, а потом смотрят, что эта точка лежит в полуплоскости, образованной этим ребром. Первое условие проверяется просто:

$$(y_i > y) \neq (y_j > y)$$

Второе условие требует немного большего вовлечения, хорошее объяснение того, как это работает, можно найти здесь. Вторая проверка выглядит следующим образом:

$$x < (x_j - x_i) * (y - y_i)/(y_j - y_i) + x_i$$

Если оба этих условия соблюдаются, можно утверждать, что луч, выходящий из нашей точки, пересекает конкретное ребро, которое мы рассматриваем. Теперь вспомним, что нам нужно запоминать, сколько ребер мы пересекли, но так как у точки есть только два состояния, внутри или снаружи полигона, мы можем просто переключать состояние каждый раз, когда условия выполнены.

Отрисовка полигона пересечения

Хорошо, теперь мы собрали оба кусочка пазла и можем написать функцию, которая возвращает вершины полигона пересечения. Начнем с получения точек пересечения, где встречаются ребра двух прямоугольников:

```
// получаем точки, где пересекаются два прямоугольника
let intersectionPoints = this.collisionCheck(otherRect)

// если нет точек пересечения, то полигона пересечения не сущетсвует
if (intersectionPoints.length == 0) {
  return false
}
```

Дальше мы должны получить вершины прямоугольников, которые лежат в соседнем прямоугольнике, для чего будем использовать подпрограмму ray-casting'a, которую мы обсуждали в предыдущем разделе. Для удобства мы можем использовать JavaScript-метод

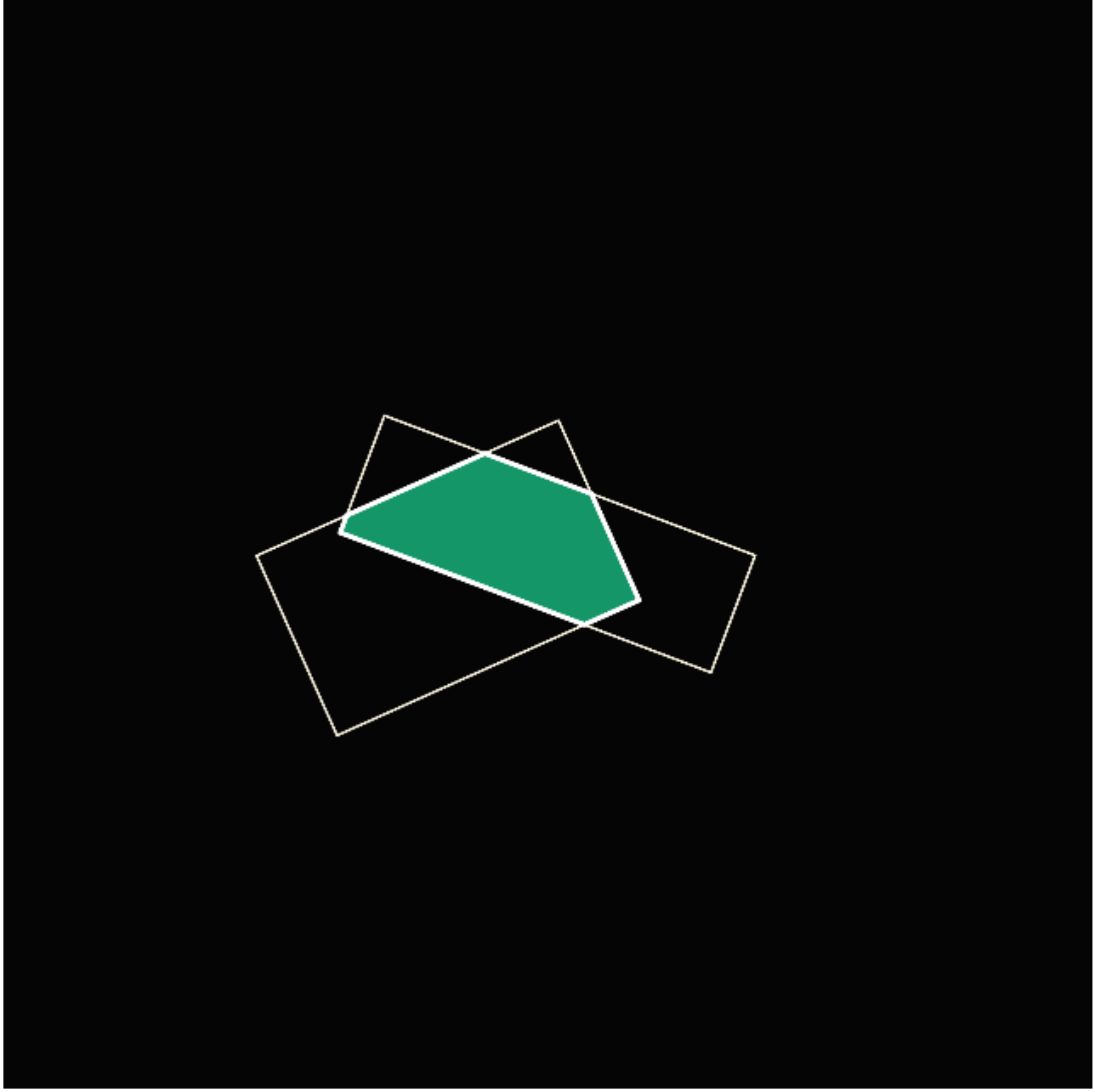
массива **filter()**, который вернет новый массив из элементов, удовлетворяющих условию, переданному в виде функции обратного вызова:

```
// находим точки другого прямоугольника, которые находятся внутри этого
let otherPointsWithinThis = otherRect.computeRotation().filter(v => this.checkIfPointWithin(

// находим точки этого прямоугольника, которые находятся внутри другого
let thisPointsWithinOther = this.computeRotation().filter(v => otherRect.checkIfPointWithin(
```

Теперь на самом деле у нас есть все точки, которые образуют полигон пересечения, однако для корректного его отображения с помощью p5js функций **beginShape()** и **endShape()**, они понадобятся нам в порядке по часовой. Мы можем сделать это, отсортировав их вокруг центральной точки (полученной как среднее между их координатами) по величине угла:

```
// получаем все точки
let allPoints = [...intersectionPoints, ...otherPointsWithinThis, ...thisPointsWithinOther]

// вычисляем координаты центр
let N = allPoints.length
let centerX = allPoints.reduce( (p, c) => p + c.x, 0)/N
let centerY = allPoints.reduce( (p, c) => p + c.y, 0)/N

// вычисляем угол каждой точки от центральной точки
let pointsAndAngs = allPoints.map( p => ({p: p, ang: atan2(centerY - p.y, centerX - p.x)}))

// сортируем точки по величине угла
pointsAndAngs.sort((a, b) => a.ang - b.ang)
```

Заметим, что это работает потому, что прямоугольники это, по факту, выпуклые четырехугольники. В случае если пересечение каким-либо образом невыпуклое, то это решение работать не будет. Ниже пример, где мы рисуем область пересечения и заливаем ее цветом:

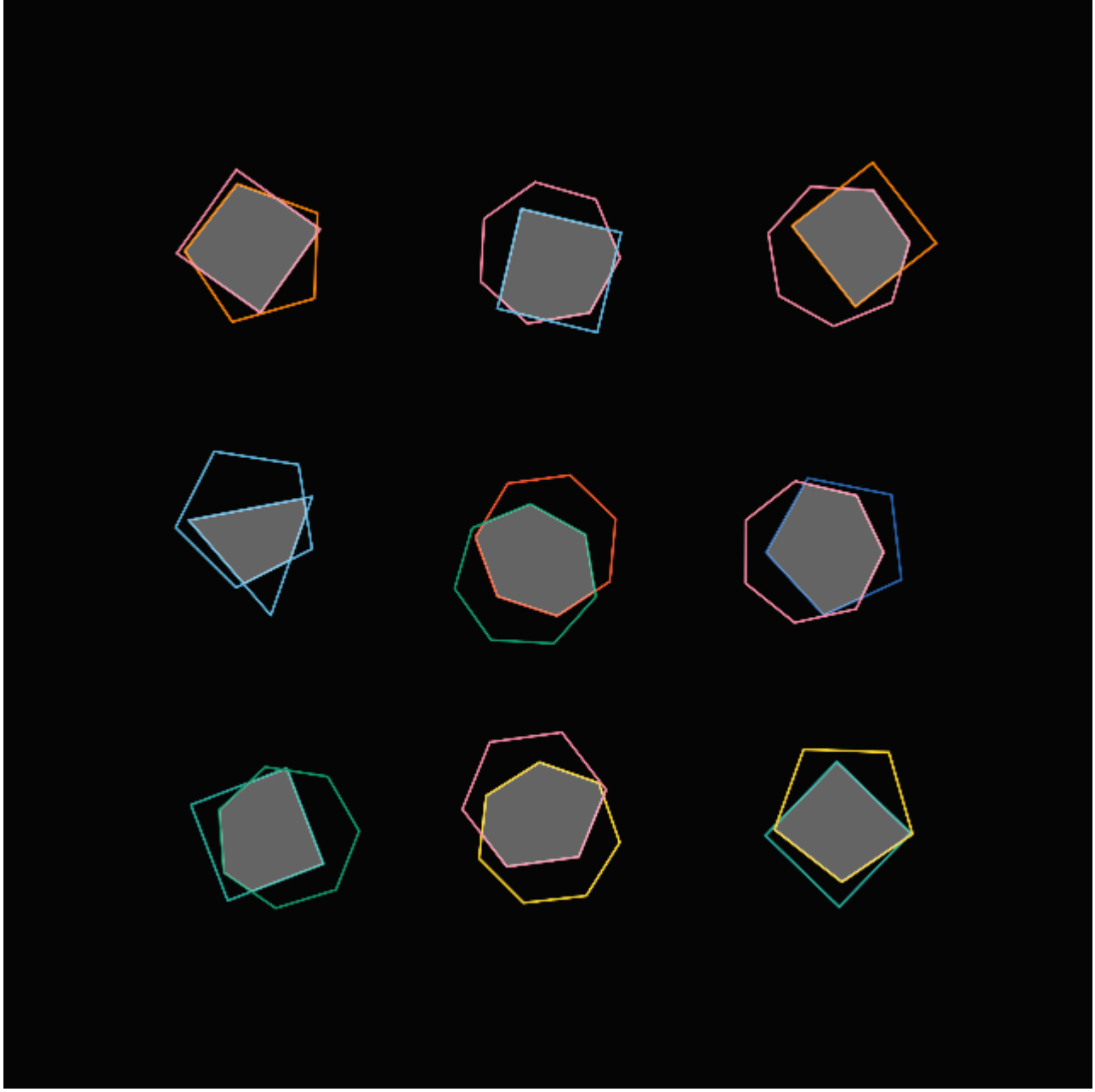


Заимствуйте этот код на openprocessing.

В следующем разделе мы посмотрим, как можно это расширить до случая выпуклых полигонов.

Отходим от прямоугольников: пересечение полигонов

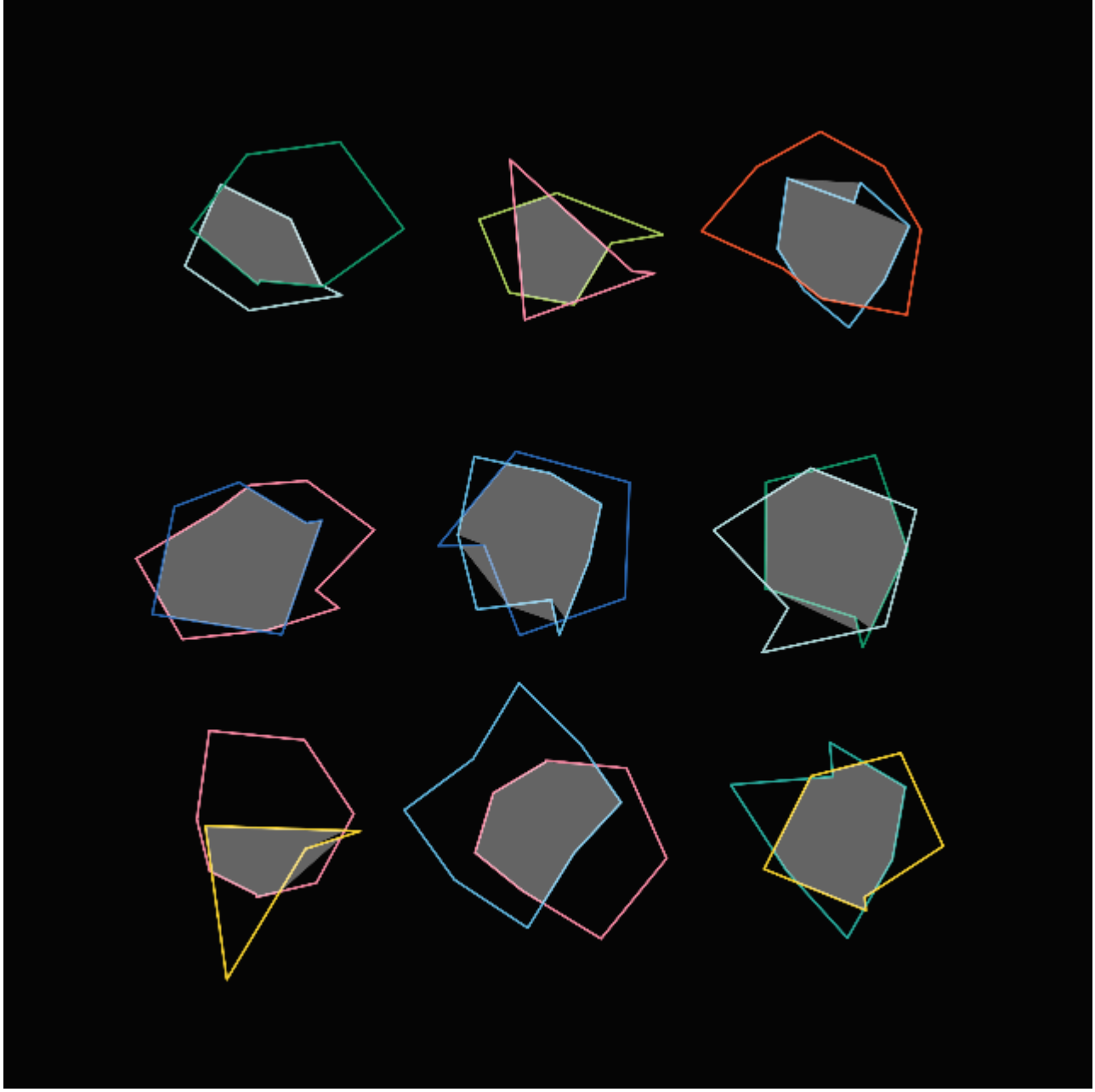
Все упомянутое до этого было сказано в контексте не выровненных по осям прямоугольников. Способ определения полигона пересечения, разработанный нами к этому моменту, однако, с помощью пары изменений может быть расширен до работы с выпуклыми полигонами! Изучите код следующего примера, чтобы посмотреть, как это работает. Нам просто нужно сделать изменения везде, где мы предполагали, что цикл пойдет через 4 ребра/точки:



Заимствуйте этот код на openprocessing.

Но по сути единственная разница состоит в том, что наш класс makeRect() становится классом makePoly(), где конструктор принимает список из вершин. Как вы создаете этот список вершин остается за вами, потому что нет единого метода создания разных типов полигонов. Пример выше демонстрирует пересечение различных N-угольников.

Технически это также может быть дополнено до невыпуклых полигонов, но в определенных случаях, когда точки не могут быть отсортированы по значению угла, это не сработает. Помните, когда мы сортировали наши вершины в порядке по часовой относительно центральной точки? В случае невыпуклых полигонов этот шаг больше не годится - порядок получается неоднозначен, - и в некоторых случаях это вытекает в некорректную отрисовку полигона пересечения. Возможно вам понадобится несколько раз обновить скетч, прежде чем вы получите фигуры, подобные таковым из следующего примера:



Заимствуйте этот код на openprocessing.

Вместо того, чтобы каким-то хитрым образом дорабатывать наш метод, возможно будет лучше использовать уже созданный метод, который не сталкивается с такой проблемой. Алгоритм Сазерленда-Ходжмана один из таких методов, и, на самом деле, довольно сильно походит на разработанный нами метод, но с различием в том, что он получает правильный порядок вершин во время построения вершин полигона пересечения. Здесь можно прочитать больше о том, почему Алгоритм Сазерленда-Ходжмана неплохо работает, и тут можно найти компактную javascript реализацию.

И напоследок я просто хочу оставить пару ссылок по теме для дальнейшего изучения. Помимо Сазерленда-Ходжмана, еще один известный алгоритм пересечения - это Weiler–Atherton clipping algorithm, хотя он является немного более сложным.

Другая смежная тема - это декомпозиция полигонов. Иногда необходимо разбить невыпуклый полигон на выпуклые части. Некоторые специальные походы являются алгоритмом Mark Bayazit'a. Простую для использования javascript реализацию алгоритма Mark Bayazit'a можно найти здесь, в дополнение к алгоритму ear clipping. Однако эти алгоритмы декомпозиции

полигонов требуют от вас расположения точек невыпуклого полигона в правильно порядке. Вы не можете просто передать кучу вершин и ожидать, что они сами определят правильный порядок. Еще вы также можете решить задачу декомпозиции полигона с помощью алгоритма триангуляции Делоне, такого как алгоритм Боуэра-Ватсона, но в итоге вы получите множество маленьких треугольников, что может оказаться неэффективным способом разделения фигуры.

Итак, вот мы и добрались до конца! Огромная благодарность за прочтение! Если вам понравилась эта статья, подумайте насчет того, чтобы поделиться ей с друзьями, загляните в мои другие посты или загляните поздороваться на Твиттер!

Теги: пересечение многоугольников, 2d, алгоритм, полигон, коллизии

Хабы: Алгоритмы, Математика

Комментарии 13

wataru 5 мая 2024 в 11:33

> проверяют, что у координата точки находится в диапазоне у координат ребра,

Тут все чуть более хитро. Там есть один крайний случай: если луч проходит через вершину многоугольника. Тут есть две разных ситуации: луч касается многоугольника в вершине, или пересекает границу в вершине. В первом случае надо подсчитать 2 или 0 пересечения, а во втором - 1.

Поэтому нельзя просто смотреть, что у координата точки лежит в интервале координат отрезка. Ведь, если концы считать за пересечение, то во всех случаях будет насчитано 2. А если не считать - то 0. А

надо как-то 1 получить при пересечении в вершине. Надо как-то по особому учитывать концы. Решение в лоб было бы отдельно смотреть на концы и соседние вершины, что бы понять, что они по разные стороны от луча. Более хитрый варинат - всегда считать только нижние концы каждого отрезка, что тут и происходит.

Условие `((yi > y) != (yj > y))` . При `y=yi` даст истину, только когда скобки дадут `false != true` , а значит `yj > yi` . Таким образом, пересечение подсчитается, только если `y = yi < yj,` т.е y - это нижняя граница отрезка.

+5 Ответить

ivankudryavtsev 5 мая 2024 в 11:51

Если что, на Rosetta Code есть реализации для чего угодно: https://rosettacode.org/wiki/Sutherland-Hodgman_polygon_clipping

Алгоритм Сазерленда-Ходжмана.

+2 Ответить

Goerging 5 мая 2024 в 13:15

Спасибо, поправил

0 Ответить

Nail_S 5 мая 2024 в 15:45

Статья понравилась, спасибо.

Я как раз тоже последние лет 5 увлекаюсь вычислительной геометрией.
И за последний год написал библиотеку для вычисления результатов пересечения многоугольников: xor, union, intersection and difference. Первое время пытаешься решать задачу (как и автор статьи) во float числах. Через какое-то время приходит осознание, что из-за потери точности задача в принципе не разрешима в общем случаи. С переходом на int появляется детерминизм и становится сильно проще. Например, параллельность отрезков можно проверять через векторное произведение и тд. С float такие фокусы не проходят. Введение epsilon не спасает, а только маскирует проблему.
С вашего позволения оставлю ссылку на свою имплементацию на
https://github.com/iShape-Rust/iOverlay
и
https://github.com/iShape-Swift/iOverlay

+1 Ответить

ivankudryavtsev 5 мая 2024 в 17:31

Интересно. Я на f32 ловил ошибки с геометрией, на f64 вроде все нормально. У Вас как работает?

0 Ответить

Nail_S 5 мая 2024 в 18:05

При использование f64 вероятность вырожденных случаев снижается, но не исключает их. Проблема в том, что вырожденные случаи если их правильно не обработать приводят к неправильной логике алгоритма, а это может повлечь за собой не только не правильный результат, но и зацикливание или краш.

0 Ответить

Graphist 6 мая 2024 в 02:08

Для работы с невыпуклыми полигонами мне помогало пронумеровать рёбра, и добавить номер ребра к параметру u в формуле для ребра p1 + u * (p2 - p1), т.е. задать в параметрическом виде весь полигон, а не одно ребро. Тогда u от 0 до 1 -- пересечение на первом ребре, от 1 до 2 -- на втором ребре и т.д., и можно собирать все точки (и исходные вершины, и пересечения) в один список (точнее, два списка -- для первого полигона и для второго), сортировать по u и, собирать в правильном порядке. Причём, при разном "правильном порядке" получался и OR, и AND, а если постараться -- то и A-B и B-A.
Но вот на ошибках вычисления с плавающей точкой я заломался, там даже f64 не хватало.

+2 Ответить

Zara6502 6 мая 2024 в 09:25

Спасибо за материал.

PS: слишком мало this в коде))) это что за язык где такая потребность?

0 Ответить

Alexandroppolus 7 мая 2024 в 01:08

А можно ли составить пересечение двух **выпуклых** полигонов за O(N+M), где N и M - количества вершин в оных?

Навскидку, если они оба представлены массивами вершин в порядке обхода по часовой стрелке, то напрашивается метод двух указателей: берем любую вершину одного, ближайшую к ней вершину второго, и далее обходим первый по часовой стрелке, двигая второй указатель вперед, только если для него следующая вершина ближе к текущей вершине первого. Так мы рано или поздно сможем поймать точку пересечение ребер (примыкающих к вершинам, на которых стоят указатели), а от неё примерно тем же способом собрать искомый контур.

0 Ответить

fujinon 7 мая 2024 в 01:46

Навскидку вроде кажется что число вершин пересечения не превышает N+M, так что такой алгоритм может и существует. Но ваш не работает потому что если ребро первого полигона пересекается двумя ребрами второго то вы проскочите точку (2 точки) пересечения.

0 Ответить

wataru 7 мая 2024 в 10:11

Можно. Ваша идея должна работать, только там всякие частные случаи надо обрабатывать.

Есть подход чуть проще - разбейте каждый полигон на 2 ломаные - верхнюю и нижнюю. Потом проходим сканирующей прямой слева-направо. В каждом промежутке между любыми x координатами вершин эти 4 ломаные - это прямые. Пересечем две нижние границы, две верхние, и верхнюю от одного с нижней от другого многоугольника. Получим максимум 4 точки пересечения, отсортируем по оси OX, потом на каждом маленьком интервале никаких пересечений уже нет и быть не может. Все границы - непересекающиеся отрезки. Осталось только взять максимум из нижних границ и минимум из верхних. Можно по средней точке их сравнить.
Потом надо будет удалить вершины на сторонах, ибо мы так можем добавить несколько коллинеарных отрезков подряд.

0 Ответить

Goerging 7 мая 2024 в 17:39

Вот тут, вроде как, утверждается, что алгоритм линейный и при том весьма простой. Но сам я пока статью не читал и не разбирался, ничего прокомментировать не могу.

0 Ответить

fujinon 7 мая 2024 в 01:50

Алгоритм сортировки вершин можно оптимизировать если сортировать не по углу а по тангенсу, который в пределах каждого квадранта монотонная функция. Сначала по знакам находим квадрант, и если оба угла в одном квадранте, сортирует по тангенсу (не надо вычислять atan)

+1 Ответить

Вы можете оставлять комментарии только к свежим публикациям

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

Erwinmal 13 часов назад

Чапаев и Матрица: почему культура 90-х бунтовала против пластмассового мира? Часть 1

Простой 9 мин 3.6K

Ретроспектива

+37 23 16 +16

 **Lunathecat** 9 часов назад

Электрогитара по доступной цене, не нуждающаяся в доработках

Простой 7 мин 4.2K

Обзор

+23 7 5 +5

 **DavidAsatryan** 10 часов назад

Agile умер: из-за своего сострадания к product- и project-менеджерам (с) Фридрих Ницше

Простой 8 мин 7.3K

Мнение

+22 50 8 +8

 **erbanovanastasia** 13 часов назад

Индия продолжает экспансию на рынок полупроводников: чего ожидать в ближайшем будущем

4 мин 1.6K

+22 3 12 +12

 **tw0face** 14 часов назад

Покажи свой стартап/пет-проект (Январь)

1 мин 1.5K

+22 17 30 +30

 **Giox_Nostr** 13 часов назад

Классика научной фантастики: хронология

Простой 22 мин 4.5K

Ретроспектива

+17 64 30 +30

 **Parfenovlgor** 12 часов назад

Опыт написания компилятора вручную

Средний 9 мин 3.2K

+16 40 8 +8

Lexx_Nimofff 13 часов назад

Игра, вдохновлённая UFO и Jagged Alliance: интервью с главным разработчиков «Спарты 2035»

Простой 9 мин 1K

Интервью

+13 3 2 +2

guselnikov 3 часа назад

То о чем многие молчат, или может не знают...

3 мин 1.9K

+11 9 9 +9

bodyawm 7 часов назад

Я купил легендарный игровой смартфон из утиля и отремонтировал его — смотрим на Nokia N-Gage Classic

10 мин 2.9K

Ретроспектива

+10 3 10 +10

Роботяги: что будет делать ИИ в промышленности к 2035 году

Турбо

Показать еще

МИНУТОЧКУ ВНИМАНИЯ

Новый год продолжается со скидками в Промокодусе

Иди со мной, если хочешь на перекур: будущее ИИ на заводах

Техноархеолог отправляется в затерянный город и спасает мир

ВАКАНСИИ

Art Lead (Game)
до 6 000 € · Wanted. · Лимассол

Motion Designer
от 1 200 до 1 500 $ · Peppermint · Минск · Можно удаленно

Frontend developer middle
от 4 100 до 4 100 $ · Pixel Point · Можно удаленно

Project manager
от 1 500 до 2 000 $ · Peppermint · Можно удаленно

Senior FrontEnd Engineer at Pixel Point
до 6 250 $ · Pixel Point · Можно удаленно

Больше вакансий на Хабр Карьере

ЧИТАЮТ СЕЙЧАС

YouTube начал показывать пользователям с блокировщиками рекламы многочасовую рекламу, которую нельзя пропустить

26K 102 +102

Куда деваются отходы в самолетных туалетах

83K 127 +127

Преподавание английского — самый большой скам 21 века

45K 116 +116

Agile умер: из-за своего сострадания к product- и project-менеджерам (с) Фридрих Ницше

7.3K 8 +8

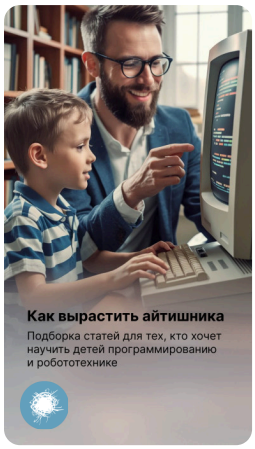
Веб-приложения будущего: что нужно знать о WebAssembly

9.1K 19 +19

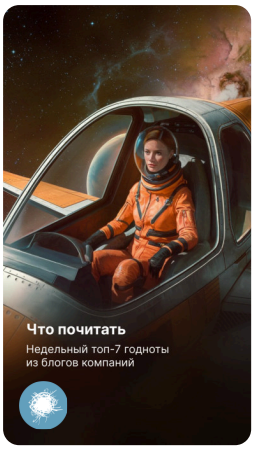
Роботяги: что будет делать ИИ в промышленности к 2035 году

Турбо

ИСТОРИИ



Как вырастить айтишника



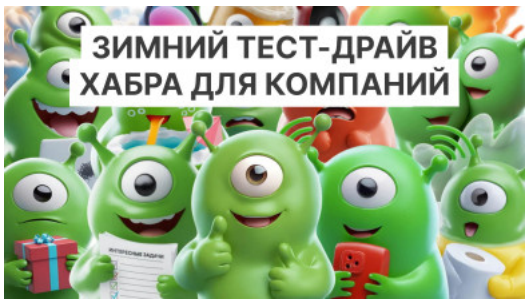
Годнота из блогов компаний

Нейрозима 2025

Статьи с новогодним вайбом

Кто выступит на конференции мечты

БЛИЖАЙШИЕ СОБЫТИЯ



30 января

Зимний тест-драйв Хабра для компаний

Москва

Маркетинг Другое

27 марта

Deckhouse Conf 2025

Москва

Разработка

Администрирование

Менеджмент

25 – 26 апреля

IT-конференция M Tatarstan 2025

Казань

Разработка Маркетин

Другое

Больше событий в календаре