src/src/solid.js

```
/**
 * @file Describes ADSODA solids
 * @author Jeff Bigot <jeff@raktres.net> after Greg Ferrar
 * @class Solid
 * @extends NDObject
 */

import { NDObject } from './ndobject.js'
import { Face } from './face.js'
import { v4 as uuidv4 } from 'uuid'
import {
  projectVector,
  isCornerEqual,
  moizeAmongIndex,
  findnormal,
  positionPoint,
  intersectHyperplanes,
  isHPEqual
} from './halfspace.js'
import * as P from './parameters.js'

function uniqBy (a, key) {
  const seen = {}
  return a.filter(function (item) {
    const k = key(item)
    return seen.hasOwnProperty(k) ? false : (seen[k] = true)
  })
}

class Solid extends NDObject {
  /*
   * @constructor solid
   * @param {string} dimension nb of dimensions of the solid
   */
  constructor (dim, halfspaces = false) {
    super('solid')
    this.dimension = dim
    this.faces = []
    this.corners = []
    this.silhouettes = []
    this.adjacenciesValid = false
    this.cornersValid = false
    this.center = []
    this.adjacentRefs = new Set()
    // TODO: verifi à quoi ça sert
    if (halfspaces) {
      const _t = this
      const halffiltered = uniqBy(halfspaces, JSON.stringify)
      halffiltered.forEach(HS => _t.suffixFace(new Face(HS)))
      this.ensureFaces()
```

```
51      }
52      this.uuid = uuidv4()
53    }
54
55    clone () {
56      const newSolid = new Solid(
57        this.dimension,
58        [...this.faces].map(f => [...f.equ])
59      )
60      newSolid.color = this.color
61      newSolid.name = this.name
62      return newSolid
63    }
64
65    /**
66     * @returns JSON
67     */
68    exportToJSON () {
69      let json = `{ "solidname" : "${this.name}" , "dimension" : ${this.dimension}
  ,
70           "color" : "${this.color}" , "faces" : [ `
71      json += [...this.faces].map(face => face.exportToJSON()).join(',')
72      json += ']}'
73      return json
74    }
75
76    /**
77     * create a face from a json
78     * @param {JSON} JSON (not a string)
79     */
80    static importFromJSON (json) {
81      const sol = new Solid(parseInt(json.dimension))
82      sol.name = json.solidname
83      sol.id = json.id || 0
84      sol.color = json.color;
85      [...json.faces].forEach(fac => {
86        sol.addFace(Face.importFromJSON(fac))
87      })
88      return sol
89    }
90
91    /**
92     * printable summary
93     * */
94    logSummary () {
95      return `Solid name : ${this.name} | dim : ${
96        this.dimension
97      } \n --- nb of faces : ${this.faces.length} \n --- nb of corners ${
98        this.corners.length
99      } corners : ${JSON.stringify(this.corners)}
100         \n --- nb of silhouettes : ${
101           this.silhouettes.length
102         } \n --- adjacencies valid ? : ${this.adjacenciesValid} `
```

```
103   }
104
105   /**
106    * @returns txt
107    */
108   logDetail () {
109     return `Solid name : ${this.name}
110          \n --- corners ${JSON.stringify(this.corners)} \n --- nb of faces : ${
111       this.faces.length
112     } `
113   }
114
115   /**
116    *
117    * @param {*} face
118    */
119   suffixFace (face) {
120     this.faces = [...this.faces, face]
121   }
122
123   /**
124    *
125    * @param {*} corner
126    * @return boolean true if corner added
127    */
128   suffixCorner (corner) {
129     if (!this.corners.find(corn => isCornerEqual(corn, corner))) {
130       this.corners = [...this.corners, corner]
131       return true
132     } else {
133       return false
134     }
135   }
136
137   /**
138    * erase corners
139    */
140   eraseCorners () {
141     this.corners.length = 0
142     this.center.length = 0
143     this.cornersValid = false
144     this.faces.forEach(face => { face.touchingCorners.length = 0 })
145   }
146
147   /**
148    * clear adajcent faces
149    */
150   eraseOldAdjacencies () {
151     this.faces.forEach(face => {
152       face.adjacentRefs.clear()
153       face.eraseTouchingCorners()
154       face.eraseAdjacentFaces()
155     })
```

```
156     this.adjacenciesValid = false
157     this.eraseCorners()
158   }
159
160   /**
161    *
162    */
163   eraseSilhouettes () {
164     this.silhouettes.length = 0
165   }
166
167   /**
168    * @todo vérifrie si nouvelle condition de filtre est utile
169    */
170   filterRealFaces () {
171     const _t = this
172     this.faces = [...this.faces].filter(
173       face =>
174       // TODO: vérifier le fonctionnement des filtres de face
175         face.isRealFace() && [...face.touchingCorners].find( corner =>
    _t.isPointInsideOrOnSolid(corner))
176     )
177   }
178
179   /**
180    *
181    * @param {*} point
182    * @todo utiliser isPointInsideFaces
183    */
184   isPointInsideSolid (point) {
185     return [...this.faces].every(face => face.isPointInsideFace(point))
186   }
187
188   /**
189    *
190    * @param {*} point
191    */
192   isPointInsideOrOnSolid (point) {
193     return [...this.faces].every(face => face.isPointInsideOrOnFace(point))
194   }
195
196   /**
197    * @todo utile de tout écraser à chaque fois ?
198    * TODO: voir si peut garder les cotés adajcents, recalculer les points
199    */
200   unvalidSolid () {
201     this.eraseOldAdjacencies() // inclue l'effacage des points
202     this.eraseSilhouettes()
203   }
204
205   /**
206    * on supprime juste les points et les silhouettes
207    * TODO: voir si peut garder les cotés adajcents, recalculer les points
```

```
208     */
209   forceUpdateSolid () {
210     this.eraseCorners()
211     this.eraseSilhouettes()
212   }
213
214   /**
215    *
216    * @param {*} face
217    */
218   addFace (face) {
219     this.suffixFace(face)
220     this.unvalidSolid()
221   }
222
223   /**
224    *
225    * @param {*} halfspace
226    */
227   cutWith (halfspace) {
228     this.addFace(new Face(halfspace))
229   }
230
231   /**
232    * mutata this
233    * @param {*} face
234    * @return the other part
235    */
236   sliceWith (face) {
237     // Attention mutation de this et retour d'un nouveau solide
238     // vérifier que c'est bon
239     const equ = [...face.equ]
240     const solid1 = this.clone()
241     solid1.name = this.name + '/outer/'
242
243     const flipEqu = equ.map(coord => -coord)
244     this.name = this.name + '/inner/'
245     solid1.cutWith(flipEqu)
246
247     this.cutWith(equ)
248     return solid1
249   }
250
251   /**
252    *
253    */
254   isNonEmptySolid () {
255     this.ensureFaces()
256     return this.dimension < this.corners.length
257   }
258
259   /**
260    *
```

```
261     * @param {*} corner
262     */
263   isCornerAdded (corner) {
264     if (!this.isPointInsideOrOnSolid(corner)) {
265       return false
266     }
267     const str = JSON.stringify(corner)
268     if (this.corners.find(el => str === JSON.stringify(el))) {
269       return false
270     }
271     this.suffixCorner(corner)
272     return true
273   }
274
275   /**
276    *
277    * @param {*} faces
278    * @param {*} corner
279    */
280   processCorner (facesref, corner) {
281     if (this.isCornerAdded(corner)) {
282       Face.updateAdjacentFacesRefs(this.faces, facesref, corner)
283     } else {
284       //
285     }
286   }
287
288   /**
289    *
290    * @param {*} facesref
291    * TODO: vérifier que les nouvelles conditions du if sont satisfaisantes
292    */
293   computeIntersection (facesref) {
294     // recherche d'une intersection
295     const intersection = Face.facesRefIntersection(this.faces, facesref)
296     // TODO: pourquoi l'intersection pourrait ne pas être sur les faces ? mais en
fait si !!!
297     if (intersection) { // && facesref.every(ref =>
this.faces[ref].isPointOnFace(intersection))
298       this.processCorner(facesref, intersection)
299     }
300   }
301
302   /**
303    *
304    */
305   computeAdjacencies () {
306     const groupsFaces = moizeAmongIndex(this.faces.length, this.dimension,
P.MAX_FACE_PER_CORNER) // this.dimension) //
307     const n = groupsFaces.length
308     for (let index = 0; index < n; index++) {
309       this.computeIntersection(groupsFaces[index])
310     }
```

```
311   }
312
313   computeCorners () {
314     const _t = this
315     this.faces.forEach(face => {
316       const nbAjaFaces = face.adjacentRefs.size
317       const groupsRefFaces = moizeAmongIndex(nbAjaFaces, _t.dimension - 1,
      _t.dimension - 1) // P.MAX_FACE_PER_CORNER) // _t.dimension-1)
318       const n = groupsRefFaces.length
319       const adjacentEqu = []
320       face.adjacentRefs.forEach(element => {
321         adjacentEqu.push(this.faces[element].equ)
322       })
323       for (let index = 0; index < n; index++) {
324         const grp = [face.equ].concat(groupsRefFaces[index].map(id =>
      adjacentEqu[id]))
325         const intersection = intersectHyperplanes(grp)
326         if (intersection && _t.isPointInsideOrOnSolid(intersection)) {
327           face.touchingCorners.push(intersection)
328         }
329       }
330     })
331     let cornerList = []
332     // TODO, voir pour remplacer uniq par la fonction infra
333     this.faces.forEach(face => {
334       cornerList = cornerList.concat([...face.touchingCorners])
335     })
336     _t.corners.length = 0
337     cornerList.forEach((corner, idx) => {
338       for (let index = idx + 1; index < cornerList.length; index++) {
339         if (isCornerEqual(corner, cornerList[index])) { return false }
340       }
341       _t.corners.push(corner)
342     })
343   }
344
345   /**
346    * TODO: ne pas mélanger avec find corners
347    */
348   findAdjacencies () {
349     this.eraseOldAdjacencies()
350     this.computeAdjacencies()
351     // TODO: erreur dans certains cas
352     const nba = this.faces.length
353     this.filterRealFaces()
354     if (this.faces.length !== nba) {
355       this.eraseOldAdjacencies()
356       this.computeAdjacencies()
357     }
358     this.adjacenciesValid = true
359     this.cornersValid = true
360   }
361
```

```
362   /**
363    * TODO: À rédiger !!!
364    */
365   findCorners () {
366     this.eraseCorners()
367     this.computeCorners()
368     this.cornersValid = true
369   }
370
371   /**
372    *
373    */
374   ensureFaces () {
375     if (!this.adjacenciesValid) {
376       this.findAdjacencies()
377     } else if (!this.cornersValid) {
378       this.findCorners()
379     } else {
380       //
381     }
382   }
383
384   /**
385    *
386    */
387   ensureSilhouettes () {
388     if (this.silhouettes.length === 0) {
389       for (let i = 0; i < this.dimension; i++) {
390         this.silhouettes[i] = this.createSilhouette(i)
391       }
392     }
393   }
394
395   /**
396    *
397    * @param {*} axe
398    * @return {array} silhouettes
399    * TODO: correction, ne pas ajouter la silhouette si existe déjà
400    */
401   createSilhouette (axe) {
402     const sFaces = [...this.faces]
403     let sil = []
404     const n = sFaces.length
405     for (let index = 0; index < n; index++) {
406       const tmp = sFaces[index].silhouette(axe, sFaces, this.center)
407       if (tmp) {
408         sil = [...sil, ...tmp]
409       }
410     }
411     const nb = sil.length
412     const filteredSil = []
413     sil.forEach((face, idx) => {
414       let res = true
```

```
415       for (let index = idx + 1; index < nb; index++) {
416         if (isHPEqual(face.equ, sil[index].equ, P.VERY_SMALL_NUM)) {
417           res = false
418           break
419         }
420       }
421       if (res) filteredSil.push(face)
422     })
423     return filteredSil
424   }
425
426   /**
427    *
428    * @param {*} axe
429    * @return {array} silhouettes
430    * TODO: correction, ne pas ajouter la silhouette si existe déjà
431    */
432
433   createAxeCutSilhouette (axe) {
434     const hype = []
435     for (let index = 0; index < this.dimension + 1; index++) {
436       hype[index] = 0
437     }
438     hype[axe] = 1
439
440     const solid1 = this.clone()
441     solid1.name = this.name + '/ cut /'
442     // attention ! le plan de coupe peut déjà être une face du solide
443     let cface = solid1.faces.find(face => {
444       return isHPEqual(hype, face.equ, 0)
445     })
446     if (!cface) {
447       cface = new Face(hype)
448       cface.name = 'cut'
449       solid1.addFace(cface)
450       solid1.ensureFaces()
451     }
452
453     const sFaces = [...solid1.faces]
454
455     const sil = cface.cutSilhouette(axe, sFaces, this.center)
456     // TODO: reprendre
457     const nb = sil.length
458     const filteredSil = []
459     // recherche de faces en doublon
460     sil.forEach((face, idx) => {
461       let res = true
462       for (let index = idx + 1; index < nb; index++) {
463         if (isCornerEqual(face.equ, sil[index].equ, P.VERY_SMALL_NUM)) {
464           res = false
465           break
466         }
467       }
```

```
468       if (res) filteredSil.push(face)
469     })
470     // TODO: voir si on retourne sil ou filteredSil
471     // return sil
472     return filteredSil
473   }
474
475   /**
476    *
477    * @param {*} axe
478    * @returns silhouettes
479    */
480   getSilhouette (axe) {
481     return this.silhouettes[axe]
482   }
483
484   /**
485    *
486    */
487   propagateSelection () {
488     const sel = this.selected
489     this.faces.forEach(face => (face.selected = sel))
490   }
491
492   /**
493    *
494    * @param {*} axe
495    * @todo for the moment, project the silhouette, need to project differents
   faces
496    * @todo vérifier que clonedeep est utile
497    * @return {array} solids
498    */
499   project (axe) {
500     // attention, dans certains cas, il a fallu forcer le recalcul du solide
501     // this.unvalidSolid()
502     this.ensureFaces()
503     this.ensureSilhouettes()
504     const _t = this
505     const halfspaces = [..._t.silhouettes[axe]].map(face =>
506       Solid.silProject(face, axe)
507     )
508     // TODO: Voir s'il y a besoin de filtrer les HS, utilisr iSHSEquel
509     // trouver d'où peut provenir cette HS à 0. Apparait pour les cubes
510     // const halfspacesf = halfspaces.filter(eq => eq.find(x => { return x >
   P.VERY_SMALL_NUM || x < -P.VERY_SMALL_NUM }))
511
512     const dim = this.dimension - 1
513     const solid = new Solid(dim, halfspaces)
514     // TODO color of projection is function of lights
515     solid.color = this.color
516     solid.selected = this.selected
517     solid.name = 'projection ' + this.name + '|P' + axe + '|'
518     solid.parentUuid = this.uuid
```

```
519     return [solid]
520   }
521
522   /**
523    *
524    * @param {*} axe
525    * @todo for the moment, project the silhouette, need to project differents
   faces
526    * @todo vérifier que clonedeep est utile
527    * @return {array} solids
528    */
529   axeCut (axe) {
530     this.ensureFaces()
531     const _t = this
532
533     const halfspaces = [..._t.createAxeCutSilhouette(axe)].map(face =>
534       Solid.silProject(face, axe)
535     )
536     // TODO: trouver d'où peut provenir cette HS à 0
537     // TODO: retrouver la fonction de vérif à 0
538     // const halfspacesf = halfspaces.filter(eq => eq.find(x => { return x >
   P.VERY_SMALL_NUM || x < -P.VERY_SMALL_NUM }))
539     // console.log('halfspaces')
540     // console.table(halfspacesf)
541     // const halfspaces = [..._t.axeCutsilhouettes[axe]].map(face =>
542     //  Solid.silProject(face, axe)
543     // )
544     const dim = this.dimension - 1
545     const solid = new Solid(dim, halfspaces)
546
547     solid.color = this.color
548     solid.selected = this.selected
549     solid.name = 'axe cut ' + this.name + '|P' + axe + '|'
550     solid.parentUuid = this.uuid
551     return [solid]
552   }
553
554   /**
555    *
556    * @param {*} vector
557    * @returns this
558    */
559   translate (vector, force = false) {
560     vector = vector.map(parseFloat)
561     this.faces.forEach(face => face.translate(vector))
562     this.forceUpdateSolid()
563     return this
564   }
565
566   /**
567    *
568    * @param {*} matrix
569    * @returns this
```

```
570     * TODO: selection des solides à translater
571     */
572    transform (matrix, center, force = false) {
573      // if (!this.selected && !force ) return this;
574      const centerl = center.map(x => -parseFloat(x))
575      this.faces = [...this.faces].map(face =>
576        face
577          .translate(centerl)
578          .transform(matrix)
579          .translate(center)
580      )
581      this.forceUpdateSolid()
582      return this
583    }
584
585    /**
586     *
587     * @param {*} solid
588     * @param {*} axe
589     * @returns corner
590     */
591    findCornerInSilhouette (solid, axe) {
592      const sil = solid.silhouettes[axe]
593      return [...this.corners].find(corner =>
594        Face.isPointInsideFaces(sil, corner)
595      )
596    }
597
598    /**
599     *
600     * @param {*} solid
601     * @param {*} point
602     * @param {*} axe
603     * @todo values behind and infront
604     * @returns -1 or 1 or false
605     *  TODO: verifier
606     */
607    checkOrientation (point, axe) {
608      for (const face of this.faces) {
609        if (!face.isPointInsideFace(point)) {
610          if (face.isBackFace(axe)) {
611            return -1
612          } else {
613            return 1
614          }
615        }
616      }
617      return false
618    }
619
620    /**
621     *
622     * @param {*} solid
```

```
623     * @param {*} axe
624     * @todo return false ou 0 ?
625     * @returns boolean
626     */
627    isInFront (solid, axe) {
628      const corner = this.findCornerInSilhouette(solid, axe)
629      if (corner) {
630        return solid.checkOrientation(corner, axe)
631      } else {
632        return false
633      }
634    }
635
636    /**
637     *
638     * @param {*} solid
639     * @param {*} axe
640     * @returns 1 if infront -1 if behind and false if disjoint
641     * @todo rename isInFront
642     */
643    order (solid, axe) {
644      const res1 = this.isInFront(solid, axe)
645      if (res1) {
646        return res1
647      }
648      return solid.isInFront(this, axe)
649    }
650
651    /**
652     *
653     * @param {*} solid
654     * @returns bool
655     * @todo il faudrait aussi vérifier que !c2.find()
656     * @todo il faut vérifier que l'écart est faible !
657     */
658    isEqual (solid) {
659      const c1 = [...this.corners]
660      const c2 = [...solid.corners]
661      if (c1.length !== c2.length) return false
662      return c1.every(corner => c2.find(c => c === corner))
663    }
664
665    /**
666     * subtract this to the solid in parameter
667     * @param {*}
668     * @returns an array of solids representing the substraction
669     * @todo vérifier s'il ne faut pas une fonction qui soustrait des solides
670     * @todo évaluer l'impact du clone de faces
671     */
672    subtract (faces) {
673      const newsolid = this.clone()
674      newsolid.ensureFaces()
675      // TODO supprimer
```

```
676     const clonefaces = faces
677     const subsolids = clonefaces
678       .reduce((subsolids, face) => {
679         return [...subsolids, newsolid.sliceWith(face)]
680       }, [])
681       .filter(solid => solid.isNonEmptySolid())
682
683     // TODO pas sur utile de recalculer les éléments
684     subsolids.forEach(solidei => {
685       solidei.unvalidSolid()
686       solidei.ensureFaces()
687       solidei.ensureSilhouettes()
688     })
689     return subsolids
690   }
691
692   /**
693
694 /**
695  *
696  * @param {*} solid
697  * @todo verifier s'il ne faut pas retourner solid
698  */
699   clipWith (solid, axe) {
700     if (this.isEqual(solid)) return solid
701     if (this.order(solid, axe) === -1) return solid
702     // TODO verify with need to clip the other
703     const tempsub = solid.subtract(this.getSilhouette(axe))
704
705     return tempsub
706   }
707
708   /**
709    *
710    * @param {*} solids
711    * @param {*} exclude
712    * TODO: remplacer concat par un reduce ?
713    */
714   solidsSilhouetteSubtract (solids, axe) {
715     const soli = solids.map(sol => sol.clone())
716     let res = []
717     for (let i = 0; i < solids.length; i++) {
718       res = res.concat(this.clipWith(soli[i], axe))
719     }
720     return res
721   }
722
723   /**
724    * @todo write
725    */
726   middleOf () {
727     const corners = []
728     for (let i = 0; i < this.dimension; i++) {
```

```
729       const vals = [...this.corners].map(corner => corner[i])
730       const maxCorner = Math.max(...vals)
731       const minCorner = Math.min(...vals)
732       corners[i] = (maxCorner + minCorner) / 2
733     }
734     return corners
735   }
736
737   /**
738    *
739    * @param {*} face
740    * @todo simplif
741    */
742   static initSolidFromFace (face) {
743     const solid = new Solid(face.equ.length - 1)
744     solid.suffixFace(face)
745     solid.selected = face.selected
746     return solid
747   }
748
749   /**
750    *
751    * @param {*} halfspaces
752    */
753   static createSolidFromHalfSpaces (halfspaces) {
754     const solid = new Solid(halfspaces[0].length - 1)
755     halfspaces.forEach((HS, id) => {
756       const face = new Face(HS)
757       face.id = id
758       solid.suffixFace(face)
759     })
760     solid.ensureFaces()
761     return solid
762   }
763
764   static silProject (face, axe) {
765     // if (face.isBackFace()) {  return false }
766     // const newface = cloneDeep(face);
767     // newface.equ=projectVector(newface.equ,axe);
768     // newface.dim = newface.dim - 1 ;
769     // const nfaces = [...face.intersectionsIntoFaces()];
770     return projectVector(face.equ, axe)
771   }
772
773   static createSolidFromVertices (vertices) {
774     const dim = vertices[0].length
775     const listgroup = moizeAmongIndex(vertices.length, dim, dim)
776     let hyperplanes = listgroup.map(el => {
777       const points = el.map(idx => {
778         return [...vertices[idx]]
779       })
780       return points
781     })
```

```
    hyperplanes = hyperplanes.map( points => {
      return findnormal(points)
    }).filter(el => el)
    // on a maintenant dans hyperplanes l'ensemble des hyperplans induits par les différents points.
    // attention cependant, il manque ceux passant par l'origine.
    // il faut maintenant vérifier l'orientation de ces hyperplans
    hyperplanes = hyperplanes.map((hype,id) => {
      const posit = vertices.map(pt => positionPoint(hype,pt))
      if (posit.every(x => x >= 0)) { return hype }
      else if (posit.every(x => x <= 0)) { return hype.map(coord => -coord) }
      else { return false }
    })
    // on a maintenant tous les hyperplans générés par les points, on a donc beaucoup de doublons,
    // que l'on filtre
    hyperplanes = uniqBy(hyperplanes, JSON.stringify)

    const hpFiltered = hyperplanes.filter(el => el)
    // on a maintenant les hyperplans englobants, on en déduit les faces.
    const sol = this.createSolidFromHalfSpaces(hpFiltered)
    return sol
  }
}

export { Solid }

```