Université Paris 8

Moteurs de Jeu [1]

Nicolas JOUANDEAU
n@up8.edu

septembre 2022

1. A la découverte des fonctions proposées par les moteurs et les jeux possibles.

4 Détection de collision

Les principales fonctions de la détection de collision sont :
- la détection de collision propre, à savoir si il y a collision entre les objets considérés ;
- le calcul de la collision, à savoir les points d'intersection des côtés des objets ;
- la profondeur de collision, à savoir l'intensité de la collision entre deux objets et les pénétrations respectives des objets entre eux.

La détection de collision est un problème avant tout géométrique, et peut se résumer pour la collision propre, à l'appartenance d'un point à une forme ; en 2D, les formes sont constuites à partir de triangles, de rectangles ou de cercles ; l'approche classique de simplification de la détection de collision est d'englober les objets avec des formes englobantes permettant des simplifications géométriques ; les collisions avec les objets sont réduits aux collisions avec les formes englobantes choisies ; quelles que soient les formes englobantes choisies, on recalcule à chaque déplacement d'un objet sa forme englobante.

Les simplifications courantes sont d'englober les objets par :
- un rectangle aligné avec les axes ; nommée *Axis Aligned Bounding Boxes* (AABB), cette représentation des objets permet de réduire les collisions entre objets à des tests de collision entre rectangle ayant des cotés verticaux et horizontaux ;
- un rectangle orienté selon l'axe principal de l'objet ; nommée *Oriented Bounding Boxes* (OBB), cette représentation des objets permet de réduire les collisions entre objets à des tests de collision entre rectangles quelconques ;
- un cercle englobant ; cette représentation des objets permet de réduire les collisions entre objets à des tests de collision entre cercles ;
- une ellipse englobante ; cette représentation des objets permet de réduire les collisions entre objets à des tests de collision entre ellipses ;
- l'enveloppe convexe de chaque objet ;
- plusieurs volumes englobants (AABB, OBB, cercles, ellipses, enveloppes convexes) ; cette représentation des objets implique d'avoir une liste de 1 à n volumes pour chaque objet ; la détection de collision entre deux objets devient un test de collision de $n \times m$ volumes.

4.1 Collision AABB

Pour un jeu tel que Super Mario Bros présenté en Fig. 19 à gauche, il sera intuitif d'utiliser des rectangles AABB pour les collisions entre personnages et pour les collisions des personnages avec des éléments de décor.

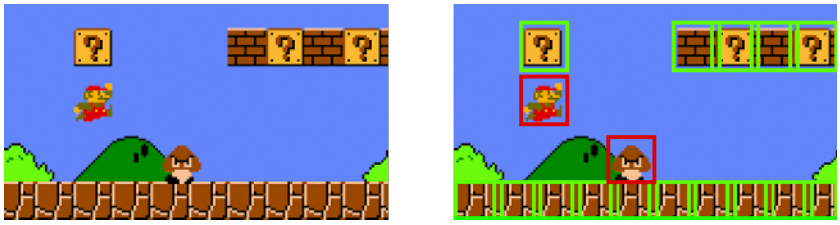


FIG. 19 – Super Mario Bros.

La partie droite de Fig 19 présente les rectangles associés à la détection de collision AABB dans le jeu Super Mario Bros ; les rectangles rouges correspondent aux personnages qui se déplacent et les rectangles verts aux éléments de décor ; il convient de vérifier la non-collision de chacun des rectangles rouges avec tous les autres rectangles (rouges et verts).

Le fichier `Rect.js` présenté ci-dessous permet de créer des objets rectangulaire, de tester la collision AABB entre deux objets et de déplacer ces objets ; un rectangle est identifié par quatre coordonnées `xi`, `yi`, `xf` et `yf` ; selon les positions relatives de ces coordonnées pour un objet vis à vis d'un autre objet nommé `otherRect`, la fonction `AABBcollide` détecte les collisions entre ces deux objets.

```
export default class Rect {
  constructor(xi, yi, xf, yf) {
    this.xi = xi; this.yi = yi;
    this.xf = xf; this.yf = yf;
  }
  AABBcollide(otherRect) {
    if(this.xi > otherRect.xf) return false;
    if(this.xf < otherRect.xi) return false;
    if(this.yi > otherRect.yf) return false;
    if(this.yf < otherRect.yi) return false;
    return true;
  }
  move(dx, dy) {
    this.xi += dx; this.xf += dx;
    this.yi += dy; this.yf += dy;
  }
}
```

Pour illustrer la détection de collision AABB, on réalise une animation presentée en Fig. 20 et composée de rectangles fixes et de rectangles mobiles ; toes les rectangles sont placés alétoirement ; les rectangles fixes sont gris ; les rectangles mobiles se déplacent et changent en fonction des collisions avec les rectangles fixes ; les rectangles mobiles sont rouges lorsqu'ils sont sans collision et sont transparents lors qu'ils sont en collision avec un rectangle fixe.

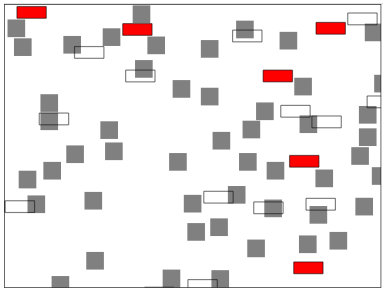


FIG. 20 – Détection de collision AABB.

`static_R` et `moving_R` contiennent respectivement la liste des rectangles fixes et celle des rectangles mobiles ; les rectangles fixes sont sans collision entre eux ; les rectangles mobiles sont également sans collision entre eux.

```
1  import Rect from "./Rect.js";
2  let cnv = document.getElementById("myCanvas");
3  let ctx = cnv.getContext("2d");
4  let velocity = 2;
5  let static_R = [];
6  let moving_R = [];
7  function collide_with(new_r, T) {
8    for (let i = 0; i < T.length; i++)
9      if(new_r.collide(T[i])) return true;
10    return false;
11 }
12 while (static_R.length < 50) {
13   let new_x = Math.floor(Math.random() * cnv.width);
14   let new_y = Math.floor(Math.random() * cnv.height);
15   let new_r = new Rect(new_x, new_y,new_x+30,new_y+30);
16   if(collide_with(new_r, static_R) == false)
17     static_R.push(new_r);
18 }
19 while (moving_R.length < 20) {
20   let new_x = Math.floor(Math.random() * cnv.width);
21   let new_y = Math.floor(Math.random() * cnv.height);
22   let new_r = new Rect(new_x, new_y,new_x+50,new_y+20);
23   if(collide_with(new_r, moving_R) == false)
24     moving_R.push(new_r);
25 }
```

La fonction `draw` est appelée à chaque MAJ de la page du navigateur ; après effacement de l'ensemble du canvas (ligne 32), les rectangles fixes sont dessinés (lignes 33 à 40) ; les rectangles mobiles sont dessinés par dessus, avec un remplissage rouge si sans collision (lignes 42 à 49) et avec un bord noir dans tous les cas (lignes 50 à 55).

La fonction `update_pos` déplace les rectangles mobiles de `velocity` pixels vers le bas ; quand un rectangle est en bas du canvas, il est replacé en haut du canvas.

```
31 function draw() {
32   ctx.clearRect(0, 0, cnv.width, cnv.height);
33   for (let i = 0; i < static_R.length; i++) {
34     ctx.beginPath();
35     ctx.rect(static_R[i].xi, static_R[i].yi,
36              static_R[i].xf-static_R[i].xi, static_R[i].yf-static_R[i].yi);
37     ctx.fillStyle = "gray";
38     ctx.fill();
39     ctx.closePath();
40   }
41   for (let i = 0; i < moving_R.length; i++) {
42     if(collide_with(moving_R[i], static_R) == false) {
43       ctx.beginPath();
44       ctx.rect(moving_R[i].xi, moving_R[i].yi,
45                moving_R[i].xf-moving_R[i].xi, moving_R[i].yf-moving_R[i].yi);
46       ctx.fillStyle = "red";
47       ctx.fill();
48       ctx.closePath();
49     }
50     ctx.beginPath();
51     ctx.rect(moving_R[i].xi, moving_R[i].yi,
52              moving_R[i].xf-moving_R[i].xi, moving_R[i].yf-moving_R[i].yi);
53     ctx.strokeStyle = "black";
54     ctx.stroke();
55     ctx.closePath();
56   }
57 }
58 function update_pos() {
59   for (let i = 0; i < moving_R.length; i++) {
60     moving_R[i].move(0, velocity);
61     if(moving_R[i].yi > cnv.height)
62       moving_R[i].move(0, -cnv.height-20);
63   }
64 }
65 function update(timestamp) {
66   update_pos();
67   draw();
68   requestAnimationFrame(update);
69 }
70 requestAnimationFrame(update);
```

4.2 Collision entre cercles

Pour un jeu tel que Bubble Bobble présenté en Fig 21, il sera intuitif d'utiliser des cercles pour les collisions entre les bulles de savon (les cercles associés sont présentés en bleu dans la partie droite du jeu).

FIG. 21 – Bubble Bobble.

Le fichier `Arc.js` présenté ci-dessous permet de créer des objets circulaires, de tester la collision entre deux objets circulaires et de déplacer ces objets ; un cercle est identifié par deux coordonnées `x`, `y` et par un rayon `r` ; selon la position et le rayon, la fonction `collide` détecte les collisions entre ces deux objets.

```
function dist(x0, y0, x1, y1) {
  let dx = x1-x0;
  let dy = y1-y0;
  return Math.sqrt( dx*dx + dy*dy );
}

export default class Arc {
  constructor(x, y, r) {
    this.x = x; this.y = y; this.r = r;
  }
  collide(otherArc) {
    let r = this.r + otherArc.r;
    return dist(this.x, this.y, otherArc.x, otherArc.y) < r;
  }
  move(dx, dy) {
    this.x += dx; this.y += dy;
  }
  moveTo(nx, ny) {
    this.x = nx; this.y = ny;
  }
}
```

Pour illustrer la détection de collision entre cercles, on réalise une animation presentée en Fig. 22 et composée de cercles fixes et de cercles mobiles ; tous les cercles sont placés alétoirement ; les cercles fixes sont gris ; les cercles mobiles se déplacent et changent en fonction des collisions avec les cercles fixes ; les cercles mobiles sont rouges lorsqu'ils sont sans collision et sont transparents lors qu'ils sont en collision avec un cercle fixe.

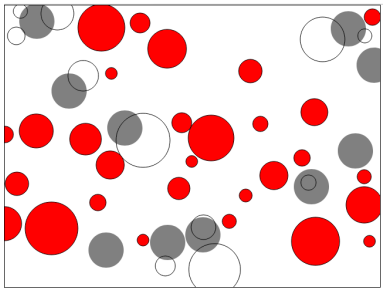


FIG. 22 – Détection de collision de cercles.

`static_A` et `moving_A` contiennent respectivement la liste des cercles fixes et celle des cercles mobiles ; les cercles fixes sont sans collision entre eux ; les cercles mobiles sont également sans collision entre eux.

```
import Arc from "./Arc.js";
let cnv = document.getElementById("myCanvas");
let ctx = cnv.getContext("2d");
let velocity = 2;
let moving_A = [];
let static_A = [];
function collide_with(new_a, T) {
  for (let i = 0; i < T.length; i++)
    if(new_a.collide(T[i])) return true;
  return false;
}
while (static_A.length < 10) {
  let new_x = Math.floor(Math.random() * cnv.width);
  let new_y = Math.floor(Math.random() * cnv.height);
  let new_a = new Arc(new_x, new_y, 30);
  if(collide_with(new_a, static_A) == false)
    static_A.push(new_a);
}
while (moving_A.length < 40) {
  let new_x = Math.floor(Math.random() * cnv.width);
  let new_y = Math.floor(Math.random() * (cnv.height-50));
  let new_r = 10+Math.floor(Math.random() * 40);
  let new_a = new Arc(new_x, new_y, new_r);
  if(collide_with(new_a, moving_A) == false)
    moving_A.push(new_a);
}
```

La fonction `draw` est appelée à chaque MAJ de la page du navigateur ; après effacement de l'ensemble du canvas (ligne 28), les cercles fixes sont dessinés (lignes 29 à 35) ; les cercles mobiles sont dessinés par dessus, avec un remplissage rouge si sans collision (lignes 37 à 43) et avec un bord noir dans tous les cas (lignes 44 à 48).

La fonction `update_pos` déplace les cercles mobiles de `velocity` pixels vers le bas ; quand un cercle est en bas du canvas, il est replacé en haut du canvas.

```
27 function draw() {
28   ctx.clearRect(0, 0, cnv.width, cnv.height);
29   for (let i = 0; i < static_A.length; i++) {
30     ctx.beginPath();
31     ctx.arc(static_A[i].x, static_A[i].y, static_A[i].r, 0, 2*Math.PI);
32     ctx.fillStyle = "gray";
33     ctx.fill();
34     ctx.closePath();
35   }
36   for (let i = 0; i < moving_A.length; i++) {
37     if(collide_with(moving_A[i], static_A) == false) {
38       ctx.beginPath();
39       ctx.arc(moving_A[i].x, moving_A[i].y, moving_A[i].r, 0, 2*Math.PI);
40       ctx.fillStyle = "red";
41       ctx.fill();
42       ctx.closePath();
43     }
44     ctx.beginPath();
45     ctx.arc(moving_A[i].x, moving_A[i].y, moving_A[i].r, 0, 2*Math.PI);
46     ctx.strokeStyle = "dark";
47     ctx.stroke();
48     ctx.closePath();
49   }
50 }
51 function update_pos() {
52   for (let i = 0; i < moving_A.length; i++) {
53     moving_A[i].move(0,velocity);
54     if(moving_A[i].y >= cnv.height) {
55       moving_A[i].moveTo(moving_A[i].x, 0);
56     }
57   }
58 }
59 function update(timestamp) {
60   update_pos();
61   draw();
62   requestAnimationFrame(update);
63 }
64 requestAnimationFrame(update);
```

4.3 Orientation

Sachant qu'il est parfois plus simple de systématiser la résolution des problèmes de géométrie en ordonnant les points, nous allons considérer la notion d'orientation ; l'orientation d'un angle est le sens dans lequel on tourne en suivant les deux segments formant cet angle ; si les deux segments sont parallèles, ils n'ont pas d'orientation particulière ; alternativement, ils sont soit dans le sens horaire, soit dans le sens anti-horaire ; le sens horaire correspond au sens de rotation des aiguilles d'une montre (*respect.* anti-horaire pour l'inverse des aiguilles).

On pourra définir une fonction `cw` :

- qui prend trois points `a`, `b`, `c` en paramètres ;
- qui retourne une valeur positive si `c` est à droite de $[ab]$ (*i.e.* $\widehat{\text{bac}}$ est de sens anti-horaire) ;
- qui retourne une valeur négative si `c` est à gauche de $[ab]$ (*i.e.* $\widehat{\text{bac}}$ est de sens horaire) ;
- qui retourne 0 si les segments $[ab]$ et $[ac]$ sont parallèles ;
- avec $\texttt{cw}(a, b, c) = (x_b - x_a)(y_c - y_a) - (y_b - y_a)(x_c - x_a)$; c'est le calcul du déterminant des vecteurs associés aux segments $[ab]$ et $[ac]$.

```
1 class Pt {
2   constructor(x, y, dx, dy) {
3     this.x = x; this.y = y; this.dx = dx; this.dy = dy;
4   }
5 }
6 function cw(a,b,c) { return (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x); }
```

Pour illustrer l'utilisation de `cw`, on réalise une animation presentée en Fig. 23 et composée de rectangles mobiles placés aléatoirement avec un déplacement également aléatoire ; on a un segment $[ab]$ de référence et les centres des rectangles définissent des points `c` ; si le point `c` est à droite du segment $[ab]$, le rectangle est dessiné en bleu ; si le point `c` est à gauche, le rectangle est dessiné en rouge ; si le point `c` est aligné avec le segment, le rectangle est dessiné en vert ; pour identifier le début du segment $[ab]$, on dessine un cercle en `a`.

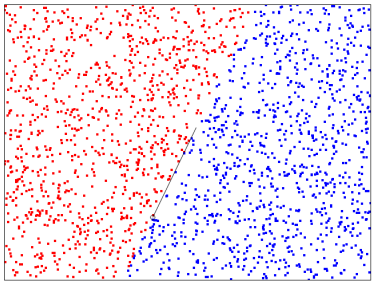


FIG. 23 – Position d'un point par rapport à un segment.

```
let cnv = document.getElementById("myCanvas");
let ctx = cnv.getContext("2d");
let pts = [];
for (let i = 0; i < 2000; i++) {
  let new_x = Math.floor(Math.random() * (cnv.width-2));
  let new_y = Math.floor(Math.random() * (cnv.height-2));
  let new_dx = Math.floor(Math.random()*4)-2;
  let new_dy = Math.floor(Math.random()*4)-2;
  pts.push(new Pt(new_x, new_y, new_dx, new_dy));
}
let A_x = 200 + Math.floor(Math.random() * 200);
let A_y = 200 + Math.floor(Math.random() * 200);
let ptA = new Pt(A_x, A_y, 0, 0);
let B_x = 200 + Math.floor(Math.random() * 200);
let B_y = 200 + Math.floor(Math.random() * 200);
let ptB = new Pt(B_x, B_y, 0, 0);
function draw() {
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  for (let i = 0; i < pts.length; i++) {
    ctx.beginPath();
    ctx.rect(pts[i].x-2, pts[i].y-2, 4, 4);
    if(cw(ptA,ptB,pts[i]) > 0) ctx.fillStyle = "blue";
    else if(cw(ptA,ptB,pts[i]) < 0) ctx.fillStyle = "red";
    else ctx.fillStyle = "green";
    ctx.fill();
    ctx.closePath();
  }
  ctx.beginPath();
  ctx.arc(ptA.x, ptA.y, 4, 0, 2*Math.PI);
  ctx.moveTo(A_x,A_y);
  ctx.lineTo(B_x,B_y);
  ctx.strokeStyle = "black";
  ctx.stroke();
  ctx.closePath();
}
function update_pos() {
  for (let i = 0; i < pts.length; i++) {
    pts[i].x += pts[i].dx;
    pts[i].y += pts[i].dy;
    if(pts[i].x > cnv.width) pts[i].x -= cnv.width;
    if(pts[i].x < 0) pts[i].x += cnv.width;
    if(pts[i].y > cnv.height) pts[i].y -= cnv.height;
    if(pts[i].y < 0) pts[i].y += cnv.height;
  }
}
function update(timestamp) {
  update_pos();
  draw();
  requestAnimationFrame(update);
}
requestAnimationFrame(update);
```

4.4 Intersection entre deux segments

Concernant l'intersection de deux segments, on commence par utiliser la fonction `cw` pour nous assurer de l'existance de l'intersection :

- si les points du deuxième segment sont tous les deux à droite ou à gauche du premier segment, il n'y a pas intersection ;
- si les points du premier segment sont tous les deux à droite ou à gauche du deuxième segment, il n'y a pas intersection ;
- si les deux segments sont colinéaires, il n'y a pas intersection.

Sachant qu'il y a intersection et sachant que les pentes sont différentes, l'intersection des deux segments considérés est équivalente à l'intersection des droites associées ; pour deux segments distincts `[ab]` et `[cd]`, on a une droite $y = ax + b$ pour `[ab]` et une droite $y = a'x + b'$ pour `[cd]` ; leur intersection (x_i, y_i) est définie par :

$$a = (y_b - y_a)/(x_b - x_a) \quad \text{et} \quad b = y_a - ax_a$$

$$a' = (y_d - y_c)/(x_d - x_c) \quad \text{et} \quad b' = y_c - ax_c$$

$$x_i = (b' - b)/(a - a')$$

$$y_i = ax_i + b$$

Ce qui implique de vérifier $x_a \neq x_b$, $x_c \neq x_d$ et $a \neq a'$.

Sachant les segments `[ab]` et `[cd]` non-colinéaire, on a $a \neq a'$.

Si $x_a = x_b$, on a $x_i = x_a$ et donc $y_i = a'x_a + b'$ sans calculer a et b.

Si $x_c = x_d$, on a $x_i = x_c$ et donc $y_i = ax_c + b$ sans calculer a' et b'.

Pour illustrer ces calculs d'intersection, pour un ensemble de segments tirés aléatoirement présenté en Fig. 24, on dessine en rouge pointillé les segments sans intersection avec un segment de référence (dessiné en gras) et un carré au point d'intersection avec les autres segments si intersection il y a.

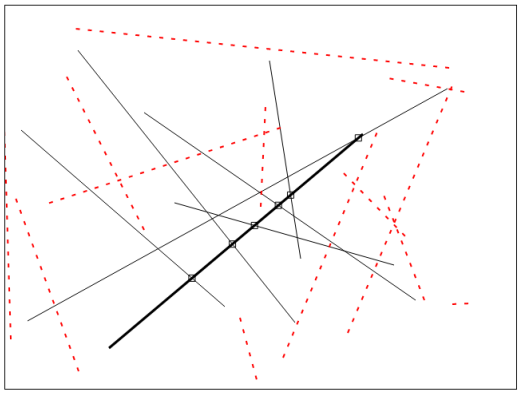


FIG. 24 – Intersection entre segments.

La fonction `intersectionTest` teste l'existance d'une intersection entre deux segments.

La fonction `intersectionPt` retourne le point d'intersection entre deux segments séquents.

```
class Sgt {
  constructor(a,b) {
    this.a = new Pt(a.x,a.y);
    this.b = new Pt(b.x,b.y);
  }
  left(otherSgt) {
    return (cw(this.a, this.b, otherSgt.a) < 0 &&
            cw(this.a, this.b, otherSgt.b) < 0);
  }
  right(otherSgt) {
    return (cw(this.a, this.b, otherSgt.a) > 0 &&
            cw(this.a, this.b, otherSgt.b) > 0);
  }
  intersectionTest(otherSgt) {
    if(this.left(otherSgt) || this.right(otherSgt)) return false;
    if(otherSgt.left(this) || otherSgt.right(this)) return false;
    if(cw(this.a, this.b, otherSgt.a) == 0 &&
       cw(this.a, this.b, otherSgt.b) == 0 &&
       cw(otherSgt.a, otherSgt.b, this.a) == 0 &&
       cw(otherSgt.a, otherSgt.b, this.b) == 0) return false;
    return true;
  }
  intersectionPt(otherSgt) {
    if(this.a.x == this.b.x) {
      let nap = (otherSgt.b.y - otherSgt.a.y) / (otherSgt.b.x - otherSgt.a.x);
      let nbp = otherSgt.a.y - nap * otherSgt.a.x;
      return new Pt(this.a.x, nap * this.a.x+nbp);
    }
    if(otherSgt.a.x == otherSgt.b.x) {
      let na = (this.b.y - this.a.y) / (this.b.x - this.a.x);
      let nb = this.a.y - na * this.a.x;
      return new Pt(otherSgt.a.x, na * otherSgt.a.x+nb);
    }
    let na = (this.b.y - this.a.y) / (this.b.x - this.a.x);
    let nb = this.a.y - na * this.a.x;
    let nap = (otherSgt.b.y - otherSgt.a.y) / (otherSgt.b.x - otherSgt.a.x);
    let nbp = otherSgt.a.y - nap * otherSgt.a.x;
    let xinters = (nbp-nb)/(na-nap);
    return new Pt(xinters, na * xinters+nb);
  }
}
```

Le premier segment du tableau `sgts` est le segment de référence ; on regarde l'intersection du premier segments avec chacun des autres segments (ligne 19) ; les lignes 15 à 30 ne font que varier la mise en forme du dessin du ième segment ; si il y a intersection, on ajoute un rectangle au point d'intersection (ligne 28).

```
1  let cnv = document.getElementById("myCanvas");
2  let ctx = cnv.getContext("2d");
3  let sgts = [];
4  for (let i = 0; i < 20; i++) {
5    let new_xi = 2+Math.floor(Math.random() * cnv.width-4);
6    let new_yi = 2+Math.floor(Math.random() * cnv.height-4);
7    let new_xf = 2+Math.floor(Math.random() * cnv.width-4);
8    let new_yf = 2+Math.floor(Math.random() * cnv.height-4);
9    sgts.push(new Sgt(new Pt(new_xi, new_yi), new Pt(new_xf, new_yf)));
10 }
11 function draw() {
12   ctx.clearRect(0, 0, cnv.width, cnv.height);
13   for (let i = 0; i < sgts.length; i++) {
14     ctx.beginPath();
15     if(i == 0) {
16       ctx.lineWidth = 3;
17       ctx.strokeStyle = "black";
18     } else {
19       if(sgts[0].intersectionTest(sgts[i]) == false) {
20         ctx.lineWidth = 2;
21         ctx.setLineDash([5,10]);
22         ctx.strokeStyle = "red";
23       } else {
24         ctx.lineWidth = 1;
25         ctx.setLineDash([]);
26         ctx.strokeStyle = "black";
27         let inters = sgts[0].intersectionPt(sgts[i]);
28         ctx.rect(inters.x-4,inters.y-4, 8,8);
29       }
30     }
31     ctx.moveTo(sgts[i].a.x,sgts[i].a.y);
32     ctx.lineTo(sgts[i].b.x,sgts[i].b.y);
33     ctx.stroke();
34     ctx.closePath();
35   }
36 }
37 draw();
```

4.5 Intersection entre segment et cercle

L'intersection entre un segment `[a,b]` et un cercle $\mathcal{C}$ de centre `o` et de rayon r équivaut à résoudre le système suivant :

$$\begin{cases} x = x_a + t(x_b - x_a) & \text{avec } t \in [0,1] \\ y = y_a + t(y_b - y_a) & \text{avec } t \in [0,1] \\ (x - x_c)^2 + (y - y_c)^2 - r^2 = 0 \end{cases}$$

Ce qui revient à résoudre $\alpha t^2 + \beta t + \gamma = 0$ avec :

$$\alpha = (x_b - x_a)^2 + (y_b - y_a)$$

$$\beta = 2((x_b - x_a)(x_a - x_c) + (y_b - y_a)(y_a - y_c))$$

$$\gamma = (x_a - x_c)^2 + (y_a - y_c)^2 - r^2$$

$$\text{Pour } \Delta = \beta^2 - 4\alpha\gamma \geq 0 \text{ , on a } c_1 = \frac{-\beta + \sqrt{\delta}}{2\alpha} \text{ et } c_2 = \frac{-\beta - \sqrt{\delta}}{2\alpha} \text{ .}$$

Pour $0 \leq c_1 \leq 1$, on a une intersection $(x_a + c_1(x_b - x_a), y_a + c_1(y_b - y_a))$.
Pour $0 \leq c_2 \leq 1$, on a une intersection $(x_a + c_2(x_b - x_a), y_a + c_2(y_b - y_a))$.

Pour un cercle (instance de `Circ` de centre `c` et de rayon `r`) et un segment `sgt`, la fonction `intersectionSgt` retourne le nombre d'intersections suivi de leurs coordonnées.

```
1  class Circ {
2    constructor(c, r) {
3      this.c = new Pt(c.x, c.y);
4      this.r = r;
5    }
6    intersectionSgt(sgt) {
7      let dx = sgt.b.x - sgt.a.x;
8      let dy = sgt.b.y - sgt.a.y;
9      let A = dx*dx + dy*dy;
10     let dx0c = sgt.a.x - this.c.x;
11     let dy0c = sgt.a.y - this.c.y;
12     let B = 2*(dx*dx0c+dy*dy0c);
13     let C = dx0c*dx0c+dy0c*dy0c-this.r*this.r;
14     let delta = B*B - 4*A*C;
15     if(delta >= 0) {
16       let c1 = (((-1)*B)+Math.sqrt(delta))/(2*A);
17       let c2 = (((-1)*B)-Math.sqrt(delta))/(2*A);
18       let inters1 = new Pt(sgt.a.x+c1*dx, sgt.a.y+c1*dy);
19       let inters2 = new Pt(sgt.a.x+c2*dx, sgt.a.y+c2*dy);
20       if((c1 <= 1.0 && c1 >= 0.0) && (c2 <= 1.0 && c2 >= 0.0))
21         return [2, inters1, inters2];
22       else if(c1 <= 1.0 && c1 >= 0.0) return [1, inters1];
23       else if(c2 <= 1.0 && c2 >= 0.0) return [1, inters2];
24     }
25     return [0];
26   }
27 }
```

Pour illustrer ces calculs d'intersection, pour un cercle et un ensemble de segments tirés aléatoirement présenté en Fig. 25, on dessine en rouge pointillé les segments sans intersection avec le cercle ; pour les segments en intersection avec le cercle, on dessine un carré à chaque point d'intersection.

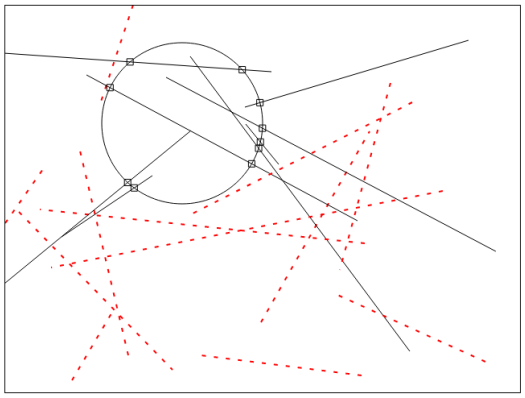


FIG. 25 – Intersection entre segment et cercle.

Les segments tirés aléatoirement sont placés dans un tableau `sgts` (ligne 9) ; le cercle est tiré aléatoirement dans le centre de la fenêtre (lignes 11 et 12) et possède un rayon de taille 100 (ligne 13) ; on appelle la fonction `draw` pour afficher segments et cercle (ligne 14).

```
1  let cnv = document.getElementById("myCanvas");
2  let ctx = cnv.getContext("2d");
3  let sgts = [];
4  for (let i = 0; i < 20; i++) {
5    let new_xi = 2+Math.floor(Math.random() * cnv.width-4);
6    let new_yi = 2+Math.floor(Math.random() * cnv.height-4);
7    let new_xf = 2+Math.floor(Math.random() * cnv.width-4);
8    let new_yf = 2+Math.floor(Math.random() * cnv.height-4);
9    sgts.push(new Sgt(new Pt(new_xi, new_yi), new Pt(new_xf, new_yf)));
10 }
11 let new_xc = 100+Math.floor(Math.random() * 200);
12 let new_yc = 100+Math.floor(Math.random() * 200);
13 let new_c = new Circ(new Pt(new_xc, new_yc), 100);
14 draw();
```

La fonction `draw` dessine segments et cercle ; la mise en forme du ième segment varie selon le retour de la fonction `intersectionSgt` de la classe `Circ` présentée page 56 ; les segments sont sans intersection sont dessinés en rouge pointillé (lignes 12 à 14) ; les intersections sont représentées par un rectangle (ligne 20 pour une intersection et lignes 22 à 23 pour deux intersections).

```
14 function draw() {
15   ctx.clearRect(0, 0, cnv.width, cnv.height);
16   ctx.beginPath();
17   ctx.arc(new_c.c.x, new_c.c.y, new_c.r, 0, 2*Math.PI);
18   ctx.strokeStyle = "dark";
19   ctx.stroke();
20   ctx.closePath();
21   for (let i = 0; i < sgts.length; i++) {
22     ctx.beginPath();
23     let res = new_c.intersectionSgt(sgts[i]);
24     if(res[0] == 0) {
25       ctx.lineWidth = 2;
26       ctx.setLineDash([5,10]);
27       ctx.strokeStyle = "red";
28     } else {
29       ctx.lineWidth = 1;
30       ctx.setLineDash([]);
31       ctx.strokeStyle = "black";
32       if(res[0] == 1) {
33         ctx.rect(res[1].x-4,res[1].y-4, 8,8);
34       } else { // res[0] == 2
35         ctx.rect(res[1].x-4,res[1].y-4, 8,8);
36         ctx.rect(res[2].x-4,res[2].y-4, 8,8);
37       }
38     }
39     ctx.moveTo(sgts[i].a.x,sgts[i].a.y);
40     ctx.lineTo(sgts[i].b.x,sgts[i].b.y);
41     ctx.stroke();
42     ctx.closePath();
43   }
44 }
```

4.6 Polygone convexe

A partir de points tirés aléatoirement, on peut créer des polygones convexes ; Fig. 26 présente un ensemble de points (représentés par des rectangles) dont l'enveloppe convexe est représentée par des traits noirs.

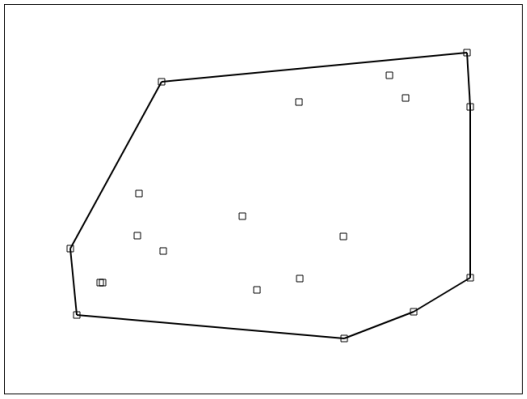
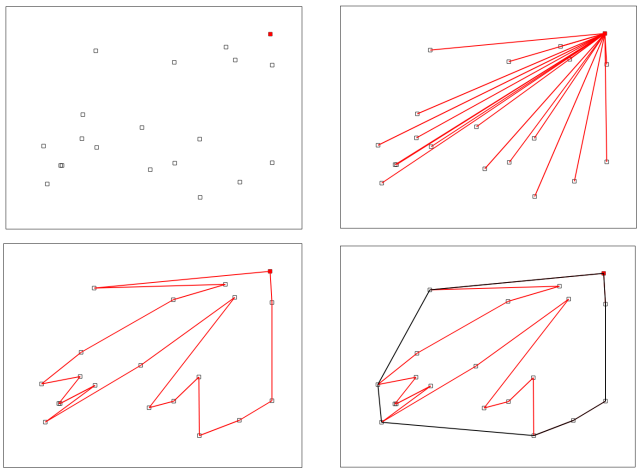


FIG. 26 – Enveloppe convexe d'un ensemble de points.

Présenté en Tab. 2, l'algorithme de Graham permet de calculer l'enveloppe convexe $\mathcal{E}$ d'un ensemble de points $\mathcal{P}$ comme suit :

- on définit le point de valeur min en `y` comme pivot (pour des `y` égaux, on prend le min en `x`) ;
- on ordonne les points de $\mathcal{P}$ selon l'angle qu'ils forment avec le pivot ;
- pour $i \in [1, n-2]$, le point $\mathcal{P}[i] \in \mathcal{E}$ ssi $\texttt{cw}(\mathcal{P}[i-1], \mathcal{P}[i], \mathcal{P}[i+1]) > 0$;
- pour $i = n-1$, le point $\mathcal{P}[n-1] \in \mathcal{E}$ ssi $\texttt{cw}(\mathcal{P}[i-1], \mathcal{P}[i], \mathcal{P}[0]) > 0$.



TAB. 2 – Etapes de construction de l'enveloppe convexe $\mathcal{E}$; en haut à droite, le pivot en rouge ; en haut à gauche, les angles avec le pivot ; en bas à droite, le parcours définit en partant du point de pivot ; le bas à gauche, l'enveloppe convexe $\mathcal{E}$ résultante.

```
class Lpt {
  constructor(cnv, n) {
    this.L = [];
    this.LA = [];
    for (let i = 0; i < n; i++) {
      let new_xi = 50+Math.floor(Math.random() * (cnv.width-100));
      let new_yi = 50+Math.floor(Math.random() * (cnv.height-100));
      this.L.push(new Pt(new_xi, new_yi));
    }
  }
  getPivot() {
    let ret = 0;
    for (let i = 1; i < this.L.length; i++) {
      if(this.L[i].y < this.L[ret].y) ret = i;
      else if(this.L[i].y == this.L[ret].y)
        if(this.L[i].x < this.L[ret].x) ret = i;
    }
    return ret;
  }
  sortByAngle(pivot) {
    this.LA = [];
    for (let i = 0; i < this.L.length; i++) {
      let dx = this.L[i].x - this.L[pivot].x;
      let dy = this.L[i].y - this.L[pivot].y;
      this.LA.push([i, Math.atan2(dy,dx)]);
    }
    this.LA.sort(function(a, b){return a[1] - b[1]});
  }
  cwReduction() {
    for (let i = 1; i < this.LA.length; i++) {
      let ii = this.LA[i][0];
      let prev = this.LA[i-1][0];
      let next = this.LA[0][0];
      if(i != this.LA.length-1) next = this.LA[i+1][0];
      if(cw(this.L[prev], this.L[ii], this.L[next]) < 0) {
        this.LA.splice(i, 1);
        return true;
      }
    }
    return false;
  }
}
```

```
let cnv = document.getElementById("myCanvas");
let ctx = cnv.getContext("2d");
let pts = new Lpt(cnv, 20);
let p = pts.getPivot();
pts.sortByAngle(p);
while(true) if(pts.cwReduction() == false) break;
draw();
function draw() {
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  for (let i = 0; i < pts.L.length; i++) {
    ctx.beginPath();
    ctx.lineWidth = 1;
    ctx.strokeStyle = "black";
    ctx.rect(pts.L[i].x-4,pts.L[i].y-4, 8,8);
    ctx.stroke();
    ctx.closePath();
  }
  for (let i = 0; i < pts.LA.length; i++) {
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    if(i == (pts.LA.length-1)) {
      ctx.moveTo(pts.L[pts.LA[i][0]].x,pts.L[pts.LA[i][0]].y);
      ctx.lineTo(pts.L[pts.LA[0][0]].x,pts.L[pts.LA[0][0]].y);
    } else {
      ctx.moveTo(pts.L[pts.LA[i][0]].x,pts.L[pts.LA[i][0]].y);
      ctx.lineTo(pts.L[pts.LA[i+1][0]].x,pts.L[pts.LA[i+1][0]].y);
    }
    ctx.stroke();
    ctx.closePath();
  }
}
```

4.7 Appartenance d'un point à un polygone convexe

En utilisant la fonction `cw` (définie section 4.3), un point appartient à un polygone convexe si :
- ce polygone est défini par une liste $\mathcal{P}$ de points ordonnés dans un unique sens (le sens peut être horaire ou anti-horaire mais doit être fixé) ;
- ce point est toujours du même côté de chacun des segments $[\mathcal{P}[i], \mathcal{P}[i+1]]$;

Autrement dit, pour le polygone convexe $\mathcal{P}$ (créé à la section 4.6) décrit dans le sens horaire, un point $a \in \mathcal{P}$:
- pour $i \in [0, n-2]$, ssi $\texttt{cw}(\mathcal{P}[i], \mathcal{P}[i+1], a) > 0$;
- pour $i = n-1$, ssi $\texttt{cw}(\mathcal{P}[i], \mathcal{P}[0], a) > 0$.

Pour illustrer l'appartenance à un polygone convexe $\mathcal{P}$, on dessine un polygone convexe et des points tirés aléatoirement comme présenté en Fig. 27 ; les points appartenant à $\mathcal{P}$ sont représentés par des rectangles rouges ; les points n'appartenant pas à $\mathcal{P}$ sont représentés par des rectangles gris clair ; on anime $\mathcal{P}$ en le faisant tourner autour de son barycentre.

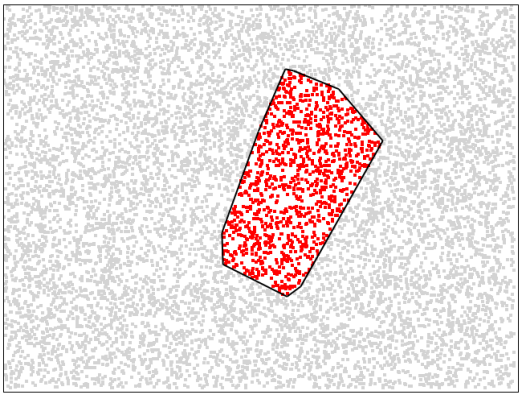


FIG. 27 – Appartenance à un polygone convexe.

La fonction `rotate` permet de faire tourner un point d'un angle `angle` autour d'un point de référence `ref`.

```
class Pt {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  rotate(ref, angle) {
    let dx = this.x - ref.x;
    let dy = this.y - ref.y;
    let da = (Math.PI / 180) * angle;
    this.x = ref.x + dx*Math.cos(da) + dy*Math.sin(da);
    this.y = ref.y + dy*Math.cos(da) - dx*Math.sin(da);
  }
}
```

La fonction `cwConstant` retourne vrai si le point `a` est dans le polygone `P`.

On défnit un polygone `P` (lignes 22 à 24).

On définit une liste `L` de 10000 points tirés aléatoirement (lignes 25 à 30).

On définit `center` le barycentre de `P` (lignes 31 à 37).

```
13 function cwConstant(a, P) {
14   for (let i = 0; i < P.length; i++) {
15     if(i == P.length-1) { if(cw(P[i], P[0], a) < 0) return false; }
16     else if(cw(P[i], P[i+1], a) < 0) return false;
17   }
18   return true;
19 }
20 let cnv = document.getElementById("myCanvas");
21 let ctx = cnv.getContext("2d");
22 let P = [new Pt(391, 157), new Pt(475, 159), new Pt(477, 169),
23          new Pt(477, 230), new Pt(438, 305), new Pt(232, 278),
24          new Pt(214, 267), new Pt(221, 178), new Pt(257, 162)];
25 let L = [];
26 for (let i = 0; i < 10000; i++) {
27   let new_xi = Math.floor(Math.random() * (cnv.width-4));
28   let new_yi = Math.floor(Math.random() * (cnv.height-4));
29   L.push(new Pt(new_xi, new_yi));
30 }
31 let center = new Pt(0,0);
32 for (let i = 0; i < P.length; i++) {
33   center.x += P[i].x;
34   center.y += P[i].y;
35 }
36 center.x /= P.length;
37 center.y /= P.length;
```

La fonction `update_pos` fait tourner chacun des points de `P` autour du barycentre de `P`.

La fonction `draw` différencie les points dans `P` (ligne 48) et les points hors de `P` (ligne 49) ; les lignes 54 à 67 dessinent le contour de `P`.

```
38 function update_pos() {
39   for (let i = 0; i < P.length; i++) {
40     P[i].rotate(center, 2);
41   }
42 }
43 function draw() {
44   ctx.clearRect(0, 0, cnv.width, cnv.height);
45   for (let i = 0; i < L.length; i++) {
46     ctx.beginPath();
47     ctx.lineWidth = 1;
48     if(cwConstant(L[i], P)) ctx.fillStyle = "red";
49     else ctx.fillStyle = "lightgray";
50     ctx.rect(L[i].x-2,L[i].y-2, 4,4);
51     ctx.fill();
52     ctx.closePath();
53   }
54   for (let i = 0; i < P.length; i++) {
55     ctx.beginPath();
56     ctx.lineWidth = 2;
57     ctx.strokeStyle = "black";
58     if(i == (P.length-1)) {
59       ctx.moveTo(P[i].x,P[i].y);
60       ctx.lineTo(P[0].x,P[0].y);
61     } else {
62       ctx.moveTo(P[i].x,P[i].y);
63       ctx.lineTo(P[i+1].x,P[i+1].y);
64     }
65     ctx.stroke();
66     ctx.closePath();
67   }
68 }
69 function update(timestamp) {
70   update_pos();
71   draw();
72   requestAnimationFrame(update);
73 }
74 requestAnimationFrame(update);
```

4.8 Appartenance d'un point à un polygone quelconque

En utilisant la fonction `intersectionTest` (définie section 4.4), un point appartient à un polygone quelconque :

- pour un segment $\mathcal{S}$ reliant le point considéré et un point infini ;
- pour un nombre n d'intersections entre $\mathcal{S}$ et les côtés de $\mathcal{P}$;
- si n est pair, alors le point considéré est en dehors de $\mathcal{P}$;
- si n est impair, alors le point considéré est dans $\mathcal{P}$.

Pour illustrer l'appartenance à un polygone quelconque $\mathcal{P}$, on dessine un polygone concave et des points tirés aléatoirement comme présenté en Fig. 28 ; les points appartenant à $\mathcal{P}$ sont représentés par des rectangles rouges ; les points n'appartenant pas à $\mathcal{P}$ sont représentés par des rectangles gris clair ; on anime $\mathcal{P}$ en le faisant tourner autour de son barycentre.

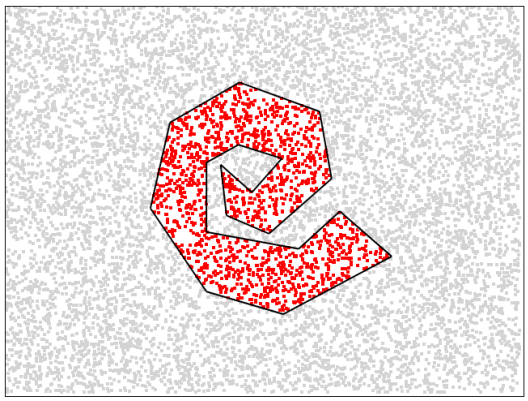


FIG. 28 – Appartenance à un polygone quelconque.

La fonction `isIn` retourne vrai si le point `p0` est dans le polygone défini par `L`.

```
function isIn(p0, L) {
  let p1 = new Pt(0,0);
  let s0 = new Sgt(p0, p1);
  let nb_inters = 0;
  for (let i = 0; i < L.length; i++) {
    let s1;
    if(i == L.length-1) s1 = new Sgt(L[i], L[0]);
    else s1 = new Sgt(L[i], L[i+1]);
    if(s0.intersectionTest(s1)) nb_inters += 1;
  }
  return ((nb_inters%2) == 1);
}
```

On défnit un polygone `P` (lignes 15 à 20).

On définit une liste `L` de 10000 points tirés aléatoirement (lignes 21 à 26).

On définit `center` le barycentre de `P` (lignes 27 à 33).

```
13 let cnv = document.getElementById("myCanvas");
14 let ctx = cnv.getContext("2d");
15 let P = [new Pt(215,79), new Pt(366,95), new Pt(429,171),
16          new Pt(429,295), new Pt(350,369), new Pt(251,362),
17          new Pt(189,277), new Pt(223,200), new Pt(325,187),
18          new Pt(356,235), new Pt(327,291), new Pt(314,241),
19          new Pt(260,255), new Pt(295,299), new Pt(340,303),
20          new Pt(388,232), new Pt(305,151), new Pt(237,161)];
21 let L = [];
22 for (let i = 0; i < 10000; i++) {
23   let new_xi = 2+Math.floor(Math.random() * (cnv.width-4));
24   let new_yi = 2+Math.floor(Math.random() * (cnv.height-4));
25   L.push(new Pt(new_xi, new_yi));
26 }
27 let center = new Pt(0,0);
28 for (let i = 0; i < P.length; i++) {
29   center.x += P[i].x;
30   center.y += P[i].y;
31 }
32 center.x /= P.length;
33 center.y /= P.length;
```

La fonction `update_pos` fait tourner chacun des points de `P` autour du barycentre de `P`.

La fonction `draw` différencie les points dans `P` (ligne 44) et les points hors de `P` (ligne 45) ; les lignes 50 à 63 dessinent le contour de `P`.

```
34 function update_pos() {
35   for (let i = 0; i < P.length; i++) {
36     P[i].rotate(center, -1);
37   }
38 }
39 function draw() {
40   ctx.clearRect(0, 0, cnv.width, cnv.height);
41   for (let i = 0; i < L.length; i++) {
42     ctx.beginPath();
43     ctx.lineWidth = 1;
44     if(isIn(L[i], P)) ctx.fillStyle = "red";
45     else ctx.fillStyle = "lightgray";
46     ctx.rect(L[i].x-2,L[i].y-2, 4,4);
47     ctx.fill();
48     ctx.closePath();
49   }
50   for (let i = 0; i < P.length; i++) {
51     ctx.beginPath();
52     ctx.lineWidth = 2;
53     ctx.strokeStyle = "black";
54     if(i == (P.length-1)) {
55       ctx.moveTo(P[i].x,P[i].y);
56       ctx.lineTo(P[0].x,P[0].y);
57     } else {
58       ctx.moveTo(P[i].x,P[i].y);
59       ctx.lineTo(P[i+1].x,P[i+1].y);
60     }
61     ctx.stroke();
62     ctx.closePath();
63   }
64 }
65 function update(timestamp) {
66   update_pos();
67   draw();
68   requestAnimationFrame(update);
69 }
70 requestAnimationFrame(update);
```

4.9 Collision entre deux polygones

Dans le cas de deux polygones $\mathcal{P}_1$ et $\mathcal{P}_2$, deux solutions sont possibles :

- la première solution consiste à vérifier que chacun des points de $\mathcal{P}_1$ n'appartient pas à $\mathcal{P}_2$;
- la seconde solution consiste à réduire l'un des deux polygones par un point et à augmenter l'autre polygone par la somme des deux polygones ; cette somme est appelée la somme de Minkowski et est notée $\bigoplus$; avec $\mathcal{P}_1 \bigoplus \mathcal{P}_2$, on obtient un nouveau polygone $\mathcal{P}_3$ en ajoutant les contours de $\mathcal{P}_2$ sur la position et les contours de $\mathcal{P}_1$; pour vérifier l'absence de collision entre $\mathcal{P}_1$ et $\mathcal{P}_2$, on vérifie que le premier point de $\mathcal{P}_2$ n'appartient pas à $\mathcal{P}_3$.

4.10 Application à un jeu de combat

Dans un jeu tel que Street Fighter II (dont Fig. 29 présente le personnage Ryu dans trois positions), il sera possible d'utiliser différentes solutions pour la détection des collisions entre personnages et éléments de décor.

FIG. 29 – Trois positions de Ryu dans Street Fighter II.

Une solution classique, appelée PixelPerfect, est de regarder pixel par pixel les collisions ; le principe est de parcourir le premier sprite, si le pixel est un pixel à considérer, alors regarder le pixel du second sprite ; on peut soit s'arrêter au premier pixel en collision, soit rechercher tous les pixels en collision pour calculer par exemple le barycentre des pixels en collision pour chaque personnage ; pour un sprite, on peut utiliser soit la composante alpha, soit une autre image binaire qui présente les pixels du sprite à considérer pour les collisions ; dans le cas du sprite de Ryu, utiliser la composante alpha n'est pas suffisant à cause de l'ombre sur le sol ; être en collision avec l'ombre sur le sol n'est pas être en collision avec Ryu ; il faudra dans ce cas utiliser une autre image, comme présenté en Fig. 30 ; pour deux personnages, on utilisera les deux masques correspondant pour détecter les collisions.

FIG. 30 – Formes AABB et masques PixelPerfect pour trois positions Ryu.

4.11 Application à un jeu de type Shoot Them Up

Dans un jeu de type Shoot Them Up (abrégé shmup), les formes varient ; il sera possible d'utiliser différentes solutions (AABB, cercles, PixelPerfect tel que présenté dans les Fig. 31 et 32) pour la détection des collisions entre vaisseaux et éléments de décor.

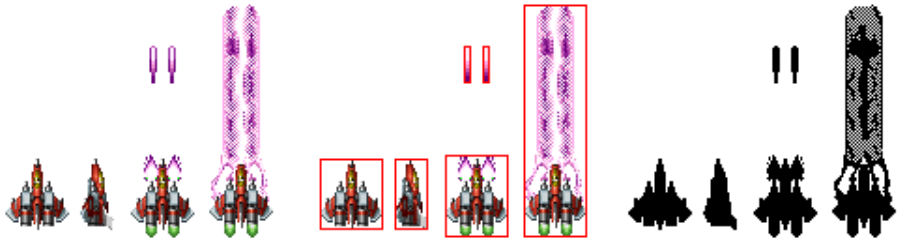


FIG. 31 – Formes et volumes englobants pour le jeu Raïden-MKII.

FIG. 32 – Formes et volumes englobants pour des vaisseaux de forme circulaire.

Alg. 2 présente la fonction de test de collision entre un volume AABB nommé $\mathcal{A}$ et un cercle nommé $\mathcal{C}$; la ligne 1 vérifie la non-appartenance du centre du cercle au volume $\mathcal{A}$; ce test se réalise soit par le test d'apartenance d'un point à un polygone convexe (présenté section 4.7), soit par le test d'apartenance d'un point à un polygone quelconque (présenté section 4.8) ; la fonction `closest` (ligne 5) retourne l'index du point de $\mathcal{L}$ le plus proche du centre de $\mathcal{C}$, pour les index de $\mathcal{L}$ compris entre 1 et 4 ; les tests d'intersection (lignes 6 et 7) correspondent au test d'intersection entre segment et cercle (présenté section 4.5).

```
1 fonction collide ( A, C ) :
2     if C.center ∈ A then return true ;
3     L ← [[A.minx, A.miny], [A.minx, A.maxy], [A.maxx, A.maxy],
4          [A.maxx, A.miny], [A.minx, A.miny], [A.minx, A.maxy]] ;
5     i ← closest ( L , 1, 4, C.center ) ;
6     if intersection ( [L[i − 1], L[i]] , C ) then return true ;
7     if intersection ( [L[i], L[i + 1]] , C ) then return true ;
8     return false ;
```

Alg. 2: Intersection entre volume AABB et cercle.

4.12 Librairie `SAT.js`

La librairie `SAT.js` (disponible sur `github.com/jriecken/sat-js`) propose des fonctions de détection de collisions 2D ; SAT vient de `Separating Axis Theorem` ; elle permet de détecter les collisions entre cercles et polygones convexes ; pour les formes convexes, elle utilise les volume AABB ; elle permet également de savoir si un point est dans un cercle ou dans un polygone convexe ; on place le fichier `SAT.js` dans le répertoire `js` et on ajoute la ligne suivante dans le fichier html.

```
1 <script src="./js/SAT.js"></script>
```

Un point est un `Vector` de deux coordonnées `x` et `y`.

```
34  let x, y;
35  let aPoint = new SAT.Vector(x, y);
```

Sur le `Vector`, on dispose des fonctions suivantes :

- Copier avec `copy(anotherVector)`
- Dupliquer avec `clone()`
- Tranformer en son vecteur orthogonal avec `perp()`
- Tourner dans le sens horaire d'un angle en radians avec `rotate(angle)`
- Inverser avec `reverse()`
- Normaliser avec `normalize()`
- Additionner un autre `Vector` avec `add(other)`
- Soustraire un autre `Vector` avec `sub(other)`
- Projeter dans une direction avec `scale(x,y)`
- Projeter sur un autre `Vector` avec `project(other)`
- Projeter sur un autre `Vector` unitaire avec `projectN(other)`
- Réfléchir sur un axe `reflect(axis)`
- Réfléchir sur un axe unitaire `reflectN(axis)`
- Calculer le produit scalaire avec un autre `Vector` avec `dot(other)`
- Calculer le carré de la longueur avec `len2()`
- Calculer la longueur avec `len()`

Un cercle est un `Circle` avec un centre de coordonnées `x` et `y`, et un rayon `r`.

```
34  let x, y, r;
35  let aCircle = new SAT.Circle(new SAT.Vector(x,y), r);
```

Un cercle possède les propriétés suivantes :

- `pos` pour sa position
- `r` pour son rayon
- `offset` pour le décalage de son centre par rapport à sa position

Sur le `Circle`, on dispose des fonctions suivantes :

- Redéfinir son décalage avec `setOffset(offset)`
- Calculer le polygone de sa boite AABB avec `getAABB()`
- Calculer sa boite AABB avec `getAABBAsBox()`

Un polygone est un `Polygon` et correspond à une liste de points énumérés dans le sens anti-horaire.

```
let aPolygon = new SAT.Polygon(new SAT.Vector(0,0),
  [new SAT.Vector(),
    new SAT.Vector(100,0),
    new SAT.Vector(100,100),
    new SAT.Vector(0,100),
    new SAT.Vector(0,0)
  ]);
```

Un polygone possède les propriétés suivantes :

- `pos` pour sa position (qui implique que les points de son contour sont en coordonnées relatives par rapport à cette position)
- `points` pour les points de son contour
- `angle` pour son angle en radians
- `offset` pour la translation à appliquer avant toute rotation des points de contour
- `calcPoints` pour ses points de collision en coordonnées globales
- `edges` pour ses contours
- `normals` pour ses normales

Sur le `Polygon`, on dispose des fonctions suivantes :

- Redéfinir ses points de contour avec `setPoints(points)`
- Redéfinir son angle en radians avec `setAngle(angle)`
- Redéfinir le décalage de ses points par rapport à sa position avec `setOffset(offset)`
- Réaliser une rotation d'un angle en radians avec `rotate(angle)`
- Translater l'ensemble de ses points de contour avec `translate(x,y)`
- Calculer le polygone de sa boite AABB avec `getAABB()`
- Calculer sa boite AABB avec `getAABBAsBox()`
- Calculer son barycentre avec `getCentroid()`

Une boite AABB est une `Box` et correspond à une position définie par deux coordonnées `x` et `y`, avec une largeur `w` et une hauteur `h`.

```
let x, y, w, h;
let aBox = new SAT.Box(new SAT.Vector(x,y), w, h);
```

Une boite AABB possède les propriétés suivantes :

- `pos` pour sa position correspondant au coin supérieur gauche de la boite AABB
- `w` pour sa largeur
- `h` pour sa hauteur

Sur la `Box`, on dispose de fonctions suivante :

- Calculer le polygone de sa boite AABB avec `toPolygon()`

Une réponse à une détection de collision entre deux objets est une `Response`.

Une réponse à une détection de collision possède les propriétés suivantes :

— `a` pour le premier objet concerné par la collision
— `b` pour le deuxième objet concerné par la collision
— `overlap` pour la pénétration associée à la collision
— `overlapN` pour le vecteur unitaire définissant la pénétration
— `overlapV` pour le vecteur définissant la pénétration
— `aInB` un booléen pour indiquer si `a` est inclus dans `b`
— `bInA` un booléen pour indiquer si `b` est inclus dans `a`

Sur la `Response`, on dispose de fonctions suivante :
— Réinitialiser une réponse avec `clear()`

Pour les tests de collision, on dispose des fonctions suivantes :
— Savoir si un point est dans un cercle avec `pointInCircle(p, c)`
— Savoir si un point est dans un polygone avec `pointInCircle(p, poly)`
— Savoir si deux cercles sont en collision avec `testCircleCircle(a, b, response)`
— Savoir si un polygone et un cercle sont en collision avec `testPolygonCircle(poly, c, response)`
— Savoir si un cercle et un polygone sont en collision avec `testCirclePolygon(c, poly, response)`
— Savoir si deux polygones sont en collision avec `testPolygonPolygon(a, b, response)`

On considère l'image présentée en Fig. 33 ; n'ayant pas de détection de collision sur les formes concaves, on approxime Ken avec 3 rectangles et la voiture avec un polygone convexe comme présenté en Fig. 34 ; la question est de savoir si les polygones représentants Ken sont en collision avec la voiture.

FIG. 33 – Ken dans le niveau bonus de destruction de la voiture verte.

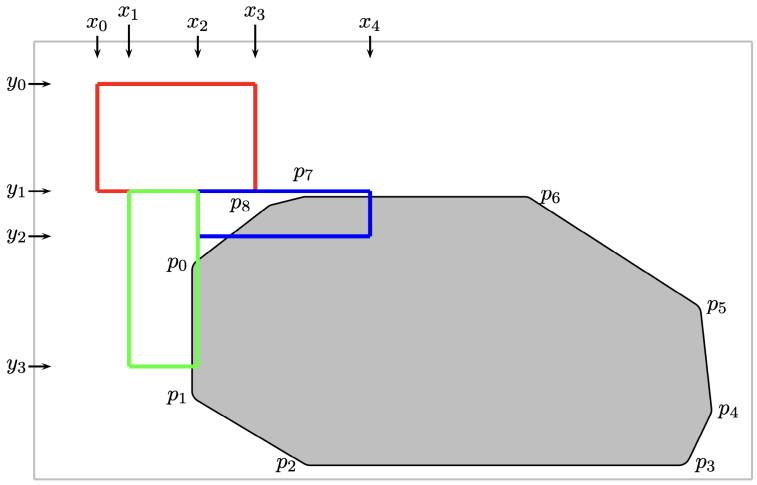


FIG. 34 – Ken dans le niveau bonus de destruction de la voiture verte.

p_0 à p_8 sont des points dont les coordonnées sont $(55, 79)$, $(55, 126)$, $(95, 141)$, $(227, 141)$, $(239, 114)$, $(232, 94)$, $(172, 55)$, $(94, 55)$, $(82, 58)$.
x_0 à x_4 sont des coordonnées sur l'axe Ox de valeurs $22, 33, 56, 75, 117$.
y_0 à y_3 sont des coordonnées sur l'axe Oy de valeurs $15, 53, 69, 115$.
On se place dans un Canvas de 250 en largeur par 155 en hauteur.

```
let cnv = document.getElementById("myCanvas");
let ctx = cnv.getContext("2d");
ctx.imageSmoothingEnabled= false;

let red_box = new SAT.Box(new SAT.Vector(22, 15), 75-22, 53-15);
let blue_box = new SAT.Box(new SAT.Vector(56, 53), 117-56, 69-53);
let green_box = new SAT.Box(new SAT.Vector(33, 53), 56-33, 115-53);
let car_polygon = new SAT.Polygon(new SAT.Vector(55,79),
    [new SAT.Vector(),
      new SAT.Vector(0,47),
      new SAT.Vector(40,62),
      new SAT.Vector(172,62),
      new SAT.Vector(184,35),
      new SAT.Vector(177,15),
      new SAT.Vector(117,-24),
      new SAT.Vector(39,-24),
      new SAT.Vector(27,-21)
]);
let red_collision = false;
let blue_collision = false;
let green_collision = false;
if(SAT.testPolygonPolygon(car_polygon, red_box.toPolygon())) {
  console.log("red_box collision"); red_collision = true;
}
if(SAT.testPolygonPolygon(car_polygon, blue_box.toPolygon())) {
  console.log("blue_box collision"); blue_collision = true;
}
if(SAT.testPolygonPolygon(car_polygon, green_box.toPolygon())) {
  console.log("green_box collision"); green_collision = true;
}
function draw_SAT_box(ctx, box, fill, stroke, color) {
  ctx.beginPath();
  ctx.rect(box.pos.x, box.pos.y, box.w, box.h);
  if(fill) {
    ctx.fillStyle = color;
    ctx.fill();
  } else if(stroke) {
    ctx.strokeStyle = color;
    ctx.stroke();
  }
  ctx.closePath();
}
```

```
function draw() {
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  if(red_collision) {
    draw_SAT_box(ctx, red_box, true, false, "red");
  }
  if(blue_collision) {
    draw_SAT_box(ctx, blue_box, true, false, "red");
  }
  if(green_collision) {
    draw_SAT_box(ctx, green_box, true, false, "red");
  }
  draw_SAT_box(ctx, red_box, false, true, "red");
  draw_SAT_box(ctx, blue_box, false, true, "blue");
  draw_SAT_box(ctx, green_box, false, true, "green");
  for (let i = 0; i < car_polygon.points.length; i++) {
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    if(i == (car_polygon.points.length-1)) {
      ctx.moveTo(car_polygon.pos.x+car_polygon.points[i].x,
        car_polygon.pos.y+car_polygon.points[i].y);
      ctx.lineTo(car_polygon.pos.x+car_polygon.points[0].x,
        car_polygon.pos.y+car_polygon.points[0].y);
    } else {
      ctx.moveTo(car_polygon.pos.x+car_polygon.points[i].x,
        car_polygon.pos.y+car_polygon.points[i].y);
      ctx.lineTo(car_polygon.pos.x+car_polygon.points[i+1].x,
        car_polygon.pos.y+car_polygon.points[i+1].y);
    }
    ctx.stroke();
    ctx.closePath();
  }
}
function update(timestamp) {
  draw();
  requestAnimationFrame(update);
}
requestAnimationFrame(update);
```

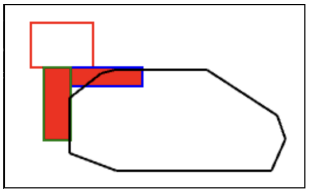


FIG. 35 – Résultat dans un Canvas de 250 par 155.