<table>
<thead>
<tr>
<th><strong>Projecte/Treball Fi de Carrera</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Estudi:</strong> Eng. Tècn. Informàtica de Sistemes. Pla 2001</td>
</tr>
<tr>
<td><strong>Títol:</strong> Eina pel desenvolupament de reflectors de conjunts òptics</td>
</tr>
<tr>
<td><strong>Document:</strong> Memòria</td>
</tr>
<tr>
<td><strong>Alumne:</strong> Jordi Molas Gaja</td>
</tr>
<tr>
<td><strong>Director/Tutor:</strong> Gustavo Patow</td>
</tr>
<tr>
<td><strong>Departament:</strong> Informàtica i Matemàtica Aplicada</td>
</tr>
<tr>
<td><strong>Àrea:</strong> LSI</td>
</tr>
<tr>
<td><strong>Convocatòria (mes/any):</strong> setembre/2007</td>
</tr>
</tbody>
</table>
ÍNDICE

1. INTRODUCCIÓ ........................................................................................................ 7
   1.1. Context ........................................................................................................ 7
   1.2. Objectiu: Reducció de costos de producció luminotècnica .................. 8
   1.3. Programari lliure .................................................................................. 8
   1.4. Eines de desenvolupament ................................................................... 8
      1.4.1. Blender .......................................................................................... 8
      1.4.2. Python ......................................................................................... 9
      1.4.3. Matlab ......................................................................................... 9
   1.5. Treball previ i temporització del projecte ........................................... 10

2. MODELAR EN BLENDER ...................................................................................... 15
   2.1. La seva història ....................................................................................... 15
   2.2. Blender a la indústria dels gràfics computacionals (CG) .................... 17
   2.3. Característiques .................................................................................... 18
   2.4. La interfície ............................................................................................. 18
      2.4.1. El teclat i el ratolí ........................................................................ 18
      2.4.2. El sistema de finestres .................................................................... 19
      2.4.3. Elements de la interfície ................................................................ 20
         2.4.3.1. El menú Split per crear finestres noves .............................. 20
         2.4.3.2. Tipus de finestres .................................................................. 21
      2.4.4. Transformacions bàsiques ............................................................. 22
         2.4.4.1. La selecció .............................................................................. 22
         2.4.4.2. Desplaçar un objecte ............................................................. 23
         2.4.4.3. Rotar un objecte .................................................................... 24
         2.4.4.4. Escalar un objecte .................................................................. 24
         2.4.4.5. Especificar amb exactitud desplaçaments, rotacions i escalats 24
         2.4.4.6. El centre de rotació ................................................................ 25
      2.4.5. Mode edició: objectes i vèrtexs ..................................................... 26
         2.4.5.1. Afegir objectes ....................................................................... 27
         2.4.5.2. Agrupació d’objectes ............................................................... 30
         2.4.5.3. Duplicar un objecte ................................................................. 31
      2.4.6. Operacions booleanes ...................................................................... 32
      2.4.7. Tracking ........................................................................................... 35
      2.4.8. Text ................................................................................................. 36
      2.4.9. Més operacions amb objectes ......................................................... 36
         2.4.9.1. Edició proporcional ................................................................ 37
         2.4.9.2. Suavitzat ................................................................................. 38
         2.4.9.3. Corbes i superfícies ................................................................. 38
         2.4.9.4. Creació de sólids de revolució ............................................... 40
      2.4.10. Screw ............................................................................................. 42
   2.5. Mòduls més rellevants .............................................................................. 43
      2.5.1. Mòdul Blender ............................................................................... 43
      2.5.2. Submòdul Mesh ........................................................................... 43
      2.5.3. Submòdul NMesh ......................................................................... 44
3. DISSENY ...................................................................................................... 57

3.1. Conceptes previs ................................................................................. 57
3.2. Classes auxiliars creades................................................................. 58
3.3. Diagrama de classes ....................................................................... 58
3.3.1. Classe Matriu ............................................................................... 59
3.4. Classe DXF ....................................................................................... 59
3.4.1. Format DXF ................................................................................ 59
3.4.2. Lectura del fitxer DXF ............................................................... 60
3.5. Desenvolupament de l’algorisme Graham’s Scan ....................... 62
3.5.1. L’envolvent convexa ................................................................ 63
3.5.2. Graham’s Scan ....................................................................... 64
3.5.3. Algorisme Graham’s Scan ......................................................... 65
3.5.4. Dibuixant els vèrtex per pantalla .............................................. 66
3.6 Creant la superfície del reflector ...................................................... 66
3.6.1. Creació dels punts interiors de la boca ................................... 66
3.6.2. El procés de la triangulació ...................................................... 67
3.7. Generació de volum del reflector .................................................... 74
3.8. Col·locació del llum sota la boca del reflector ............................. 75
3.9. Procés per crear el pla a il·luminar ............................................... 75
3.10. Procés de renderització ................................................................. 76
3.11. Verificació del resultat en Matlab ............................................... 78
3.11.1. Obtenció de les corbes de nivell .......................................... 79

4. IMPLEMENTACIÓ ...................................................................................... 83

4.1 Lectura del fitxer DXF ...................................................................... 83
4.2 Preparació de l’algorisme Graham’s Scan .................................... 84
4.3 Desenvolupament de la funció de Graham’s Scan ....................... 87
4.4 Omplir l’interior de la boca del reflector amb punts .................... 89
4.5 Realització de la triangulació ............................................................ 92
4.6 Creació del pla situat sota el reflector .......................................... 114
4.7 Creació de la càmera ..................................................................... 115
4.8 Realització del fitxer de render ...................................................... 117
4.9 Obtenció de les corbes de nivell utilitzant el Matlab ........................................ 120
4.10 Creació de la interfície ..................................................................................... 123
4.10.1 Utilització dels events ................................................................................. 125

5. RESULTATS I CONCLUSIONS ........................................................................... 131
5.1. Resultats obtinguts ......................................................................................... 131
5.2. Conclusions del projecte ................................................................................. 139

6. TREBALLS FUTURS I AGRAÏMENTS ............................................................. 143
6.1. Treballs Futurs ............................................................................................... 143
6.2. Agraïments ..................................................................................................... 143

QUADRE DE FIGURES I TAULES .......................................................................... 147

BIBLIOGRAFIA ........................................................................................................ 153

APÈNDIX 1. EL PROGRAMARI LLIURE .............................................................. 157
1. Història del programari lliure ........................................................................... 157
2. Llibertats del programari lliure ........................................................................ 157

APÈNDIX 2. LLENGUATGE DE PROGRAMACIÓ PYTHON ........................................ 159
1. Història de Python ............................................................................................ 159
2. Característiques ............................................................................................... 160
3. Tipus de dades .................................................................................................. 161
3.1. Enters i flotants ............................................................................................... 161
3.2. Valors lògics .................................................................................................. 162
4. Operadors lògics i de comparació ...................................................................... 162
5. Assignacions amb operador ............................................................................. 165
6. Variables no inicialitzades .................................................................................. 166
7. Tipus de dada cadena .......................................................................................... 166
8. Funcions predefinides ...................................................................................... 167
9. Funcions definides en mòduls .......................................................................... 168
9.1. El mòdul math ................................................................................................ 169
10. Mètodes ............................................................................................................. 169
11. Programes ......................................................................................................... 170
11.1. Lectura de dades des de teclat .................................................................... 170
11.2. Comentaris .................................................................................................... 171
12. Estructures de control ....................................................................................... 171
12.1. Sentències condicional .................................................................................. 171
12.1.1. Sentències condicionals IF ..................................................................... 171
12.1.2. Sentències condicionals imbricades ..................................................... 172
12.1.3. En cas contrari (else) ............................................................................. 172
12.1.4. Una forma compacta per a estructures condicionals múltiples (elif) ....... 172
12.2. Sentències iteratives ..................................................................................... 173
12.2.1. La sentència WHILE ............................................................................ 173
12.2.2. El bucle FOR-IN ................................................................................... 174
12.2.3. Ruptura de bucles: BREAK .............................................................. 174
12.2.4. Imbricament d’estructures .......................................................... 174
13. Captura i tractament d’excepcions ....................................................... 175
14. Tipus estructurats: seqüències ............................................................. 175
  14.1. Cadenes .................................................................................................. 176
    14.1.1. Longitud d’una cadena ............................................................... 176
    14.1.2. Indexació .................................................................................... 177
    14.1.3. Recorregut de cadenes ............................................................... 177
    14.1.4. Referència a cadenes ................................................................. 177
  14.2. Llistes .................................................................................................. 178
    14.2.1. Comparació de llistes................................................................. 179
    14.2.2. Afegir elements en una llista ...................................................... 180
    14.2.3.Eliminar elements d’una llista ..................................................... 180
  14.3. Passar de cadenes a llistes i a l’invers .............................................. 180
  14.4. Matrïus .................................................................................................. 181
    14.4.1. Creació de matrïus ................................................................. 181
    14.4.2. Lectura de matrïus ................................................................. 182
  14.5. Funcions .............................................................................................. 182
    14.5.1. Utilització de funcions .............................................................. 182
    14.5.2. Definició de funcions ............................................................... 183
      14.5.2.1. Definició i utilització de funcions amb un sol paràmetre .. 183
      14.5.2.2. Definició i utilització de funcions amb més d’un
                    paràmetre ....................................................................... 184
      14.5.2.3. Procediments: funcions sense retornar valor .......... 184
    14.5.3. Variables locals i variables globals ............................................ 184
    14.5.4. Accés a variables globals des de funcions ................................. 185
  15. Tipus estructurats: registres ................................................................. 185
    15.1. Associant dades relacionals ............................................................ 185
    15.2. Registres .......................................................................................... 186
    15.3. Definició de nous tipus de dades .................................................... 186
1. Introducció

1.1. Context

Avui en dia, una bona il·luminació vial és molt important, tant per conductors com per vianants. Els fanals que trobem al carrer (veure figura 1) han de il·luminar correctament la zona corresponent sense deixar parts del carrer sense il·luminació.

Trobar la forma exacte d’un reflector amb propietats òptiques en particular és de gran importància en el sector luminotècnic. En aquest sector la major part de l’esforç és utilitzat per a la construcció de maquetes on els càlculs són elevats i amb la possibilitat de malgastar un munt de temps si els models emprats no s’ajusten als resultats d’il·luminació que s’esperaven obtenir al terra o en una superfície del carrer. El fet de no aproximiar-se als resultats esperats comporta tornar a iniciar el procés construint una nova maqueta i així successivament. Aquest procés és extremadament costós, tant per l’elevat cost que suposa la construcció de la maqueta, com també, pel temps invertit, essent com a cicle de treball bastants mesos.

Figura 1.2. Conjunt òptic
Trobar la forma que ha de tenir el reflector és només una part del conjunt òptic anomenat així pel sector luminotècnic, i compost per una font de llum que és la làmpada, el reflector i la carcassa que els protegeix (veure figura 2).

1.2. Objectiu: reducció de costos de producció luminotècnica

Portar a terme el disseny i desenvolupament d'un sistema capaç d'ajudar a obtenir el reflector adequat, competint amb la metodologia tradicional emprada pot beneficiar el sector luminotècnic. D'aquesta manera, partint de la intensitat lumínica de la làmpada utilitzada, la reflectància del material, altura del reflector i la forma i dimensions de l'obertura d'aquest, el conjunt òptic obtingut proporcionarà una intensitat lumínica especificada. Si a més d’implementar un software que s’encarregui de realitzar tot aquest procés hi afegim que es tracta d’una eina de lluire distribució, és a dir, programari lliure, suposarà una millora en el temps de disseny i una disminució de costos en la producció de les lluminàries.

1.3. Programari lliure

Programari lliure (en anglès free software) és el programari que, un vegada obtingut, pot ser usat, copiat, estudiat, modificat i redistribuït lliurement. El programari lliure sol estar disponible gratuïtament, però no cal associar programari lliure a programari gratuït, o a preu del cost de la distribució a través d'altres mitjans; no obstant això no és obligatori que sigui així i, encara que conservi el seu caràcter de lliure, pot ser venut comercialment. Anàlogament, el programari gratis o gratuït (denominat usualment freeware) inclou en algunes ocasions el codi font; no obstant això, aquest tipus de programari no és lliure en el mateix sentit que el programari lliure, a no ser que es garanteixin els drets de modificació i redistribució de les dites versions modificades del programa.

No hem de confondre programari lliure amb programari de domini públic. Aquest últim és aquell pel que no cal sol·licitar cap llicència i els drets d'explotació del qual són per a tota la humanitat, perquè pertany a tots per igual. Qualsevol pot fer ús d'ell, sempre amb fins legals i consignant la seva autoria original. Aquest programari seria aquell autor del qual ho dóna a la humanitat o els drets d'autor de la qual han expirat. Si un autor condiciona el seu ús sota una llicència, per molt fèble que sigui, ja no és domini públic. En resum, el programari de domini públic és la pura definició de la llibertat d'utilització d'una propietat intel·lectual que té la humanitat perquè així ho ha decidit el seu autor o la llei després d'un termini comptat des de la mort d'aquest, habitualment 70 anys.

1.4. Eines de desenvolupament

1.4.1. Blender

Blender és programari lliure multiplataforma, dedicat especialment al modelatge i creació de gràfics tridimensionals. El programa va ser inicialment distribuït de forma gratuïta però sense el codi font, amb un manual disponible per a la venda. Actualment
és compatible amb totes les versions de Microsoft Windows, Linux, Solaris, FreeBSD, IRIX i MacOS X.

Té una molt peculiar interfície gràfica d'usuari, que es critica com poc intuïtiva, perquè no es basa en el sistema clàssic de finestres; però té a la vegada avantatges importants sobre aquestes, com la configuració personalitzada de la distribució dels menús i vistes de càmera.

1.4.2. Python

Python és un llenguatge de programació creat per Guido van Rossum l'any 1990. És comparamat habitualment amb TCL, Perl, Scheme, Java i Ruby. En l'actualitat Python es desenvolupa com un projecte de codi obert, administrat per la Python Software Foundation. L'última versió estable del llenguatge és actualment (juliol 2007) la 2.5.1.

Python és considerat com la oposició lleial a Perl, llenguatge amb el qual manté una rivalitat amistosa. Els usuaris de Python consideren que aquest llenguatge és molt més net i elegant per programar.

Python permet dividir el programa en mòduls reutilitzables des d'autres programes Python. Ve amb una gran col·lecció de mòduls estàndard que es poden utilitzar com a base dels programes (o com a exemples per començar a aprendre Python). També hi ha mòduls inclosos que proporcionen E/S de fitxers, crides al sistema, sockets i fins a interfeïces a GUI (interfeiça gràfica amb l'usuari) com Tk, GTK, Qt entre altres...

Python és un llenguatge interpretat, la qual cosa estalvia un temps considerable en el desenvolupament del programa, perquè no cal complir ni enllaçar. L'intèrpret es pot utilitzar de manera interactiva, la qual cosa facilita experimentar amb característiques del llenguatge, escriure programes d'un sol ús o provar funcions durant el desenvolupament del programa. També és una calculadora molt útil.

El nom del llenguatge prové de l'afecció del seu creador original, Guido Van Rossum, pels humoristes britànics Monty Python. El principal objectiu que persegueix aquest llenguatge és la facilitat, tant de lectura, com de disseny.

1.4.3. Matlab

MATLAB és l'abreviatura de Matrix Laboratory (laboratori de matrius). És un programa d'anàlisi numèrica creat per The MathWorks el 1984. Es troba disponible per a les plataformes Unix, Windows i Mac US X.

Es poden ampliar les seves capacitats amb Toolboxes, algunes d'elles estan destinades al processat digital de senyal, adquisició de dades, economia, intel·ligència artificial, lògica difusa... També compta amb altres eines com Simulink, que serveix per simular sistemes.

La primera versió va sorgir amb la idea d'emprar uns paquets de subrutines escrites en Fortran en els cursos d'Àlgebra lineal i anàlisi numèrica, sense necessitat d'escrivir programes en Fortran.
Usa un llenguatge de programació creat el 1970 per proporcionar un senzill accés al programari de matrius \textit{LINPACK} i \textit{EISPACK} sense haver d'usar Fortran. També té el seu propi compilador.

És un programari molt usat en universitats, centres d'investigació i per enginyers. En els últims anys ha inclòs moltes més capacitats, com la de programar directament processadors digitals de senyal, crear codi VHDL i altres.

\textbf{MATLAB} és un programa de càlcul numèric, orientat a matrius i vectors. Per tant des del principi cal pensar que tot el que es pretengui fer amb ell, serà molt més ràpido i efectiu si es pensa en termes de matrius i vectors.

1.5. Treball previ i temporització del projecte

El treball previ per portar a terme una eina de desenvolupament de reflectors de conjunts òptics es pot dividir en cinc fases:

1. \textbf{Estudi de les eines que permeten la construcció de cossos en 3 dimensions}. Les possibilitats de crear objectes 3D amb algun programari informàtic és ampli, però havia de ser un programari de baix cost i, que el mateix temps, permetés la seva manipulació mitjançant codi de programació entenedor; per aquest motiu vam escollir el Blender.

2. \textbf{Coneixement del modelatge en Blender}. Abans de poder construir qualsevol aplicació, era necessari conèixer l'entorn el qual volíem treballar. Així que va ésser una feina costosa a l'hora de documentar-nos amb llibres i tutorials per a poder aprendre a modelar objectes en 3D des de Blender (veure capítol 2, \textit{Modelar en Blender}).

3. \textbf{Coneixement de l'eina de desenvolupament de Python}. El codi de programació que reconeix Blender és el llenguatge Python, doncs vam haver de documentar-nos sobre les instruccions que utilitzava aquest llenguatge de programació per a portar a terme la nostra aplicació (veure Apèndix 2, \textit{Llenguatge de Programació Python}).

4. \textbf{Coneixement dels mòduls de Blender per aplicar Python}. Una vegada après com modelar en Blender i com programar en Python, vam haver d'endinsar-nos dins el codi de Blender i aprendre quins mòduls s’utilitzaven per crear vèrtexs, línies, càmeres, llums, etc. (veure capítol 2, \textit{Modelar en Blender}).

5. \textbf{Matlab}. La obtenció d’un bon estudi de les mostres que havíem obtingut de la nostra aplicació, era imprescindible per aconseguir una bona aproximació de la intensitat de llum que ofereix el reflector en la realitat, per tant era important escollir un bon programari que calculés i permetés fer anàlisi dels resultats (veure capítol 4, \textit{Implementació}).

Pel què fa a la temporalització del projecte, en el diagrama de la \textbf{Figura 3} es pot comprovar la durada aproximada de cada una de les fases més importants del projecte.

<table>
<thead>
<tr>
<th>Descripció de la tanda</th>
<th>Duració</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estudi de les eines de disseny</td>
<td>100 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modelatge en Blender</td>
<td>100 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programació en Python</td>
<td>80 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mòduls de Blender</td>
<td>200 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disseny de Reflectors</td>
<td>150 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementació</td>
<td>120 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proves</td>
<td>70 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentació</td>
<td>90 dies</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figura 1.3.** Temporització del projecte.

Les fases de la figura 3 són:

1. **Estudi de les eines de disseny.** Aquesta fase correspon a l’estudi realitzat en el treball previ sobre els diferents programaris de disseny en 3D.

2. **Modelatge en Blender.** Aquesta fase aporta l’aprenentatge de l’eina de disseny escollida anteriorment.

3. **Programació en Python.** Aprenentatge sobre la programació en Python, ja que no havíem programat mai en aquest llenguatge.

4. **Mòduls de Blender.** Etapa en què ens havíem de documentar sobre com programar Blender a través dels seus diferents mòduls.

5. **Disseny de Reflectors.** Etapa on era necessari dissenyar els reflectors en la nostra eina de disseny per tal d’obtenir uns resultats finals òptims.

6. **Implementació.** Etapa d’implementació de tots els aspectes dissenyats anteriorment.

7. **Proves.** Fase de proves on verificàvem l’absència d’errors en el codi i, per tant, el seu correct funcionament. Aquesta es va realitzar durant tot el projecte mitjançant trossos de codi.

8. **Documentació.** Fase on s’ha anat documentant tots els elements referents al projecte.
Eina pel desenvolupament de reflectors de conjunts òptics
2 MODELAR EN BLENDER
2. Modelar en Blender

2.1. La seva història

El 1988, Ton Roosendaal va co-fundar l'estudi d'animació Holandès NeoGeo. NeoGeo ràpidament es va convertir en l'estudi més gran d'animació 3D a Holanda i en una de les més destacades cases d'animació a Europa. NeoGeo va crear produccions que van ser premiades (European Corporate Video Awards de 1993 i 1995) per a grans clients corporatius tals com la companyia multinacional d'electrònica Philips. A NeoGeo, Ton va ser el responsable tant de la direcció artística com del desenvolupament intern del programari. Després d'una acurada deliberació, Ton va decidir que l'actual eina 3D utilitzada en l'estudi de NeoGeo era massa vella i voluminosa de mantenir i actualitzar i necessitava ser recrpta des del principi. El 1995, aquesta reescritura va començar i estava destinat a converdr-se en el programari de creació 3D que ara coneixem com Blender. Mentre NeoGeo continuava refinant i millorant Blender, Ton es va donar conta que Blender podria ser utilitzat com a una eina per a altres artistes fora de l'estudi NeoGeo.

El 1998, Ton va decidir crear una nova companyia anomenada Not a Number (NaN) derivada de NeoGeo per fomentar el mercat i desenvolupar Blender. En la base de NaN, estava el desig de crear i distribuir gratuïtament una suite de creació 3D compacta i multiplataforma. En aquest moment, això va ser un concepte revolucionari ja que la majoria dels programes comercials de modelatge costaven milers de dòlars. NaN esperava aconseguir una eina de modelatge i animació d'un nivell professional a l'abast del públic en general. El model de negoci de NaN consistia a proporcionar productes comercials i serveis al voltant de Blender. El 1999, NaN va asistir a la seva primera conferència en el Siggraph en un esforç encara major per promocionar Blender. La primera convenció del Siggraph per a Blender el 1999 va ser un autèntic èxit i va provocar un enorme interès tant de la premsa com dels assistents a la convenció. Blender va ser un gran èxit i es va confirmar el seu enorme potencial!

En vist del gran èxit del Siggraph, al principi de l'any 2000, NaN va aconseguir un finançament de 4,5 milions d'euros procedents d'uns inversors. Aquesta gran aportació de diners va permetre a NaN estendre ràpidament les seves operacions. Aviat NaN va arribar a tenir més de 50 treballadors treballant al voltant del món intentant millorar i promocionar Blender. A l'estiu del 2000, Blender 2.0 va ser publicat. Aquesta versió de Blender integrava un motor de jocs a la suite 3D. Al final del 2000, el nombre d'usuariis registrats al lloc web de NaN va sobrepassar els 250.000.

Desafortunadament, les ambicions i oportunitats de NaN no van coincidir amb les capacitats de la companyia ni amb la realitat del mercat de l'època. Aquest sobredimensionament de l'empresa va conduir a una reestructuració creant una companyia (NaN) mes petita i amb nous fons procedents dels inversors. Sis mesos més tard, el primer producte comercial de NaN, Blender Publisher va ser llançat. Aquest producte va ser dirigit a l'emergent mercat de mitjans interactius en 3D basats en entorns web. A causa de les decebadores vendes i al continu clima de dificultats econòmiques, els nous inversors van decidir donar per acabades les activitats de NaN. Això també incloua aturar el desenvolupament de Blender. Si bé existien clarament defectes en l'actual versió de Blender, amb una arquitectura interna del programari complexa, característiques inacabades i una GUI no gaire comù, la magnifica ajuda de la comunitat i els clients que havien comprat Blender Publisher en el passat va provocar
que Ton no pogués permetre que Blender desaparegués en l'oblit. Com reactivar una nova companyia amb un equip suficientment gran de desenvolupadors no era factible, al març de 2002, Ton Roosendaal va fundar l'organització no lucrativa Blender Foundation (Fundació Blender).

El primer objectiu de la Fundació Blender va ser trobar una manera de continuar el desenvolupament i la promoció de Blender com un projecte de codi obert basat en la comunitat d'usuaris. Al juliol de 2002, Ton va aconseguir obtenir dels inversors de NaN un "sí" perquè la Fundació Blender dugués a terme el seu pla que Blender fos codi obert. La campanya de *Liberad a Blender* havia d'obtenir 100.000 EUR perquè la Fundació pogués comprar els drets del codi font i els de propietat intel·lectual de Blender als inversors de NaN i, posteriorment, alliberar Blender a la comunitat de codi obert. Amb un entusiasta grup de voluntaris, entre els que es trobaven diversos ex-empleats de NaN, va ser llançada la campanya de *Liberad a Blender*. Per al delit i sorpresa de tot el món, la campanya va aconseguir l'objectiu de 100.000 EUR en tan sols 7 setmanes. El diumenge 13 d'octubre de 2002, Blender va ser alliberat al món sota els termes de la Llicència Pública General de GNU (GPL). El desenvolupament de Blender continua fins als nostres dies conduït per un equip de dedicats voluntaris procedents de diverses parts del món i liderats pel creador de Blender, Ton Roosendaal.

Història de Blender i full de ruta:

- 1.00 - gener de 1995 - Desenvolupament de Blender en l'estudi d'animació NeoGeo.
- 1.23 - gener de 1998 - Versió per a SGI (IrisGL) publicada en el web.
- 1.30 - abril de 1998 - Versió per a Linux i FreeBSD, es porta a OpenGL i a X.
- 1.3x - juny de 1998 - Creació de NaN.
- 1.4x - setembre de 1998 - Versió per a Sun i Linux Alpha publicada.
- 1.50 - novembre de 1998 - Primer manual publicat.
- 1.60 - abril de 1999 - C-key (noves característiques necessiten ser desbloquejades, $95), la versió de Windows s'allibera.
- 1.6x - juny de 1999 - Versió per a BeOS i PPC publicada.
- 1.80 - juny de 2000 - Fi de la C-key, Blender és totalment gratuït novament.
- 2.00 - agost de 2000 - Motor en temps real i reproductor interactiu.
- 2.20 - agost de 2001 - Sistema d'animació de personatges.
- 2.21 - octubre de 2001 - Blender Publisher llançat.
- 2.2x - desembre de 2001 - Versió per a Mac OSX publicada.
- 13 d'octubre de 2002 - Blender es converteix en codi obert, primera Conferència de Blender.
- 2.25 - octubre de 2002 - Blender Publisher està novament disponible gratuïtament.
- 1 - octubre de 2002 - Es creuï el branca experimental de Blender, un lloc de proves per als programadors.
- 2.26 - febrer de 2003 - La primera versió de Blender sent codi obert.
- 2.27 - maig de 2003 - La segona versió de Blender sent codi obert.
- 2.28x - juliol de 2003 - La primera de les sèries 2.28x.
2.30 - octubre de 2003 - En la segona conferència de Blender, la interfície 2.3x és presentada.

2.31 - desembre de 2003 - Actualització a la interfície de la versió 2.3x, ja que és estable.

2.32 - gener de 2004 - Gran revisió de la capacitat del render intern.

2.33 - abril de 2004 - Oclusió Ambiental, nous procediments de Textures, el motor de joc ha tornat!

2.34 - agost de 2004 - Grans millores: Interaccions de Partícules, mapejat LSCM UV, integració funcional de YafRay, Plecs compensats en Subdivisió de Superfícies, Ombreig d'Inclinació (Ramp), OSSA complet i moltes més.

2.35 - novembre de 2004 - Una altra versió plena de millores.

2.36 - febrer de 2005 - Versió més estable, més treball en l'escena, poques noves millores, mapejat Normal i de Desplaçament, principalment.


2.40 - desembre de 2005 - Un major avanç: reescriptura del sistema d'esquelets, tecles de formes, pell amb partícules, fluids i cossos rígids.

2.41 - gener de 2006 - Principalment afegits al Motor de joc i correccions.

2.2. Blender a la indústria dels gràfics computacionals (CG)

Encara sent una eina relativament nova, ha gaudit de l'acceptació de molts animadors independents. En la indústria de Generació de gràfics avança com un projecte prometedor, si bé les superproduccions no ho han usat per generar seqüències CGI (Imatge generada per ordinador). Hi ha projectes actuals que han començat a usar-lo professionalment:

- Actualment s'està duent a terme la producció del primer llargmetratge animat realitzat integrament amb Programari Llindre, usant a Blender com principal eina; Es tracta de Plumifers, projecte que està impulsant el desenvolupament de Blender encara més, sobretot a nivell d'animació i maneig de llibreries a gran escala. S'espera l'estrena de Plumifers per a finals de 2007.

- Pel·lícules com Spiderman 2 que ho ha usat per fer una previsualització d'escenes (Screen-Board Test), han fet servir les capacitats del popular programa GNU/GPL.

- Algunes propostes més portades a la producció i integració amb gràfics mitjançant Motion Track tals com Friday or another day que és dels primers esbossos del seu ús a 35mm.

- Altres projectes fets en participació de diversos usuaris de Blender inclòs Ton Rosendaal, el curtmetratge Elephants Dream 2 són experiments de les seves capacitats, esteses gràcies a la possibilitat de poder editar el seu codi font, aportant d'aquesta experiència a la resta d'usuaris amb innovacions fonamentals: Un sistema de control de gestos (Morph system), un sistema de composició de textura i post producció (Composite), entre altres.
• Té un llarg camí per recórrer, Blender 3d té totes les bases que garanteixen el seu desenvolupament, en un futur no gaire llunyà, dins els àmbits professionals en l’ús de programari lliure.

2.3. Característiques

Multiplataforma, lliure, gratuït i amb una grandària d'origen realment petita comparat amb altres paquets de 3D, depenen del sistema operatiu en què s'executa.

• Capacitat per a una gran varietat de primitives geomètriques, incloent corbes, malles poligonals, buits, NURBS, metaballs.

• Al costat de les eines d'animació s'inclouen cinemàtica inversa, deformacions per armadura o quadrícula, vèrtexs de càrrega i partícules estàtiques i dinàmiques.

• Edició d'Àudio i sincronització de vídeo.

• Característiques interactives per a jocs com a detecció de col·lisions, recreacions dinàmiques i lògica.

• Possibilitats de renderització interna versàtil i integració externa amb potents traçadors de raïos o raytracer lliures com kerkythea o YafRay.

• Llenguatge Python per automatitzar o controlar diverses tasques.

• Blender accepta formats gràfics com TGA, JPG, Iris, SGI, o TIFF. També pot llegir fitxers Inventor.

• Motor de jocs 3D integrat, amb un sistema de maons lògics. Per a més control s'usa programació en llenguatge Python.

• Simulacions dinàmiques per a cossos tous, partícules i fluids.

• Modificadors apilables, per a l'aplicació de transformació no destructiva sobre malles.

• Sistema de partícules estàtiques per simular cabells i pelatges, a què s'han agregat noves propietats per aconseguir textures realistes.

2.4. La interfície

La interfície d'usuari és el mecanisme d'interacció mútua entre l'usuari i el programa. L'usuari es comunica amb el programa mitjançant el teclat i el ratolí, el programa respon per mitjà del que mostra en pantalla.

2.4.1. El teclat i el ratolí

La interfície de Blender treu profit dels ratolins de tres botons i una àmplia gamma de dreceres de teclat. Si el ratolí només té dos botons, és possible emular el botó central. És possible usar un ratolí amb roda, però no és obligatori, ja que també hi ha dreceres de teclat que compleixen la mateixa funció.
Resum de tecles

Nomenclatura
- **BER**: click en el Botó Esquerra del ratoli
- **BDR**: click en el Botó dret del ratoli
- **KP1, KP2...**: tecles 1, 2... del teclat numèric

Vistes
- **KP1**: Vista frontal
- **KP3**: Vista lateral
- **KP7**: Vista superior
- **KP5**: Modes Perspectiva / Ortogràfic

Desplaçaments
- **BER**: Situa el cursor
- **Alt+BER+Arrastrar**: Canvia el punt de vista
- **Ctrl+Alt+BER+Arrastrar**: Zoom
- **Shift+Alt+BER+Arrastrar**: Pa

Altres
- **Ctrl+X**: Començar de zero

2.4.2. El sistema de finestres

La Figura 2.1 mostra la vista de pantalla que ha d'obtenir-se en haver iniciat Blender (excepte per les fletxes i el text afegit). Per defecte està separada en tres finestres: el menú principal a la part alta, la vista 3D gran i la Finestra de Botons a sota. La majoria de les finestres tenen un encapçalament (la franja amb un fons gris clar que
conté botons amb icones - per a això se l’anomena, també, Barra d’Eines); si està present, l’encapçalament pot ubicar-se a la part alta (com la Finestra de Botons) o a la part baixa (com la vista 3D) de l’àrea d’una finestra.

Si es mou el ratolí sobre una finestra, és possible notar que el seu encapçalament canvia a un gris més clar. Això significa que aquesta està enfocada; totes les dreceres de teclat que es pressionin afectaran ara el contingut d’aquesta finestra.

És possible personalitzar el sistema de finestres de Blender per satisfacer les necessitats i gustos de l’usuari. Una finestra nova pot ser creada fraccionant una existent. Això es fa enfocant la finestra que es vol fraccionar (posant el punter del ratolí dins d’aquesta), fer click en el contorn amb BCR (Botó Central del ratolí) o BDR (Botó dret del ratolí) i seleccionar Split Area (Figura 3-2). Ara és possible ubicar el nou contorn fent click amb BER (botó esquerra del ratolí), o cancel·lar el fraccionament pressionant la tecla ESC.

La nova finestra serà una còpia de la finestra que va ser fraccionada, però ara pot ser convertida a una finestra d’un altre tipus, o fer que mostri l’escena des d’un altre punt de vista.

2.4.3. Elements de la interfície

Les etiquetes en els botons de la interfície, els elements de menú i, en general, qualsevol text mostrat en la pantalla ho hem remarcat de la següent manera:

2.4.3.1. El menú Split per crear finestres noves

Un nou contorn vertical és creat en escollir Split Area (veure figura 2.2) en un contorn horitzontal i viceversa. Cada finestra pot ser redimensionada arrossegant un contorn amb el BER. Per reduir el nombre de finestres, es fa click en un contorn entre dues finestres amb BCR o BDR i s’escull Join Areas. La finestra resultant rep les propietats de la finestra enfocada prèviament.

Per ajustar la posició d’un encapçalament es fa click amb el BDR sobre l’encapçalament i s’escull Top o Bottom. També és possible ocultar l’encapçalament seleccionant No Header, però aquesta opció només es recomana si totes les dreceres de teclat relatius es coneixen. Un encapçalament ocult es pot tornar a mostrar fent click en el contorn de la finestra amb BCR o BDR i seleccionant Add Header.
2.4.3.2. Tipus de finestres

Quan inicialitzem el programa, la vista per defecte ens mostra la vista 3D a la finestra superior i la finestra de botons a la part inferior. Es pot seleccionar la finestra que ens mostra Blender a cada espai mitjançant el botó **Tipus de finestra**.

![Figura 2.3. Menú de la finestra de botons](image)

A la finestra de botons (veure **Figura 2.3**) es poden seleccionar diferents botoneres:

- Lògica
- Script
- Sombrejat
- Objecte
- Edició
- Escena

Dins de sombrejat se’ns apareixen 5 opcions
- Làmpades
- Materials
- Textures
- Radiositat
- Món

Dins de **escena** se’ns apareixen tres opcions
- Render (vista per defecte)
- Animació i reproducció
- Bloc de so

Cadascun dels panells de les diferents botoneres pot minimitzar-se usant el triangle blanc que hi ha al costat del nom del panell. A més a més, els panells poden disposar-se de manera lliure (podrem desplaçar els botons com desitgem), horitzontal...
(és a dir, un panell al costat de l'altre) o vertical (un sobre l'altre, disposició adequada per a finestres d'aspecte vertical).

Per seleccionar la disposició només cal fer BDR (Botó dret del ratolí) en qualsevol lloc de la botonera i ens apareixerà un menú com el de la figura 2.4.

\[
\begin{array}{|c|c|c|c|}
\hline
& \text{Align buttons} & \\
\hline
& \text{Free} & \\
& \text{Horizontal} & \\
& \text{Vertical} & \\
\hline
\end{array}
\]

**Figura 2.4.** Menú emergent

En cas que necessitem treballar amb comoditat amb una finestra, podem fer que passi a ocupar tota la pantalla mitjançant la combinació de tecles Ctrl+CursorAbaix (polsarem Ctrl+CursorAmunt per tornar a l'estat inicial).

En la finestra 3D podrem seleccionar el tipus d'ombreig utilitzat per a l'escena amb el botó corresponent (veure figura 2.5).

Les opcions són, textura, sombrejat, sòlid, filferro o caixa. Les corresponents tecles de accés ràpid són Z, Shift+Z, Ctrl+Z, Alt+Z.

**Figura 2.5.** Menú de tipus d'ombreig.

2.4.4. Transformacions bàsiques

En aquesta secció es veuran les transformacions bàsiques: selecció, moviment, rotació i escalat d'objectes. Aquestes són les transformacions que ens permeten, a partir d'un conjunt limitat d'objectes bàsics, construir un món complex.

Es tracta d'operacions que es repetiran a dotzenes cada vegada que es realitzi un model, pel que és de vital importància dominar-les per poder usar el programa de manera eficient.

2.4.4.1. La selecció

Si Blender no està en el seu estat inicial, es reiniciará (polsant Ctrl+X). En la situació inicial apareix un cub de color rosa, que vol dir que està seleccionat. Per tant, se li poden aplicar diferents accions, com moure'l o rotar-lo.
Què cal fer per seleccionar un altre objecte (la càmera, per exemple)?

Cal situar el cursor sobre l'objecte a seleccionar i fer **BDR**. En seleccionar la càmera, s'obté la imatge de la figura 2.6. La càmera està seleccionada (i per tant de color rosa) i la galleda no (i per tant de color negre).

Si es fa **BDR** novament sobre la galleda es torna a la posició inicial, amb la galleda seleccionat (i per tant de color rosa) i la càmera no (i per tant de color negre).

### 2.4.4.2. Desplaçar un objecte

Amb la càmera seleccionada novament, per desplaçar-la per la vista de planta, es polsa la tecla **G** (inicial de grab), amb la qual cosa prendrà el color blanc (veure figura 2.7). En arrossegar el ratolí, l'objecte seleccionat segueix al cursor.

Per deixar anar l'objecte, cal fer **BER**. Si es vol cancel·lar l'acció, cal polsar **BDR** o **ESC**. És aconsellable, abans d'iniciar una acció, que el cursor estigui a prop de l'objecte seleccionat. Cal tenir en compte que el moviment que podem realitzar queda determinat per la vista que tinguem en pantalla.
2.4.4.3. Rotar un objecte

Sempre amb la càmera seleccionada, es polsa la tecla R (de Rotació) per rotar-la en la direcció desitjada. Novament, la rotació dependrà de la vista seleccionada.

La vista de càmera s'activa amb KP0. Així, si tenim un punt de vista com l'actual, obtindrem una vista com la de la figura 2.8.

2.4.4.4. Escalar un objecte

En la vista que tenim seleccionada, seleccionem el cub que hi ha en pantalla (amb BDR), polsem la tecla S (de Scale) i, desplaçant el ratolí li canviarem la grandària. Sense activar res més, canviem la grandària de l'objecte de manera uniforme: tant la seva amplada com la seva altura i la seva profunditat. Sovint voldrem que una operació es limiti només a un eix. Per aconseguir-ho podem actuar de diverses formes.

La primera és restringir l'acció a un sol eix amb la tecla Alt. Per exemple, seleccionem un objecte (G), polsem Alt+BER i desplacem el ratolí horitzontalment. Per pantalla apareixerà una guia de color verd, i només es desplaçarà l'objecte al llarg de la guia. Per a finalitzar l'acció polsem novament BER. El mateix passarà si el primer moviment que fem és en vertical.

Una segona forma, més eficient, és, després de seleccionar l'acció polsar X, I o Z, en funció de l'eix que vulguem restringir l'acció. I una tercera forma, si es desitja, és fer BCR.

2.4.4.5. Especificar amb exactitud desplaçaments, rotacions i escalats

Hi ha dos modificadors a l'hora de fer transformacions amb el ratolí. La tecla Majus. fa que el moviment sigui més precís, mentre que la tecla Ctrl realitza una modificació del tipus snap to grid, o ajustar-se a la reixeta.
Una segona possibilitat, més precisa, és la d'entrar numèricament les magnituds a operar. Per a això, un vegada seleccionat l'objecte que desitgem, polsem la tecla N, amb la qual cosa obtenim la finestra de la figura 2.9.

Figura 2.9. Finestra de propietats

En aquesta finestra podem especificar exactament les unitats en què desitgem desplaçar (Loc), rotar (Rot) o escalar (Size) l'objecte respecte a cadascun dels tres eixos (X, Y, Z).

Per modificar cadascuna les cel·les podem fer dues coses:
• polsar BER en la cel·la i arrossegar el ratolí;
• polsar Majus+BER sobre la cel·la, amb la qual cosa apareixerà un cursor i podrem introduir la magnitud exacta.

2.4.4.6. El centre de rotació

En fer una rotació és imprescindible especificar al voltant de quin punt es va a fer la rotació. Aquest punt l’anomenem centre.

Per especificar-ho s'utilitza el menú que tenim a la imatge. La figura 2.10 ens mostra les quatre opcions disponibles.

Figura 2.10. Menú de rotació

• **Individual Object Centers**: En cas que tinguem més d'un objecte seleccionat, cadascú girarà al voltant del seu propi centre.
• **3D Cursor**: La rotació es produirà al voltant del cursor.
• **Median Point**: La rotació es produirà al voltant del centre de gravetat dels objectes seleccionats.
• **Bounding Box Center**: Blender calcularà la menor caixa possible que conté als objectes seleccionats, i els farà girar al voltant d’aquesta caixa.
2.4.5. Mode edició: objectes i vèrtexs

També podem accedir a cadascun dels vèrtexs de l'objecte. Per poder-ho fer pòsum la tecla TAB. Si tenim un objecte seleccionat (i per tant de color rosa) i pòsum TAB quedaran en rosa només els seus vèrtexs, tal i com veiem a la \textit{figura 2.11}.

\begin{center}
\includegraphics[width=0.4\textwidth]{figura211.png}
\end{center}

\textit{Figura 2.11.} Mode edició

Pòsumant \textbf{TAB} novament tornariem a seleccionar tot l'objecte.

En mode d'edició de vèrtexs, el color rosa no indica que els vèrtexs estiguin seleccionats. Per seleccionar tots els vèrtexs, pòsum la tecla \textbf{A}. Els vèrtexs queden seleccionats i ressaltats en groc.

Per seleccionar un sol vèrtex, hem de pòsar \textbf{BDR} sobre el vèrtex desitjat (veure \textit{figura 2.12}).

\begin{center}
\includegraphics[width=0.4\textwidth]{figura212.png}
\end{center}

\textit{Figura 2.12.} Selecció de vèrtexs.
Si volem seleccionar més d'un vèrtex, seleccionem el primer (BDR), polsem la tecla Majus, i anem fent BDR en la resta de vèrtexs que vulguem seleccionar.

Finalment, si desitgem seleccionar tots els vèrtexs dins una zona rectangular, polsem B, amb la qual cosa activem la selecció rectangular. Polsem BER en un extrem de l'àrea desitjada, i amb el botó premut, el deixem anar a l'extrem oposat. Es seleccionaran els vèrtexs dins el rectangle seleccionat.

2.4.5.1. Afegir objectes

En aquest mòdul mostrem, finalment, com afegir nous objectes a una escena. Començarem mostrant com afegir objectes de tipus 'malla poligonal', amb els que podrem construir models complexos. Mostrem les diferents malles que ofereix Blender, i les característiques particulars de cadascuna. Veurem les diferents accions que podem executar sobre les malles poligonals, com l'agrupació, duplicació, operacions booleanes o el tracking. Finalment, veurem com afegir text a l'escena.

Amb la tecla ESPAI accedim a la Caixa d'eines, que apareix de dues manera diferents, depenent de si estem en el mode d'edició de punts (veure figura 2.13) o d’objectes (veure figura 2.14).
Capítol 2. Modelar en Blender

Per defecte, la caixa d'eines s'activa amb l'opció Add (Afegir), des de la que podem afegir a la nostra escena tot tipus d'objectes. Ens assegurarem d'estar en mode d'edició d'objectes, i en primer lloc ens centrarem en la opció Mesh (malla poligonal), que disposa de les subopcions que apareixen a la figura 2.15.

- Pla
- Cub
- Cercle (amb un número seleccionable de punts).
- UV Esfera (amb un número seleccionable de segments i anelles).


Figura 2.15. Submenú Mesh
Eina pel desenvolupament de reflectors de conjunts òptics

- **Iosfera** (amb un número variable de subdivisions).
- **Cilindre** (amb un número variable de vèrtexs).
- **Tub** (amb un número variable de vèrtexs).
- **Con** (amb un número variable de vèrtexs).
- **Graella** (amb resolucions variables pels eixos X i Y).
- **Mono** (crea una imatge que s’associa com la mascota de Blender)

Quan seleccionem la UVesfera (UVSphere) ens apareixen dos requadres de diàleg consecutius (veure **figura 2.16**) que ens demanen el número de segments i el de les anelles amb què volem construir l’esfera.

![Figura 2.16. Quadres de diàleg de UVSphere](image)

Per moure'ns al quadre de diàleg, podem fer **BER** i arrossegar el ratolí a dreta i esquerra o bé utilitzar les dues fletxes que hi ha a cada costat. Una tercera opció, més exacta, és usar el teclat, fent prèviament **Majús.+BER** sobre el quadre.

![Figura 2.17. Selecció de tots els vèrtexs de l’esfera. (Mode edició de vèrtexs).](image)

L’esfera apareix en mode edició de vèrtexs (veure **figura 2.17**), amb tots ells seleccionats. Polsant TAB passem del mode d’edició de vèrtexs al d’edició d’objectes (veure **figura 2.18**).
2.4.5.2. Agrupació d’objectes

Sovint ens interessarà treballar amb diversos objectes com si es tractés d’un sol objecte. Per agrupar tants objectes com es desitgi, primerament, hem de pensar en quin objecte farà d’objecte pare (veure figura 2.19).

Una vegada fet això, seleccionem tots els objectes a unir, i fem que estigui actiu l’objecte pare. Polsem Majus.+G i acceptem el quadre de diàleg que apareix (veure figura 2.20). A partir d’ara totes les accions afectaran per igual a tot el grup. La relació es trenca quan polsem Alt+P.

Si polsem Majus.+G accedirem a un menú, com el que veiem a la figura 2.21, des del que podrem seleccionar:
2.4.5.3. Duplicar un objecte

Si tenim un objecte seleccionat com el de la figura 2.22, per duplicar-ho n'hi ha prou amb pulsar Majus.+D. Amb això es crearà una còpia exacta, situada exactament en la mateixa posició. Per portar el nou objecte a una posició diferent, només cal desplaçar el ratolí i, una vegada aconseguida la ubicació desitjada, fer BER (veure figura 2.23).

Una segona opció és crear un duplicat enllaçat com el de la figura 2.24. En aquest cas crearem un segon objecte que conservarà sempre les propietats del primer. Això és, si canviem qualsevol propietat del primer objecte, els canvis també afectaran el segon, si es fan en el mode d'edició. Per crear un duplicat enllaçat s'usa Alt+D.
2.4.6. Operacions booleanes

Una manera d'operar amb malles és usar operacions booleanes.

Per començar una operació booleana has de seleccionar els dos objectes a operar i pulsar W.

Una bona manera de veure com s'utilitzen les operacions booleanes és veure’n un exemple:

Construirem un cendrer (veure figura 2.25). Usem un galleda deformada com a cos del cendrer. Li restarem una esfera per construir el bol en què cau la cendra, i després quatre cilindres per fer les peces sobre les quals deixarem un cigar.

Col·locarem l'esfera sobre el cub, de la forma que veiem a la figura 2.26. Seleccionem ambdós objectes (primer el cub i després l'esfera, ja que volem agafar el cub i restar-li l'esfera), polsem W i seleccionem la diferència dels dos objectes.
Eina pel desenvolupament de reflectors de conjunts òptics

Amb això crearem un objecte com el que veiem a la figura 2.27. Només falta restar-li quatre vegades el cilindre per construir el cendrer.

Una vegada vist el resultat de la sostracció, faltaria veure les altres dos operacions booleanes. Com la intersecció de dos objectes el resultat serà l’espai en què coincidixen els dos (veure figura 2.28).

En el mode edició, quan polsem la tecla W s’activa el menú Specials (veure figura 2.29), que ens permet realitzar la següent sèrie d’accions:
Subdivide: divideix cada una de les cares seleccionades
Subdivide fractal: també divideix cada una de les cares, però desplaça aleatòriament cada un dels nous vèrtexs creats, amb un grau d’aleatorietat seleccionat per l’usuari.

Merge: uneix tots els punts seleccionats en un sol punt, que es pot situar en el centre dels punts seleccionats o bé en el punt en que es troba el cursor.

Remove Doubles intenta eliminar els vèrtexs duplicats d'entre els seleccionats. Per a això calcula les distàncies entre vèrtexs, i elimina aquells que es troben a menys de 0.0001 unitats entre si.
Hide oculta els vèrtexs seleccionats. Amb Reveal es tornen a mostrar tots els vèrtexs ocultats anteriorment.
Select swap inverteix la selecció: els vèrtexs seleccionats passen a no estar-ho, i viceversa.
Flip Normals inverteix les normals de cadascuna de les arestes seleccionades (no té un efecte obvi).
Smooth intenta suavitzar la malla, desplaçant lleugerament cada vèrtex cap al baricentre dels vèrtexs seleccionats (en el cas de l'esfera que il·lustra aquesta pàgina, l'únic efecte aparent seria la disminució de grandària de la icosfera).
Eina pel desenvolupament de reflectors de conjunts òptics

- **Bevel** és l'últim mètode per afegir vèrtexs i suavitzar una malla. Ens demana un nivell de recursivitat i permet controlar la disposició dels vèrtexs afegits arrossegant el ratoli.

![Figura 2.30. Selecció de vèrtexs.](image)

2.4.7. Tracking

Sovint ens interessarà que un objecte apunti continuament a un altre, i no tenir que actualitzar la direcció en què mira cada vegada que movem el segon objecte. En particular, és interessant que la càmera apunti a l'objecte principal de l'escena, sense necessitat d'anar-la girant.

Això s'aconsegueix amb el 'tracking' (veure figura 2.31).

![Figura 2.31. Menú de Tracking.](image)

Per aconseguir-ho seleccionarem primer l'objecte que ha d'apuntar a l'altre, i després el que ha de seguir i polsarem Ctrl+T. Per trencar el seguiment es polsa Alt+T.

Pot especificar-se els eixos en què es fa el seguiment des del quadre Anim Settings del panell Object que mostrem en la figura 2.32.

![Figura 2.32. Panell Anim Settings](image)
2.4.8. Text

Un altre recurs per afegir objectes a una escena és l'ús de text (Espai > Add > Text). Podem substituir el text per defecte de Blender pel que desitgem. Al panell d'edició apareixerà un quadre Font, com el que mostrem a la figura 2.33, des del que podrem controlar diversos aspectes del text, com són l'alineació (Left, Middle, Right i Flush), la grandària (Size), l'espaiat (Spacing), la inclinació (Shear), l'interlineat (Linedist), i la posició (X offset i Y offset).

![Figura 2.33. Panell de text.](image)

A més a més, poden seleccionar-se tipus de lletra diferents mitjançant el botó Load. Blender pot utilitzar fonts PostScript Type 1 i True Type (com les que utilitza Windows). Com amb la resta de corbes i superfícies, és convenient usar l'extrusió per donar volum al text creat.

Finalment, per obtenir un major control sobre el text, pot convertir-se el text en corbes de tipus Bézier mitjançant (ALT+C). Un vegada convertit el text en una corba ja no podrem editar-ho des del teclat.

2.4.9. Més operacions amb objectes

Altres operacions per a la creació o milloració d’objectes és l'edició proporcional, que ens permetrà editar malles complexes de manera més suau i natural.

A continuació veurem el suavitzat, que ens permetrà donar un aspecte més suau i natural, i menys 'informàtic' als objectes creats prèviament, encara que aquests no presentin una gran complexitat en el disseny original.

A continuació mostrarem com crear objectes a partir de corbes i superfícies Bézier, fent ús de l'extrusió.

Finalment, veurem com crear sòlids de revolució a partir de corbes.
2.4.9.1. Edició proporcional

Si treballem amb una malla complexa, com la de la figura 2.34 (obtinguda a partir d'un pla i unes quantes subdivisions fractals), resulta complicat fer alteracions en la posició, escala o rotació d'un grup de punts sense alterar de manera brusca la posició de les cares adjacents.

Per suavitzar aquest efecte, es polsa la tecla O, amb la qual cosa a la capçalera de la finestra apareix la icona:

![Icona de mode proporcional](image)

Aquesta icona indica que s'ha entrat en mode proporcional. Ara, en desplaçar un punt o grup de punts, aquest desplaçament afectarà lleument als vèrtexs confrontats, amb la qual cosa es suavitza l'efecte produït.

Es pot escollir entre un efecte més suau (Smooth Falloff) o més brusc (Sharp Falloff), polsant Majus.+O o usant el menú.

El radi d'influència del suavitzat pot incrementar-se o decrementar-se polsant + i - en el teclat numèric o amb la roda central del ratolí.
2.4.9.2. Suavitzat

En crear objectes en Blender, l'aspecte per defecte és 'de gràfic per ordinador': una sèrie de cares planes i arestes marcades. Això pot minimitzar-se activant les capacitats de suavitzat.

Per fer-ho accedirem al panell d'edició (veure figura 2.35) i, en el quadre Link and Materials, fem click en el botó Set Smooth. Podem tornar a l’estil original amb el botó Set Solid.

És possible seleccionar quines cares volem que apareguin en cadascun dels dos estils utilitzant els botons des del mode edició, seleccionant prèviament les cares que desitgem que apareguin amb cada estil.

Una segona opció és utilitzar el botó Auto Smooth del quadre Mesh (veure figura 2.36). Amb aquesta opció seleccionada, Blender calcula els angles entre cares contigües i, si l'angle és suficientment baix, aplica el suavitzat automàticament. L'angle a partir del qual s'aplica el suavitzat pot controlar-se mitjançant el botó Degr. El suavitzat automàtic s'aplica a aquelles cares a què se'ls ha fet un Set Smooth amb anterioritat.

2.4.9.3. Corbes i superfícies

Les corbes bézier són corbes que permeten una gran flexibilitat en el disseny. Es pot afegir una corba, polsant espai, i seleccionant Add > Curve > Bezier Curve. En el dibuix ens apareix una corba amb sis punts, distribuïts en dos segments de color rosa (veure figura 2.37).
El punt central del segment determina l'extrem corresponent de la corba, mentre que els altres dos determinen la seva corbatura. És convenient 'jugar' amb tots els controls fins a adquirir certa flexibilitat en el seu maneig.

A part de desplaçar-se els sis punts de control, poden realitzar-se les accions següents:
- Amb Ctrl+BER afegim un nou punt a la corba, a partir de l'extrem que tinguem seleccionat.
- Si seleccionem dos punts consecutius, podem afegir un punt intermig amb W > Subdivide.
- Es pot eliminar el punt seleccionat amb X (o Supr) > Selected.
- En cas que desitgem que la unió de les dues corbes sigui brusques, i no suaus, seleccionarem el punt desitjat i pòlsarem V.
- Per tancar la corba pòlsarem C.
- Utilitzant aquest procediment pot dibuixar-se una figura com la següent:

Si passem al mode d'objectes i pòlsarem Z podrem veure que hem definit una superfície.

Una altra possibilitat de què disposem és la de crear forats en la nova superfície. Per fer-ho, seleccionem Add > Curve > Bezier Circle.

Les circumferències de Bezier es modelen de la mateixa manera que les corbes.
Capítol 2. Modelar en Blender

Per tenir un objecte utilitzable, ens falta donar-li volum al conjunt. Per això hem de fer una extrusió. Per fer-ho, tenim que anar al panell d’edició (F9) com el que veiem a la figura 2.40.

Els controls que ens interessen són:

- **Width** determina la grossor de l'extrusió. No és convenient usar valors alts. El deixarem al valor per defecte, 1.
- **Ext1** determina l’altura que guanyarà l’objecte. Podem usar un valor de 0.4.
- **Ext2** determina la grandària del bisellat. Usarem un valor de 0.09.
- Finalment, **BevResol** determina la resolució del bisellat. Li donarem el valor de 1.

2.4.9.4. Creació de sòlids de revolució

Es pot crear una copa utilitzant la tècnica de rotar una corba al voltant d’un eix (veure figura 2.41). En primer lloc, en mode edició dibuixem el perfil de la copa (només cal anar afegint punts i les corresponents arestes). Com si del seu equivalent en la vida real es tractés, és convenient que el vidre tingui un cert grossor.
Eina pel desenvolupament de reflectors de conjunts òptics

En el panell d’edició, que veiem a la figura 2.42, tenim el quadre Mesh Tools (o eines per a malles), on tenim el botó Spin, que ens permetrà rotar el perfil de la copa 360 graus per construir-la.

El botó Degr especifica els graus que rotarem l'objecte (i per tant hem de deixarlo a 360 graus), mentre que Steps decideix quantes aturades es faran en la rotació. La rotació es produirà al voltant del cursor, i l'eix quedarà determinat per la vista en què ens trobem. Per tant, haurem de col·locar-nos en una vista superior, col·locar el cursor en el centre de la copa i fer click en Spin. En el cas que tinguem més d’una vista tridimensional en pantalla, haurem de seleccionar quina determina la rotació. És aconsellable eliminar els punts duplicats que haurem creat, des del menú Specials (tecla W).

Una eina utilitzada amb Spin ens permet fer duplicats d’un objecte i repartir-los automàticament al llarg d’una circumferència. Farem 8 còpies de la nostra copa i les col·locarem formant un cercle.
Per a això, primer col·locarem la copa allunyada del centre d'una hipotètica taula (aprofitarem per escalar-la a la grandària adequada).

![Figura 2.44. Procés de duplicació d'un objecte amb el mètode SPIN 1.](image)

Ens posem en mode edició, seleccionem tots els punts de la copa, col·loquem en el botó **Degr** els 360 graus que volem que girin les nostres copes a la taula, seleccionem 8 a **Steps**, usem una vista superior, col·loquem el cursor en el 'centre de la taula' i fem click en el botó '**Spin Dup**'.

De la mateixa manera que amb **Spin** havíem creat vèrtexs de més, ara hem duplicat una copa. Per eliminar-la, n'hi ha prou amb pulsar **Supr** (o la tecla X) i seleccionar Vèrtexs en el menú que apareixerà (veure figura 2.45).

![Figura 2.45. Procés de duplicació d'un objecte amb el mètode SPIN 2.](image)

**2.4.10. Screw**

**Screw** (cargolat) és una variació dels sòlids de revolució que afegeix un desplaçament vertical durant la rotació. És parteix sempre d’una vista frontal i una corba poligonal com la que s'aprecia en la imatge.

Ha de col·locar-se el cursor en l'eix, al voltant del qual vol fer-se girar la corba, i en el panell Mesh Tools (del panell d'edició) seleccionem a **Steps** la quantitat de passos a donar i a **Turns** el nombre de girs abans de fer click en **Extrude**.
Eina pel desenvolupament de reflectors de conjunts òptics

Seleccionant 10 passos i 4 girs s’obté, a partir de la corba exemple, el resultat de la dreta.

Figura 2.46. Panell d’edició (Mesh Tools).

Figura 2.47. Procés SCREW.

2.5. Mòduls més rellevants

En aquest apartat mostrarem tots els mòduls que hem utilitzat a l’hora de programar amb Python per tal d’utilitzar totes les eines necessàries de Blender i així automatitzar els processos.

2.5.1 Mòdul Blender

- **Redraw( )**
  
  Dibuixa totes les finestres 3D. S’utilitza per si es vol que aparegui algun tipus de finestra nova per pantalla. Fa com un refresc.

2.5.2. Submòdul Mesh

- **Get(name=None)**
  
  Aconsegueix l’objecte malla de Blender anomenat amb el nom que li hem passat per paràmetre.
Paràmetres:
  o **Name** – El nom de l’objecte malla. [Type = String].

**Retorna:**
Si el nom que li passem existeix retornarà aquesta malla, si no existeix no retorna res. Si no li passem paràmetre retornarà totes les malles que es trobin a l’escena en aquests moments.

- **Mode (mode=0)**
  Aconsegueix i/o afegeix a la malla seleccionada els modes per a l’edició de malla. Els modes es poden veure quan ens trobem a la finestra en 3D per a edició de Malla.

  Paràmetres:
  o **Mode** – El mode seleccionat. Els modes es poden combinar. Si l’omitim, el mode seleccionat no canviarà. [Tipus = Int].

  **Retorna:**
  El mode seleccionat actualment. [Tipus = Int].

**Modes Disponibles:**
  o VERTEX: mode selecció de vèrtexs.
  o EDGE: mode contorn seleccionat.
  o FACE: mode cara seleccionat.

- **New (name=’Mesh’)**
  Crea una nova malla anomenada amb el nom de ‘name’.

  Paràmetres:
  o **Name** – El nom de l’objecte malla. [Tipus = String].

  **Retorna:**
  Una nova malla de Blender. [Tipus = Mesh].

2.5.3. Submòdul NMesh

- **New (name=’Mesh’)**
  Crea una nova malla.

  Paràmetres:
  o **Name** – un nom opcional per crear la malla. [Tipus = String].

  **Retorna:**
  Un nou (buit) objecte NMesh.

- **Vert(x=0, y=0, z=0)**
Eina pel desenvolupament de reflectors de conjunts òptics

Aconsegueix un nou objecte vèrtex.

**Paràmetres:**
- \( X \) – la coordenada x del vèrtex. [Tipus = Float].
- \( Y \) – la coordenada y del vèrtex. [Tipus = Float].
- \( Z \) – la coordenada z del vèrtex. [Tipus = Float].

**Retorna:**
Un nou Objecte \( NMVert \). [Tipus = \( NMVert \)].

2.5.3.1. Classe \( NMFace \)

- **Append(\( \text{vèrtex} \))**
  Afegeix un vèrtex de la llista de vèrtexs a aquesta cara.

**Paràmetres:**
- \( \text{Vèrtex} \) - Un objecte \( NMVert \). [Tipus = \( NMVert \)].

2.5.4. Submòdul \( Scene \)

- **GetCurrent( )**
  Aconsegueix l’escena de Blender que actualment es troba en actiu.

**Retorna:** L’escena activa actual. [Tipus: Escena de Blender].

- **New( name ).**
  Crea una nova Escena de Blender.

**Paràmetres:**
- \( \text{Name} \) – El nom de l’escena. [Tipus = String].

**Retorna:** L’escena creada. [Tipus = Escena de Blender].

- **Get( name = None )**
  Aconsegueix l’escena (s) de Blender.

**Paràmetres:**
- \( \text{Name} \) – El nom de l’escena. [ Tipus = String].

**Retorna:**
Depèn del paràmetre ‘name’:
- \( (name) \): l’escena que rep el nom ‘name’.
- (): Una llista de totes les escenes existents actualment a Blender.
  [Tipus: Escena de Blender o una llista d’escenes de Blender].

UdG 45
2.5.4.1 Classe *Scene*

- **GetName( )**
  Agafa el nom d’aquesta escena.

  **Retorna:** un String.

- **Link(object)**
  Enllaça un objecte a aquesta escena.

  **Paràmetres:**
  - **Object** – Un objecte Blender. [Tipus = Objecte de Blender].

- **SetName(name)**
  Dóna un nom a aquesta escena.

  **Paràmetres:**
  - **Name** – el nou nom. [Tipus = String].

2.5.5. Submòdul *Window*

- **FileSelector(callback, title='SELECT FILE', filename='<default>')**
  Obre la finestra de selecció d’arxius de Blender. Seguidament l’usuari pot seleccionar un arxiu el qual se li passa com a paràmetre ‘callback’ que li hem donat a la funció FileSelector.

  **Paràmetres:**
  - **Callback**: La funció que se li ha d’administrar al FileSelector i rebrà l’arxiu com a paràmetre. [Tipus = funció que accepta un string: \texttt{f(str)}].
  - **Title**: Caràcters que apareixen sobre el botó que confirma la selecció i retorna l’arxiu seleccionat de la finestra. [Tipus = String].
  - **Filename**: un nom d’arxiu. Aquest per defecte de Blender. Get(‘filename’). [Tipus = String].

2.5.6. Submòdul *Lamp*

- **Get (name = None)**
  Aconsegueix l’objecte Llum de Blender a qui li correspon el nom entrat com a paràmetre.

  **Paràmetres:**
  - **Name** – El nom de l’objecte Llum. [Tipus = String].

  **Retorna:**
  Depèn del nom que li hem passat com a paràmetre.
  - **(name)**: L’objecte Lamp a qui li correspon el nom.
Eina pel desenvolupament de reflectors de conjunts òptics

- ( ): Una llista de tots els objectes Lamp que es troben actualment a l’escena.

- **New** *(type = ‘Lamp’, name = ‘LampData’)*
  Crea un nou objecte Lamp.

  **Paràmetres:**
  - *Name* – El nom del Lamp.

  ** Retorna:**
  L’objecte llum creat. [Tipus = Llum de Blender].

2.5.6.1. Clase Lamp

- **SetMode** *(m=None, m2=None, m3=None, m4=None, m5=None, m6=None, m7=None, m8=None)*
  Afegeix el mode del llum. Els modes que els hi passem es posaran a ‘on’.
  Sense arguments tots els modes es posaran a off.

  **Paràmetres:**
  - *M* – un mode. De 1 a 8 modes podem afegir al nostre llum. [Tipus = String].

  **Modes:** Els modes del llum.
  - ‘Shadows’
  - ‘Halo’
  - ‘Layer’
  - ‘Quad’
  - ‘Negative’
  - ‘OnlyShadow’
  - ‘Sphere’
  - ‘Square’
  - ‘NoDiffuse’
  - ‘NoSpecular’
  - ‘RayShadow’

2.5.7. Submòdul Object

- **Get** *(name = None)*
  Aconsegueix l’objecte de Blender.

  ** Paràmetres:**
  - *Name* – El nom de l’objecte. [Tipus = String].
Retorna: Depèn del paràmetre ‘name’.
  ▪ (name): L’objecte amb el nom que li hem passat per paràmetre.
  ▪ (): Una llista amb tots els objectes que es troben actualment a
de l’escena.

- GetSelected()
  Aconsegueix la selecció de l’usuari. Si no hi ha objectes seleccionats,
retornarà una llista buida.

Retorna: Una llista de tots els objectes seleccionats actualment a l’escena.

- New(type, name = 'type')
  Crea un nou objecte.

Paràmetres:
  o Name - el nom de l’objecte. Per defecte, el nom serà el mateix com
    el tipus d’objecte. Si el nom ja existeix, aquest nou objecte tindrà
    un número al final del nom.

Retorna: L’objecte creat.

2.5.7.1. Classe Object

- GetLocation(space)
  Paràmetres:
  o Space – L’espai desitjat per a la localització.

Retorna: La localització de l’objecte (x, y, z). [Tipus = String].

- GetSize(space)
  Paràmetres:
  o Space – L’espai desitjat de la mida.


- Link(datablock)
  Enllaça l’objecte amb una dada. La dada ha de ser un tipus d’objecte.

Paràmetres:
  o Datablock – una dada que sigui un tipus d’objecte.

- SetLocation(x, y, z)
  Afegeix la localització de l’objecte relatiu a l’objecte pare (si existeix).

Paràmetres:
  o X – la coordenada x de la nova posició. [Tipus = Float].
Eina pel desenvolupament de reflectors de conjunts òptics

- \( Y \) – la coordenada y de la nova posició. [Tipus = \textit{Float}].
- \( Z \) – la coordenada z de la nova posició. [Tipus = \textit{Float}].

- **SetMaterials (\textit{materials})**
  Afegeix els materials. L’argument ha de ser una llista de 16 articles o menys. Cada element de la llista és un material.

  \textbf{Paràmetres:}
  - \textit{Materials} – una llista d’objectes tipus ‘\textit{material}’ de Blender. [Tipus = Llista de Materials].

- **SetName (\textit{name})**
  Afegeix el nom de l’objecte. Si entrem una cadena de caràcters més gran de 20 caràcters, aquesta serà escurçada.

  \textbf{Paràmetres:}
  - \textit{Name} – El nou nom de l’objecte. [Tipus = \textit{String}].

- **GetSize(\textit{x}, \textit{y}, \textit{z})**
  Afegeix la mida de l’objecte, relatiu a l’objecte pare (si existeix).

  \textbf{Paràmetres:}
  - \textit{X} – la mida multiplicadora de la X. [Tipus = \textit{Float}].
  - \textit{Y} – la mida multiplicadora de la Y. [Tipus = \textit{Float}].
  - \textit{Z} – la mida multiplicadora de la Z. [Tipus = \textit{Float}].

2.5.7.2. Classe \textit{Constraint}

- **Append (\textit{type})**
  Afegeix una nova restricció al final de la pila de restriccions.

  \textbf{Paràmetres:}
  - \textit{Type} – una especificació constant del tipus de la restricció per crear. [Tipus = Restricció \textit{enter}].

  \textbf{Retorna:} una nova restricció. [Tipus = \textit{Constraint}].

\textbf{Tipus de Constraints:}
  - \textit{TRACKTO}
  - \textit{IKSOLVER}
  - \textit{FOLLOWPATH}
  - \textit{COPYROT}
  - \textit{COPYLOC}
  - \textit{CPOYSIZE}
  - \textit{ACTION}
  - \textit{LOCKTRACK}
  - \textit{STRETCHTO}
  - \textit{FLOOR}
  - \textit{LIMITLOC}
2.5.8. Submòdul Material

- **New** *(name = 'Mat')*
  Crea un nou objecte *Material*.

  **Paràmetres**:
  - **Name** – El nom del material. [Tipus = String].

  **Retorna**: L’objecte *Material* creat. [Tipus = *Material* de Blender].

2.5.8.1. Classe Material

- **setMode**(param, stringN = None)
  Afegeix el mode a aquest *Material*. Hi ha 22 modes que podem donar-li i especificar quin d’ells es troba a ‘on’. Si no l’indiquem el mode, aquest estarà a ‘off’, per tant aquesta funció sense arguments posa a ‘off’ tots els modes del Material.


  **Paràmetres**:
  - **Param** – un mode vàlid (int) or un ‘flag’ (string). També pot estar buit. [Tipus = String, None or Int].
  - **StringN** – un ‘flag’ mode. Els 22 flags poden estar actius al mateix temps. [Tipus = String].

- **SetTexture**(index, texture, texco, mapto)
  Assigna una *Textura* de Blender a l’objecte.

  **Paràmetres**:
  - **Index** – un índex de [0, 9] de la textura del material. [Tipus = Int].
  - **Texture** – Un objecte *Textura* del Blender. [Tipus = Una *Textura* de Blender].
  - **Texco** – opcional *Ored bitflag*. Per defecte *TexCo.ORCO*. [Tipus = Int].
  - **Mapto** – opcional *Ored bitflag*. Per defecte *MapTo.COL*. [Tipus = Int].
2.5.9. Submòdul Draw

- **Create** *(value)*
  Crea un objecte *Button* per defecte.

  **Paràmetres:**
  - *Value* – El valor reservat pel botó. [Tipus = *Int*, *float*, *string* o 3 *floats*].

  **Retorna:** el botó creat. [Tipus = Botó de Blender].

- **Label** *(string, x, y, w, h)*
  Dibuixa un text a la pantalla.

  **Paràmetres:**
  - *String* – El text a dibuixar. [Tipus = *String*].

  **Retorna:** *None*. [Tipus = *None*].

- **Menu** *(name, event, x, y, width, height, default, tooltip)*
  Crea un nou objecte tipus botó de menús.

  **Paràmetres:**
  - *Name* – Definim el contingut del botó. [Tipus = *String*].
  - *Event* – el número d’event que li passem al botó quan l’activem. [Tipus = *Int*].
  - *X* – La posició x més baixa per l’esquerra (horitzontal). Coordenada del botó. [Tipus = *Int*].
  - *Y* – la posició y més baixa per l’esquerra (vertical). La coordenada del botó. [Tipus = *Int*].
  - *Width* – amplada del botó. [Tipus = *Int*].
  - *Height* – alçada del botó. [Tipus = *Int*].
  - *Default* – el número de la opció que estarà seleccionada per defecte. [Tipus = *Int*].
  - *Tooltip* – el quadrat que ens mostra un consell. [Tipus = *String*].
  - *Callback* – un argument opcional que pot contenir una crida a una funció. La funció s’executarà quan aquest botó es premi. Aquesta funció pot acceptar fins a dos paràmetres d’entrada (*event*, *val*). [Tipus = *Function*].

  **Retorna:** El botó creat. [Tipus = Botó de Blender].

- **PushButton** *(name, event, x, y, width, height, default, tooltip = None, callback = None)*
  Crea un nou objecte tipus Botó *(push)*.
Paràmetres:
- **Name** – Definim el contingut del botó. [Tipus = *String*].
- **Event** – el número d’event que li passem al botó quan l’activem. [Tipus = *Int*].
- **X** – La posició x més baixa per l’esquerra (horitzontal). Coordenada del botó. [Tipus = *Int*].
- **Y** – la posició y més baixa per l’esquerra (vertical). La coordenada del botó. [Tipus = *Int*].
- **Width** – amplada del botó. [Tipus = *Int*].
- **Height** – alçada del botó. [Tipus = *Int*].
- **Tooltip** – el requadre que ens mostra un consell. [Tipus = *String*].
- **Callback** – un argument opcional que pot contenir una crida a una funció. La funció s’executarà quan aquest botó es premi. Aquesta funció pot acceptar fins a dos paràmetres d’entrau (*event, val*). [Tipus = *Function*].

- **Slider** (*name, event, x, y, width, heih, initial, min, max, realtime = 1, tooltip = None, callback = None*)
  Crea un nou objecte tipus botó *Slider*.

Paràmetres:
- **Name** – Definim el contingut del botó. [Tipus = *String*].
- **Event** – el número d’event que li passem al botó quan l’activem. [Tipus = *Int*].
- **X** – La posició x més baixa per l’esquerra (horitzontal). Coordenada del botó. [Tipus = *Int*].
- **Y** – la posició y més baixa per l’esquerra (vertical). La coordenada del botó. [Tipus = *Int*].
- **Width** – amplada del botó. [Tipus = *Int*].
- **Height** – alçada del botó. [Tipus = *Int*].
- **Initial** – el valor inicial. [Tipus = *Int* o *float*].
- **Min** – el valor mínim. [Tipus = *Int* o *float*].
- **Max** – el valor màxim. [Tipus = *Int* o *float*].
- **Realtime** – si no és zero (per defecte), el ‘slider’ emetrà events com en edició. [Tipus = *Int*].
- **Tooltip** – el requadre que ens mostra un consell. [Tipus = *String*].
- **Callback** – un argument opcional que pot contenir una crida a una funció. La funció s’executarà quan aquest botó es premi. Aquesta funció pot acceptar fins a dos paràmetres d’entrau (*event, val*). [Tipus = *Function*].

Retorna: El botó creat. [Tipus = Botó de Blender].

- **Toggle** (*name, event, x, y, width, height, default, tooltip = None, callback = None*)
Crea un nou objecte tipus botó *Toggle*.

**Paràmetres:**
- *Name* – Definim el contingut del botó. [Tipus = *String*].
- *Event* – el número d’event que li passem al botó quan l’activem. [Tipus = *Int*].
- *X* – La posició x més baixa per l’esquerra (horitzontal). Coordenada del botó. [Tipus = *Int*].
- *Y* – la posició y més baixa per l’esquerra (vertical). La coordenada del botó. [Tipus = *Int*].
- *Width* – amplada del botó. [Tipus = *Int*].
- *Height* – alçada del botó. [Tipus = *Int*].
- *Default* – el número de la opció que estarà seleccionada per defecte. [Tipus = *Int*].
- *Tooltip* – el requadre que ens mostra un consell. [Tipus = *String*].
- *Callback* – un argument opcional que pot contenir una crida a una funció. La funció s’executarà quan aquest botó es premi. Aquesta funció pot acceptar fins a dos paràmetres d’entrada (event, val). [Tipus = *Function*].

**Retorna:** El botó creat. [Tipus = Botó de Blender].

2.5.10. Submòdul *BGL*

- *glRasterPos*(x, y, z, w)

  *glRasterPos2d, glRasterPos2f, glRasterPos2i, glRasterPos2s, glRasterPos3d, glRasterPos3f, glRasterPos3i, glRasterPos3s, glRasterPos4d, glRasterPos4f, glRasterPos4i, glRasterPos4s, glRasterPos2dv, glRasterPos2fv, glRasterPos2iv, glRasterPos2sv, glRasterPos3dv, glRasterPos3fv, glRasterPos3iv, glRasterPos3sv, glRasterPos4dv, glRasterPos4fv, glRasterPos4iv, glRasterPos4sv

Especifica la posició del refresc per operació de cada píxel de pantalla.

**Paràmetres:**
- *X, y, z, w* – Específica la x, y, z i w com a objecte les coordenades de la posició de l’escombrat.

2.5.11. Submòdul *Camera*

- *Get*(name)

  Agafa l’objecte o objectes tipus *camera* de Blender amb el nom corresponent.

  **Paràmetres:**
  - *Name* – el nom de la càmera. [Tipus = *String*].

  **Retorna:** depèn del paràmetre ‘name’ que li passem:
  - *(name)*: L’objecte càmera amb el nom especificat.
[ ]: una llista amb els objectes càmera que es troben a l’escena. [Tipus = Càmera de Blender o llista de Càmeres de Blender].

- **New**(type='persp', name='CamData')
  Crea un nou objecte Camera

  Paràmetres:
  o **Type** - El tipus de càmera: ‘persp’ o ‘ortho’. [Tipus = String].
  o **Name** – El nom de la càmera. [Tipus = String].

  **Retorna:** L’objecte camera creat. [Tipus = Càmera de Blender].

2.5.11.1. Classe Camera

- **setLens**(lens)
  Afegeix el valor de la lent.

  Paràmetres:
  o **Lens** – el nou valor de la lent. [Tipus = Float].

- **setScale**(scale)
  Afegeix el valor de l’escala.

  Paràmetres:
  o **Scale** – El nou valor de l’escala de la càmera entre [0.01, 1000.00]. [Tipus = Float].

- **setType**(type)
  Afegeix el tipus de càmera.

  Paràmetres:
  o **Type** – el tipus de càmera: ‘persp’ o ‘ortho’. [Tipus = String].
3 DISSENY
3. Disseny

3.1. Conceptes previs

El què explicarem en aquest apartat fa referència el disseny del nostre programa. El nostre objectiu és construir un reflector i per fer-lo ens cal importar una boca. La boca és la part del reflector per on es subjecte la carcassa. Aquesta boca l’aconseguim a partir d’un pla on hi tenim marcat una sèrie de vèrtexs. Aquest conjunt de vèrtexs es troben en un fitxer DXF el qual haurem de carregar prèviament, és a dir, farem una importació de la nostra boca. Haurem de portar a terme la importació del fitxer DXF.

En importar el fitxer DXF és necessari executar una algorisme el qual ens calculi l’envolvent convexa, ja que podem trobar vèrtexs que no pertanyin al contorn de la boca del nostre reflector. L’algorisme que hem implementat s’anomena algorisme de *Graham Scan*, que explicarem més endavant, a l’apartat 3.5, *Algorisme de Graham Scan*.

Una vegada haguem carregat els nostres vèrtexs de la boca del reflector li haurem de donar volum, perquè el què captem de l’arxiu DXF que importem és la boca del reflector però sobre un pla. Per donar volum al reflector són necessaris dos processos: inserir punts a l’interior de la boca i crear la superfície del reflector. Primerament inserirem punts a la part interior de la boca, d’aquesta manera podrem triangular els vèrtexs que componen la superfície del reflector. Seguidament agafarem els vèrtexs necessaris i els aixecarem sobre l’eix z tal i com veiem a la figura 3.1. on alcem els vèrtexs seleccionats (veure capítol 2, *Modelar en Blender*).

Amb el reflector ja acabat de construir només ens cal afegir-hi el llum, el qual anirà situat al centre del reflector.

És necessari crear un pla per sota el reflector que simularà una superfície la qual volem il·luminar, com per exemple un carrer. Aquest pla estarà situat en una distància determinada del reflector la qual l’usuari pot escollir abans de crear el pla. El pla és, simplement, una representació visual del carrer per a l’usuari, ja que el què ens interessa, és calcular la intensitat de llum que arriba sobre el pla i així qualsevol enginyer luminotècnic pot estudiar-ne el resultat. La forma que nosaltres utilitzem per extreure el càlcul d’intensitats és mitjançant imatges, que obtenim de la renderització, realitzada a través d’una càmera situada sobre el pla i, posteriorment, mostrar els valors obtinguts d’aquestes imatges mitjançant la seva representació en corbes de nivell. La
renderització sobre càmeres al terra és una eina auxiliar que nosaltres hem decidit utilitzar però existeixen altres eines per realitzar càlculs d’intensitat de llums.

3.2. Classes auxiliars creades

El plantejament que hem fet a l’hora de crear el nostre reflector ha estat utilitzar unes estructures de dades i així poder guardar tots el vèrtexs que formaran la boca del reflector. Cada vèrtex estarà compost per tres coordenades (x, y, z) on hem determinat que z prendrà el valor zero per defecte i d’aquesta manera treballarem amb objectes de dues dimensions, tot i que com és d’esperar, Blender ens permet treballar en objectes de tres dimensions. Així doncs a través de Python hem utilitzat diferents classes predefinides per nosaltres; unes serveixen per emmagatzemar vèrtexs i algunes altres per emmagatzemar triangles, tot i que totes les classes serveixen per emmagatzemar vèrtexs (sigui de la manera que sigui). Seguidament us mostrarem algunes de les classes principals que hem utilitzat.

3.3.1. Classe Matriu

Primer de tot cal aclarir el concepte de Matriu que pren en aquest projecte. A diferència del què entenem com a matriu en el càlcul matemàtic definim una “Matriu” dins del nostre projecte com el conjunt de vèrtexs que componen el nostre reflector; és una agrupació de vèrtexs, el qual el seu accés es realitza a través d’un índex.

La classe Matriu ens permet guardar els vèrtexs que intervenen en el projecte. Aquesta classe conté una matriu formada per vectors de tres elements, els quals identifiquen la component x, y, z dels vèrtexs.

Com podem comprovar la figura 3.2 veiem que la classe Matriu està formada per una sèrie d’atributs i diferents mètodes que els explicarem a continuació.

<table>
<thead>
<tr>
<th>Clase Matriu</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Elements: int</td>
</tr>
<tr>
<td>- Matriu.Vertex.Boca [[0,0,0]] : float</td>
</tr>
<tr>
<td>+ Copiar_Matriu(matriu: Matriu)</td>
</tr>
<tr>
<td>+ Netejar_Matriu ()</td>
</tr>
<tr>
<td>+ Eliminar_Matriu ()</td>
</tr>
<tr>
<td>+ Buscar_Posicio (x: float, y: float, z: float): int</td>
</tr>
<tr>
<td>+ Mostrar_Matriu ()</td>
</tr>
<tr>
<td>+ Existeix_Vertex_Boca (x: float, y: float, z: float): boolean</td>
</tr>
<tr>
<td>+ Afegir_Vertex ()</td>
</tr>
</tbody>
</table>

**Figura 3.2. Classe Matriu**

**Atributs**
Capítol 3. Disseny

- **Element**: Ens indica la quantitat de vèrtexs que conté la matriu. Ens és molt útil com a índex.
- **Matriu_Vertex_Boca**: és una estructura d’elements on cada element conté un vector de tres posicions (coordenades x, y, i z). Per accedir a algun element d’aquests utilitzarem l’índex que comença per 0, per tant si ens volem adreçar a l’element que es troba a la posició m, utilitzarem \textit{Matriu_Vertex_Boca[m]}. Les coordenades de cada element es troben a la posició 0, 1, i 2, corresponents a x, y i z, respectivament. Així doncs, si volem accedir a la coordenada y de l’element que es troba a la posició m, utilitzarem \textit{Matriu_Vertex_Boca[m][1]}.

**Mètodes**

- **Copiar_Matriu**: Fem una copia d’una matriu a una altra, copiant tots els elements d’una matriu amb tots els seus components.
- **Netejar_Matriu**: Elimina qualsevol element que pugui tenir la matriu.
- **Eliminar_Matriu**: Elimina la instància.
- **Buscar_Posicio**: A partir d’un vèrtex entrat ens retorna la posició (o element) en què es troba aquest vèrtex dins de la matriu.
- **Mostrar_Matriu**: Mostrem tots els elements que conté la matriu per pantalla.
- **Existeix_Vertex_Boca**: A partir d’un vèrtex, ens retorna cert o fals depenent de si existeix aquest vèrtexs en la matriu.
- **Afegir_Vertex**: Entrem el vèrtex que li hem passat per paràmetre a la matriu.

3.4. Classe DXF

Per carregar la boca del reflector hem de tenir en compte que aquesta estarà emmagatzemada en un fitxer tipus DXF i per tant hem de procedir a la lectura d’aquest fitxer, processar les dades emmagatzemades i aconseguir els vèrtexs que formen la boca del reflector.

3.4.1. Format DXF

El format DXF (\textit{Drawing Interchange Format}) és un format vectorial que l'empresa \textit{Autodesk} va llançar per permetre l'intercanvi d'arxius de dibuix entre els diferents programes de CAD. Suporta fins a 256 colors (8 bits).
Hi ha dues versions de DXF (ASCII i binari), que no utilitzen cap algorisme de compressió. Els fitxers de la versió ASCII contenen nombres i ordres a realitzar escrits en codificació ASCII, per la qual cosa poden ser oberts i llegits amb qualsevol editor de text, com el *Notepad* de *Windows*. Aquesta informació indica la ubicació de punts flotants matemàtics o *floating points*, utilitzats per exhibir la imatge en pantalla. Aquest sistema, més lent que el d'altres formats, requereix maquinari avançat per poder funcionar correctament.

Els fitxers en format DXF són reconeguts per la majoria de sistemes CAD i per alguns programes de disseny gràfic vectorial, com *Corel Draw* i *Adobe Illustrator*, que poden manipular-ho sense dificultats.

3.4.2. Lectura del fitxer DXF

Com ja us hem explicat abans, els arxius DXF es poden obrir com a fitxers de text perquè contenen codi ASCII. Dins d’aquest fitxer hi ha guardada la informació de la posició dels vèrtexs que formen la boca del nostre reflector. Així doncs des de Python només ens cal utilitzar una de les instruccions ja explicades anteriorment i obrir el fitxer. Un cop el fitxer està obert simplement ens cal llegir línia a línia buscant la posició dels vèrtexs. Com que el fitxer es troba organitzat per grups, haurem de buscar el grup que a nosaltres ens interessa (grup *ENTITIES*). Un cop tenim el grup *Entities* localitzat ens hem d’assegurar quin és el vèrtex que estem llegint, és a dir, saber si aquest vèrtex que llegim formarà part d’una línia o d’un arc; això ho sabrem perquè a la següent línia del *ENTITIES* ens marca línia amb LINE i arc amb ARC. A continuació hi ha la posició del vèrtexs. Aquesta és la rutina que nosaltres haurem d’executar repetidament fins a final de fitxer.

A continuació mostrem una part de l’algorisme de lectura del fitxer DXF on tracta una línia o un arc segons el què es llegeixi a la línia del fitxer. Comprovem que l’algorisme busca si la línia indica final de fitxer o final de secció, en cas contrari tractarà una línia o un arc.

**ALGORISME DXF::lectura**

**INICI**

**MENTRES** Línia <> “ENDSEC” and Línia <> “EOF”

**SI** Línia <> “ENDSEC”

**SI** Línia == “LINE” **LLAVORS**

*ProcessLineEntity*(Fitxer)

**SINÓ, SI** Línia == “ARC” **LLAVORS**

*ProcessArcEntity*(Fitxer)

**FI_SI**

**ELSE**

*Llegim_Parell_Línies*(Fitxer)

**FI_SI**

**FI_SI**

**FI_MENTRES**

**FI**

**Funció Tractar_Linia (línia)**
A l’hora d’importar línies del fitxer DXF, és a dir, llegir línies del fitxer de text, hem d’agafar caràcter per caràcter fins que troben un retorn de carro i d’aquesta manera reconstruir la línia que es troba guardada dins del fitxer de text; reconstrueix la línia sense espais en blanc.

A continuació mostrem l’algorisme de la funció \textit{Tractar\_Línia(línia)}:

\begin{verbatim}
ALGORISME DXF::Tractar\_Línia (Línia: String)
ENTRADA: La cadena de caràcters.
SORTIDA: La nova línia sense espais en blanc.
INICI
    Nova\_Línia = ' '; FOR i = Longitud(Línia) FER
        SI Línia[i] <> " " and Línia[i] <> ‘\n’ LLAVORS
            Nova\_Línia = Nova\_Línia + Línia[i]
        FI_SI
    FI_FOR
FI

Funció \textit{Llegim\_Parell\_Línies(filename)}

A partir del nom del fitxer, llegim línia a línia, mitjançant un \textit{readline()} del Python i la tractem de la manera que hem explicant a l’apartat anterior. Ens cal llegir les línies de dos en dos per tal de localitzar la informació correcta, ja que a la primera línia ens trobarem amb el conjunt al qual pertany i a continuació, a la següent línia, ens trobarem amb el codi a què pertany.

\begin{verbatim}
ALGORISME dxf::Llegim\_Parell\_Línies(Filename)
ENTRADA: El nom del fitxer que volem llegir.
SORTIDA: El grup a què pertany la línia.
INICI
    GrupCodi = Filename.readline()
    GrupCodi = Tractar\_Línia (GrupCodi)
    Línia = Filename.readline()
    Línia = Tractar\_Línia (Línia)
FI

Funcions \textit{ProcessLineEntity(filename)} i \textit{ProcessArcEntity (Filename)}

El fitxer DXF estarà compost per un seguits de codis; a mesura que nosaltres anem llegint aquest codi sabrem si el què ve a continuació és la posició d’una línia que haurèm de crear nosaltres pel nostre reflector. La funció que realment realitza el \textit{ProcessLineEntity} és obtenir la posició del vèrtex (x, y, z) quan a la línia que llegim del fitxer apareix ‘LINE’. D’igual manera funciona el \textit{ProcessArcEntity} però s’activa quan en el fitxer DXF llegim un ‘ARC’.

A continuació mostrem una part de l’algorisme de les dues funcions:
Tal i com podem observar en el nostre algorisme, depenent de quin codi hem llegit, guardarem la nostra línia d’una manera o una altra.

En l’algorisme per tractar les línies podem apreciar diferències en els codis que busquem:

3.5. Desenvolupament de l’algorisme de Graham Scan

Abans d’explicar l’algorisme de Graham Scan explicarem quin és el motiu que ens ha portat al desenvolupament d’aquest algorisme al nostre projecte. Primerament explicarem el problema de l’envolvent convexa, que és la situació en què ens trobàvem a l’hora de carregar la boca del reflector.
3.5.1. L’envolvent convexa

El càlcul de l'envolvent convexa d'un conjunt de punts ha estat, sens dubte, un dels problemes més bàsics i més àmpliament estudiats al llarg de la història de la Geometria Computacional. A més a més, els camps en què hi ha resultat un càlcul útil sorprenent per l'aparentment poc relacionats amb les matemàtiques en general i la geometria en particular.

Un conjunt convex del pla es defineix com el conjunt que verifica que el segment que uneix a dos qualsevol dels seus punts està totally contingut en ell. Evidentment, aquesta definició pot ser aplicada en qualsevol espai n-dimensionals. La idea de conjunt convex ens porta immediatament a la d'envolvent convexa, considerant a aquest com el menor conjunt convex que conté el conjunt de la \figura{3.3}. Aquest concepte és fàcil d'entendre d'alguna manera si considerem a l'envolvent convexa d'un conjunt de punts en el pla, com la forma que adquiriria una goma elàstica embolicant a tots els punts del conjunt que estan fixes sobre el pla. Novament aquesta idea és extensible intuïtivament a altres dimensions, i pot servir de partida per construir algorismes simples per al càlcul de l'envolvent convexa.

\figura{3.3}{Envolvent convexa S.}

L'estudi de l'envolvent convexa en dues dimensions ha estat objecte d'especial interès. Existeixen una sèrie de característiques associades als punts que formen part de l'envolvent convexa en el pla i que han servit igualment per definir-la i construir-la. Un punt \( q \in S \) pertany a l'envolvent convexa de \( S \), \( q \in CH(S) \), si és possible traçar una recta passant per aquest punt de manera que tot el conjunt \( S' \), \( S' = S-\{q\} \) quedi a un costat d'aquesta recta.

Com s'ha indicat en anterioritat, el càlcul de l'envolvent convexa ha estat aplicat àmpliament a disciplines a vegades alienes a la Geometria Computacional i en espais diferents de dues dimensions, com en el cas del reconeixement de patrons, processament d’imatges, etc.

Els algorismes proposats per a aquest problema són realment nombrosos. Sens dubte és un dels plantejaments de la Geometria Computacional que més solucions ha aconseguit. Des de fins i tot abans del naixement oficial d'aquesta disciplina es coneix el treball de Graham l'any 1972. S'han proposat estratègies següint els paradigmes més empleats com el mètode incremental i el \textit{divideix i guanyaràs}. Tots aquests algorismes
Eina pel desenvolupament de reflectors de conjunts òptics

La construcció de l'envolvent convexa en temps $\Theta(n \log n)$. El mètode proposat per Preparata i Shamos el 1985 va ser denominat Quickhull per la seva similitud amb el mètode d'ordenació Quicksort. Una altra proposta diferent es va donar a conèixer el 1973 com La marxa de Jarvis. Tots aquests treballs constitueixen el conjunt clàssic d'algorismes per a la resolució de l'envolvent convexa, a què caldria afegir altres tècniques posteriors.

L'envolvent convexa posseeix una cota de $\Omega(n \log n)$, és a dir, no es pot trobar una algorisme que treballi en menor temps per a tots els casos.

3.5.2. Graham’s Scan

Aquest és sens dubte un dels algorismes més importants per al càlcul de l'envolvent convexa, no únicament per la seva senzillesa sinó també per la seva rapidesa.

El procés realitza primer una ordenació angular dels punts. El punt agafat per referència ha de ser un punt de l’envolvent convexa, per exemple el de menor ordre i al que anomenarem $p_0$. La resta de punts queden ordenats d'esquerra a dreta obtenint la sequència $p_1, p_2, \ldots, p_{n-1}$ (veure figura 3.4).

![Figura 3.4. Ordenació de punts segons Graham’s Scan](image)

El procés seguit per Graham agafa successivament triangles escollits segons l'ordre aconseguit després de l'ordenació. Sempre que el triangle format $abc$, tingui una superfície positiva, el vèrtex central, és a dir, el vèrtex $b$ serà tingut en compte com a probable punt de l'envolvent. Però quan aquesta àrea és negativa, $b$ és descartat definitivament.
3.5.3. Algorisme Graham’s Scan

ALGORISME GRAHAM (VAR P: TipusNuvolPunts, n:Enter, VAR Env: TipusPila)
ENTRADA: El núvol de punts P de mida n.
SORTIDA: L’envolvent convexa de P, CH(P).

VARIABLES:
  Min: Enter // manté l’índex del punt de menor ordre.
  t: Enter // serveix per treure i afegir dades en la pila.
  i: Enter // recórrer tots els elements de P.

INICI
  Min ← PuntMenorOrdenat(P,n)
  OrdenarAngularment(P,n,min)
  PilaCrea(Env)
  PilaPush(Env,P[n-1])
  PilaPush(Env,P[0])
  I ← 1

MENTRES i < n-1 REPETIR
  t ← PilaPop (Env)
  SI Esquerra (PilaTop(Env), t, P[i])
    Llavors
      PilaPush(Env, t)
      PilaPush(Env, i)
      i ← i+1
  SINÓ t ← PilaPop (Env)
  FI_SI
FI_MENTRES
FI

L'ordre d'execució d'aquest algorisme ve donat per la necessitat d'ordenar els punts angularment. Qualsevol ordenació necessita almenys un temps $O(n \log n)$. A aquest temps se li ha d'afegir el del procés ideat per Graham. Per esbrinar aquest temps només cal d'adonar-se que en cada pas un punt és tingut en compte o és descartat. En el pitjor dels casos, quan l'envolvent convexa és un triangle, necessitem de l'ordre de $2n-3$ operacions.

Tot això ens indica que Graham's Scan és un algorisme que treballa en temps òptim $O(n \log n)$.

Així doncs, aquest algorisme l’apliquem sobre el nostre projecte per resoldre l’envolvent convexa d’aquesta manera tenim els nostres punts que formen el contorn de la nostra boca del reflector ordenats. Juntament amb les funcions FindLowest(matriu) i Preparar_Unio_De_Vertex(matriu) i La nostra funció Graham_Scan(matriu_vella, matriu_nova) realitzen el codi necessari per dur a terme aquest algorisme.

La funció FindLowest ens busca l’element més petit de la matriu necessari per aplicar l’algorisme de Graham Scan i la funció Preparar_Unio_De_Vertex ens reordena la nostra matriu segons l’angle que formen els vèrtexs en el pla horitzontal, també necessari per aplicar l’algorisme de Graham Scan.
3.5.4. Dibuixant els vèrtexs per pantalla

Una de les funcions més utilitzades al nostre disseny és *Dibuixar Punts(matriu)*. Aquesta funció s’encarrega d’agafar tots els punts continguts a la matriu que li passem per paràmetre i els dibuixa per pantalla. Utilitza el mòdul *NVert* tal i com us expliquem en l’apartat de mòduls més rellevants de Blender (capítol 2, *Modelar en Blender*) on existeix una funció *append* la qual ens permet dibuixar un vèrtex sobre la superfície de Blender.

3.6. Creant la superfície del reflector

Per crear una superfície en Blender, com a mínim ens fan falta tres vèrtexs. Amb tres vèrtexs podem crear una cara i per tant podriem percebre aquesta cara a l’hora de fer una renderització de la imatge. En el nostra cas, doncs, un cop hem dibuixat els vèrtexs del contorn de la boca ens cal unir aquests vèrtexs entre ells per tal d’aconseguir una superfície plana. De totes maneres, no volem aconseguir només una superfície plana, perquè els reflectors necessiten prendre volum i per donar volum al reflector haurem d’inserir nous punts a l’interior de la boca i així poder-los alçar en direcció de l’eix z.

3.6.1. Creació dels punts interiors de la boca

La funció que hem utilitzat per crear tots aquests punts s’anomena *CrearPuntsInteriors(matriu)*. S’encarrega d’agafar els vèrtexs que componen la boca del reflector i crear una segona matriu on els vèrtexs de la boca es trobin dins d’aquesta última. Tenim un exemple en la figura 3.5.

![Figura 3.5](image)

A partir d’aquesta matriu creada, només ens cal recórrer tots els vèrtexs d’aquesta i comprovar que el vèrtexs pertanyen a l’interior de la boca. Si un vèrtex pertany a l’interior de la boca l’emmagatzemarem, en cas contrari el descartarem.

Per saber si un vèrtex pertany, o no, a l’interior del reflector ho farem mitjançant la funció *InterseccióVertical(P1, P2, P3)* amb la qual P3 serà el nostre vèrtex candidat a ser acceptat o a ser descartat. Mitjançant un bucle comprovarem si P3 és candidat, o no,
a pertànyer a l’interior de la boca, recorrent tots els punts que formen el contorn de la boca. Si en algun parell de punts P3 pertany a l’interior de la boca, aquest serà acceptat, en cas contrari, aquest serà descartat. P1 i P2 són dos punts consecutius que formen part del contorn de la boca. Si des de P3 tirem una línia vertical i passa creuant la línia entre P1 i P2 un nombre senar de vegades, aquest pertanyerà a l’interior de la boca; si no creua no pertany a l’interior de la boca i si creua un nombre parell de vegades tampoc pertany a l’interior de la boca. La figura 3.6 ens mostra aquest procés amb més claredat.

\[
P_{1x} (1 - t) + P_{2x} * t = P_{3x},
\]

\[
P_{1y} (1 - t) + P_{2y} * t = P_{3y} + S
\]

\[
t = \frac{P_{3x} - P_{1x}}{P_{2x} - P_{1x}} \quad t \in [0, 1]
\]

\[
S \geq 0
\]

**Figura 3.6.** Procés per a conèixer si un vèrtex pertany a l’interior de la boca del reflector.

3.6.2. El procés de la triangulació

La triangulació és necessària perquè una vegada tenim creats tots els vèrtexs que formaran la nostra boca del reflector, ens falta unir-los entre ells (veure figura 3.7) i formar cares triangulars per així poder generar la seva superfície. Com que per formar cares ens fan falta quatre vèrtexs, o com a mínim tres vèrtexs per a formar-ne un triangle, tenim setze combinacions.

**Figura 3.7.** Procés de triangulació.
Les combinacions les formem amb zeros i uns on un 0 significa que el vèrtex no correspon a l’interior de la boca, i un 1 significa que el vèrtex pertany a l’interior de la boca. A continuació us presentem un algorisme que està inspirat en l’algorisme *Marching Cubes*, utilitzat a l’àmbit de la imatge mèdica.

Figura 3.8. Combinacions entre vèrtexs a l’hora de fer la triangulació. D’esquerra a dreta i de dalt a baix: casos amb quatre vèrtexs dintre, quatre vèrtexs fora, i tres vèrtexs dintre i un fora de la boca.

Les primeres sis combinacions (veure figura 3.8) ens mostren diferents casos els quals es troben detallats més endavant; la seva implementació es pot veure en el capítol 4. Implementació. Cal esmentar la diferència entre els casos 1111 i 0000, que els mostrem de costat, on el cas que conté quatre uns vol dir que els seus quatre vèrtexs es troben a l’interior de la boca del reflector, en canvi, el cas que conté quatre zeros vol dir que els seus quatre vèrtexs es troben a fora de la boca del reflector. La resta de casos de la figura 3.8 contenen un sol vèrtex a fora de la boca.

Figura 3.9. Combinacions entre vèrtexs a l’hora de fer la triangulació. Casos amb dos vèrtexs dintre i 2 vèrtexs fora de la boca.
La figura 3.9 conté casos en què tenen el mateix nombre de zeros que de uns, això vol dir que dos vèrtexs pertanyen a l’interior de la boca del reflector i dos vèrtexs no pertanyen a l’interior de la boca (pertanyen a l’exterior del reflector). És important veure que existeixen dos casos en la figura 3.9 en què és impossible que s’esdevinguin com és el cas 1001 i el cas 0110, ja que els seus dos vèrtexs formen diagonal (veure figura 3.10).

![Figura 3.10. Casos impossibles de trobar-nos, ja que el reflector hauria de ser molt prim.](image)

Els quatre últims casos de la figura 3.11 contenen tres zeros i només un sol un. Això vol dir que tres vèrtexs pertanyen a fora de la boca i només un sol vèrtex pertany a l’interior de la boca del reflector.

![Figura 3.11. Combinacions entre vèrtexs a l’hora de fer la triangulació. Casos amb tres vèrtexs fora i un vèrtex dintre.](image)
Eina pel desenvolupament de reflectors de conjunts òptics

A partir d’aquestes combinacions ja podrem crear la triangulació utilitzant un vector de quatre posicions on cada posició indicarà el número de vèrtex i l’omplirem de zeros i uns, depenent de si el punt es troba a fora o a dins.

Considerem un vector de quatre posicions on cada una de les seves posicions és un vèrtex. Una de les seves possibles combinacions podria ésser 1,1,1,1 el qual s’interpreta que tots els seus quatre vèrtexs es troben dins l’interior de la boca. Una altra possible combinació podria ésser 0,0,0,0 i en aquest cas voldria dir que tots els seus vèrtexs es troben a fora de la boca del reflector. Seguidament mostrem totes les possibles combinacions que ens podem trobar agafant quatre vèrtexs i com els triangularitzzem:

**Cas 1111:**

Com podem observar en la figura comprovem que si tenim els quatre vèrtexs a l’interior de la boca podem formar dos triangles: un triangle amb els vèrtexs 1,2 i 3 i un altre triangle amb els vèrtexs 2,3 i 4.

El nostre disseny simplement ha de tenir en compte que quan es trobi amb aquesta situació haurà de crear aquests dos triangles.

**Cas 0000:**

Aquest resulta ser el cas més fàcil perquè no hem de crear cap triangle perquè els quatre vèrtexs es troben a fora de la boca del reflector.

**Cas 0111:**

Ara tenim tres vèrtexs que es troben a l’interior de la boca i un sol vèrtex a l’exterior. El què s’ha de fer en aquest cas és, primerament, trobar la intersecció entre dos punts que pertanyen al contorn del reflector i el vèrtex que es troba a fora de la boca, tant horitzontal com vertical i crear tres triangles. Així doncs, agafarem els vèrtexs 1 i 2 i comprovarem que existeix una intersecció horitzontal. El que realment fem és comprovar, a cada parell de vèrtexs que agafem, si existeix cap intersecció amb tots els vèrtexs que componen el contorn de punts del reflector. Quan trobem una intersecció agafem el nostre punt a on interseca i l’afegim a la nostra matrícula de triangulació. D’igual manera actuem quan trobem una intersecció vertical. Un cop tenim els punts a on interseguen, ja podem crear els tres triangles. El primer triangle es realitzarà amb el punt on hi ha intersecció vertical, el punt on hi ha intersecció horitzontal i el vèrtex 3; el segon triangle es crea a partir del punt on hi ha intersecció horitzontal, el vèrtex 2 i el vèrtex 3; el tercer i últim triangle es crea amb els vèrtexs 2,3 i 4.
Cas 1011:

És el mateix cas que l’anterior però ara els vèrtexs que es troben a l’interior de la boca són 1, 3 i 4; el vèrtex que es troba a l’exterior és el 2. Per tant crearem 3 triangles. El primer triangle estarà format pel vèrtex 1, la intersecció horitzontal entre el vèrtex 1 i el 2 i finalment el vèrtex 3; el segon triangle es formarà amb els dos punts d’intersecció localitzats i el vèrtexs 3. El tercer triangle apareix unit el vèrtex 3, el vèrtex 4 i el vèrtex que apareix amb la intersecció vertical entre el vèrtex 2 i el vèrtex 4.

Cas 1101:

Un cas similar als dos vistos anteriorment perquè també podrem formar tres triangles. Ara, però, el punt que queda a l’exterior de la boca és el vèrtex 3, la resta es troben situats a l’interior de la boca.

El primer triangle es formarà amb el vèrtexs 1, 2 i la intersecció vertical entre el vèrtexs 1 i el vèrtexs 3; el segon triangle es formarà amb el vèrtexs 2 i les dues interseccions trobades, tant la horitzontal com la vertical; el tercer triangle surt a partir de la unió entre el vèrtex 2, el vèrtex 4 i la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4.

Cas 1110:

Ara el vèrtexs que queda situat a la part exterior de la boca del reflector és el vèrtex 4.

El primer triangle que formem és la unió dels vèrtexs 1, 2 i 3; el segon triangle el formarem amb la unió del vèrtex 2, el vèrtex 3 i la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4; per últim el triangle format pel vèrtex 2 i les dues interseccions, una entre el vèrtex 2 i el vèrtex 4 i l’altra entre el vèrtex 3 i el vèrtex 4.

Cas 0011:

Aquest cas a diferència dels anteriors podrem crear dos triangles, en comptes de tres, perquè ens trobem amb dos vèrtexs a l’interior de la boca i dos vèrtexs que es troben a l’exterior de la boca. Els vèrtexs 3 i 4 es troben a l’interior de la boca mentre que els vèrtexs 1 i 2 es troben a l’exterior de la boca. Aquí també trobem interseccions però en aquest cas es tractaran de interseccions verticals, així doncs hi ha intersecció vertical entre els vèrtexs 1 i 3 i els vèrtexs 2 i 4.

Els triangles que formarem seran: el primer triangle, per la intersecció vertical entre el vèrtex 1 i el vèrtex 3, la intersecció vertical entre el vèrtex 2 i el vèrtex 4 units.
amb el vèrtex 3; el segon triangle es formarà a partir del vèrtex 3, el vèrtex 4 i la intersecció entre el vèrtex 2 i el vèrtex 4.

**Cas 1100:**

Un cas similar al d’abans és el cas 1100 on els vèrtexs 1 i 2 es troben a l’interior de la boca del reflector i els vèrtexs 3 i 4 es troben a l’exterior de la boca.

Els triangles formats són: el primer, unint el vèrtex 1, el vèrtex 2 i la intersecció vertical entre el vèrtex 1 i el vèrtex 3; el segon triangle es formarà amb la unió del vèrtex 2, la intersecció vertical entre el vèrtex 1 i el vèrtex 3, i la intersecció vertical entre el vèrtex 2 i el vèrtex 4.

**Cas 0101:**

Els vèrtexs que es troben a l’interior de la boca són el vèrtex 2 i el vèrtex 4.

Els dos triangles que formarem seran: el primer, amb la unió de la intersecció horitzontal entre el vèrtex 1 i el vèrtex 2, el vèrtex 2, i la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4; el segon triangle es formarà amb el vèrtex 2, la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4, i el vèrtex 4.

**Cas 1010:**

Aquest cas també ens apareixen dos triangles, on els vèrtexs que es troben a l’interior de la boca són el vèrtex 1 i el vèrtex 3.

El primer triangle es formarà amb el vèrtex 1, la intersecció horitzontal entre el vèrtex 1 i el vèrtex 2, i el vèrtex 3; el segon triangle es formarà amb la intersecció horitzontal entre el vèrtex 1 i el vèrtex 2, el vèrtex 3, i la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4.

**Cas 1000:**

Aquest cas es diferencia de la resta de casos anteriors perquè només tenim un vèrtex que es troba a l’interior de la boca, per tant, tal i com podem veure en la figura, el vèrtex 1 es trobarà a l’interior de la boca.

L’únic triangle que podem formar és amb la unió del vèrtex 1 amb la intersecció horitzontal entre el vèrtex 1 i el vèrtex 2, i la intersecció vertical entre el vèrtex 1 i el vèrtex 3.
Cas 0100:

Tornem a trobar un únic vèrtex pertanyent a l’interior de la boca del reflector, el vèrtex 2.

Igual que en el cas anterior podem formar un únic triangle. El triangle el formarem a partir de la intersecció horitzontal entre el vèrtex 1 i el vèrtex 2, el vèrtex 2 i la intersecció vertical entre el vèrtex 2 i el vèrtex 4.

Cas 0010:

El vèrtex 3 és l’únic vèrtex que trobem a l’interior de la boca del reflector.

El triangle que podem formar, tal i com observem a la figura, és a partir de la unió de la intersecció vertical entre el vèrtex 1 i el vèrtex 3, el vèrtex 3, i la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4.

Cas 0001:

L’últim cas que trobem és amb el vèrtex 4 pertanyent a l’interior de la boca i la resta de vèrtexs pertanyents a l’exterior de la boca del reflector.

L’únic triangle que podem formar és amb la unió de la intersecció vertical entre el vèrtex 2 i el vèrtex 4, amb la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4, i el vèrtex 4.

Tots aquests casos són les úniques possibles combinacions que podem formar amb quatre vèrtexs i, per tant, s’han hagut d’implementar tants casos com combinacions hi pugui haver, tot i que per a la similitud hem pogut reduir-ho a quatre grups i per tant quatre funcions diferents. Aquestes les hem pogut agrupar per vèrtexs implicats, és a dir, el grup que formarà dos triangles hi tindrem implicats quatre vèrtexs, el grup que formarà un triangle hi tindrem implicats tres vèrtexs i el grup que formen tres triangles hi tindrem implicats cinc vèrtexs. Aquest darrer grup l’hem dividit amb dos grups per tant en total tindrem quatre agrupacions les quals formaran quatre funcions diferents i cada una d’elles actuarà d’una manera diferent.
3.7. Generació de volum del reflector

La primera vegada que visualitzem el reflector sobre la interfície del Blender observem que és completament pla (veure figura 3.12, per tant haurem de donar-li volum.

**Figura 3.12.** Interfície de Blender on es pot apreciar la boca del reflector acabada de carregar. La boca és completament plana.

Per donar volum a la boca haurem d’obrir el mode edició i d’aquesta manera ens podrem situar sobre els punts desitjats i moure’ls en la direcció a l’eix Z. D’aquesta manera la nostra boca prendrà el nostre volum desitjat (veure figura 3.13).

**Figura 3.13.** Interfície de Blender en mode d’edició. Podem donar alçada als vèrtexs seleccionats mitjançant el requadre “Transform Properties” per a més precisió.
Si recordem l’apartat de Blender (capítol 2, *Modelar en Blender*), sabrem que també podem moure els vèrtexs utilitzant coordenades exactes i per tant podrem donar la nova distància que vulguem que el vèrtex seleccionat prengui com a nou valor per a la seva coordenada z, d’aquesta manera evitarem que la resta de coordenades es modifiquin sense voler. Podem veure una mostra del panell a on podem introduir el valor de les coordenades en la figura 3.13.

3.8. Col·locació del llum sota la boca del reflector

Per crear el llum prenem les coordenades del reflector i el dibuixem al centre d’aquest mitjançant la funció `New()` pertanyent a la classe `Lamp`. Cal destacar que podem seleccionar dos tipus de llum: `Lamp` i `Spot`. Dependent del tipus de llum que selecioni l’usuari cridarem a una funció o una altra.

**ALGORISME ***CrearLlum*

**INICI**

SI llum = ”Spot” LLAVORS
L ← CrearLlumNou(Spot)
FI_SI
SI llum=”Lamp” LLAVORS
L ← CrearLlumNou(Lamp)
FI_SI
FI

3.9. Procés per crear el pla a il·luminar

Una vegada tenim el reflector creat ens cal crear un pla, el qual simularà la superfície del carrer on estarà situat el fanal. El motiu el qual construïm un pla és pel sol fet que l’usuari observi la distància que hi ha entre la superfície del terra i el reflector. El què a un enginyer luminotècnic li interessa, és la intensitat de llum que arriba a la superfície del terra i és generada pel reflector. Per aquest motiu, prendrem les imatges des del pla, utilitzant una càmera. Com que normalment la distància entre el reflector i el pla és gran, es pot seleccionar-la mitjançant el control que mostrem a la figura 3.14.

![Figura 3.14](image)

**Figura 3.14.** Interfície on es mostra, amb una fletxa vermella, el control que determina la distància entre el reflector i el pla.
Eina pel desenvolupament de reflectors de conjunts òptics

Aquest pla ens serveix per situar la càmera amb la qual extraurem les imatges de la renderització. El pla ens limita l’àrea per on la càmera es desplaçarà. Aquesta àrea ve determinada per la grandària del reflector, és a dir, el pla ha de ser proporcionalment més gran que el reflector que hem inserit. A continuació mostrem un petit algorisme on realitzem la creació del pla:

**ALGORISME** *CrearPla*(x: Real, y: Real, z: Real)
**ENTRADA:** les coordenades x, y, z de tipus real
**INICI**
  Vèrtex.CrearVèrtex(x,y,z)
  Pla.Afegir_Vèrtex(Vèrtex)
  Vèrtex.CrearVèrtex(màximaX, y, z)
  Pla.Afegir_Vèrtex(Vèrtex)
  Vèrtex.CrearVèrtex(màximaX, màximaY, z)
  Pla.Afegir_Vèrtex(Vèrtex)
  Vèrtex.CrearVèrtex(x, màximaY, z)
  Pla.Afegir_Vèrtex(Vèrtex)
  Pla.Redimensionar(2)
  Cara.CrearCara()
  Cara.AfegirPla(Pla)
**FI**

Com podem comprovar en l’algorisme anterior, veiem que utilitzem una funció anomenada *Redimensionar*, aquesta ens escala l’objecte tantes vegades com li passem per paràmetre, en el nostre cas seran dues vegades.

### 3.10. Procés de renderització

El procés de renderització consta de diferents parts. Primerament necessitem saber quantes imatges pel render hem de treure. Hem de partir de la idea que l’usuari pugui escollir el número d’imatges que vol aconseguir del reflector. Hem implementat un control que permet l’entrada del número de renders (veure figura 3.15). Hem de remarcar que si volem molta més qualitat en el resultat final, s’haurà d’escollir un número d’imatges molt més elevat. Això voldrà dir que el preu a pagar per una millor qualitat serà un temps, en calcular el resultat final, molt més elevat.

![](image)

**Figura 3.15.** Interfície on es mostra amb una fletxa el control on podem especificar el número d’imatges que volem obtenir per a a la renderització.

Un cop obtenim la quantitat d’imatges que es vol aconseguir pel render, haurem de distribuir la càmera de tal manera que ens quedin imatges equidistants.
Per distribuir la càmera ho farem mitjançant dos bucles. A partir del número d’imatges que l’usuari vulgui obtenir en farem l’arrel quadrada, d’aquesta manera els bucles arribaran fins a l’arrel quadrada obtinguda. Podem veure l’algorisme a continuació:

**ALGORISME MoureCamera(Nimatges: Int)**  
ENTRADA: El número d’imatges que volem aconseguir.  
INI\(\text{c}\)  
\[
\begin{align*}
\text{LongitudX} & \leftarrow \text{màximaX-minimaX} \\
\text{LongitudY} & \leftarrow \text{màximaY-minimaY} \\
\text{Parts} & \leftarrow \text{ArrelQuadrada(Nimatges)} \\
\text{DivisionsX} & \leftarrow \text{Longitudx / Parts} \\
\text{DivisionsY} & \leftarrow \text{Longitudy / Parts}
\end{align*}
\]
FOR n = 0 to Parts FER  
\[
\begin{align*}
\text{Y} & \leftarrow \text{mínimaY} \\
\text{MitjaX} & \leftarrow (2x + \text{DivisionsX}) / 2
\end{align*}
\]
FOR m = 0 to Parts FER  
\[
\begin{align*}
\text{MitjaY} & \leftarrow (2y + \text{DivisionsY}) / 2 \\
\text{Càmera.SituarPosició(mitjaX, mitjaY, z)}
\end{align*}
\]
FI_FOR
FI

A cada pas del bucle imbricat, el què farem és posicionar la càmera al seu lloc i sempre enfocant a la nostra boca del reflector, des d’una punta del pla creat anteriorment fins a l’altre extrem del mateix pla. Una altra de les tasques que hem de cuidar, és realitzar el render, és a dir, extreure la imatge de la càmera en aquell precis instant i guardar-lo en un fitxer. A l’hora de crear el fitxer de text hem de ser cuidadosos i mantenir el nom original de la imatge canviant els tres darrers caràcters els quals indicaran el número de renderitzacions que portem fetes, per exemple, 001, 002, 003, etc.

**ALGORISME ConstruirNomFitxer(NomFitxer)**  
ENTRADA: li passem el nom del fitxer que l’usuari ha entrat  
SORTIDA: retornem el nom del fitxer amb els tres darrers caràcters corresponents  
INI\(\text{c}\)  
\[
\begin{align*}
\text{Part} & \leftarrow '001' \\
\text{Nom} & \leftarrow \text{NomFitxer + part} \\
\text{Número} & \leftarrow \text{ConvertirCaràcterAEnter(part)} \\
\text{Número} & \leftarrow \text{Número +1} \\
\text{SI Número<10 LLAVORS} \\
\quad \text{Part} & \leftarrow '00' + \text{ConvertirEnterACaràcter(Número)} \\
\text{FIN, } \\
\text{SI Número <10 i Número >9 LLAVORS} \\
\quad \text{Part} & \leftarrow '0' + \text{ConvertirEnterACaràcter(Número)} \\
\text{FIN_SI} \\
\text{FINÓ}
\end{align*}
\]
Per altra banda també hem de tenir en compte el directori on l’usuari vol guardar aquests arxius que el render crearà, controlar que si el directori no existeix, aquest sigui creat i si l’arxiu que volem crear ja existeix, aquest sigui substituït pel nou fitxer de render.

3.11. Verificació del resultat en el Matlab

En aquest apartat explicarem com extreu re la informació en forma de gràfica, i així poder estudiar les intensitats de llum que ofereix el nostre reflector sobre un terra, mitjançant el programa de càlculs numèrics amb vectors i matrius, MATLAB. L’avantatge que té el MATLAB és que pot treballar amb nombres escalars (tant reals com complexes), amb cadenes de caràcters, etc. Això ens afavoreix la feina a l’hora de llegir els nostres fitxers de text, i les imatges que hem generat anteriorment quan hem realitzat la renderització amb Blender, o el fitxer de text que conté les posicions corresponents a cada imatge obtinguda. A més, una de les capacitats més importants de MATLAB és la de realitzar una àmplia varietat de gràfics de dues i tres dimensions. Cal dir que MATLAB té un llenguatge de programació propi, el qual no aprofundirem en aquest projecte, però si explicarem el codi que hem utilitzat per obtenir el nostre gràfic en el capítol 4, Implementació (4.10. Obtenció de les Corbes de Nivell Utilitzant el Matlab).

Figura 3.16. Finestra inicial de Matlab.

Part ⇐ ConvertirEnterACaràcter(Número)

FI_SI

FI
La figura 3.16 ens mostra la pantalla inicial que ens apareix quan obracem el Matlab. A partir d’aquí és on hem de generar el nostre codi de programació basat en Matlab i així poder llegir les nostres imatges del reflector i les seves coordenades.

3.11.1. Obtenció de les corbes de nivell

Per obtenir les corbes de nivell de la intensitat de llum que ofereix el nostre reflector ens cal crear, primerament, un algorisme que llegeixi les imatges i les seves coordenades.

**ALGORISME LlegirImatgesICoordenades(ruta,nom)**

**ENTRADA:** Passem a la funció una cadena de caràcters que indicarà un ruta de directori existent al nostre ordinador i el nom del fitxer on guardem les coordenades.

**SORTIDA:** Nom de totes les imatges llegides amb les coordenades corresponents, en forma de matriu.

**INICI**

FitxerText ← Concatenar(ruta,nom)
FitxerId ← ObrirFitxer(FitxerText)
[Nom, Count] ← llegir(FitxerId)
TancarFitxer(FitxerId)

FOR i=1 TO Count

NomImatge ← LlegirImatgeJPG(nom(i))
ImatgeTemporal ← Passar_de_RGB_a_GRAY(NomImatge)
A(i).ValorImatge ← Sum(ImatgeTemporal) / 255
A(i).ValorCoor ← Count(i)

FI_FOR

Si ens fixem en l’algorisme anterior, s’encarrega de emmagatzemar les intensitats de totes les imatges que es trobin en el directori que li hem passat per paràmetre, i relacionar-les amb les coordenades que tenim entrades a l’arxiu de text, que també li hem passat per paràmetre. El bucle FOR es repeteix tantes vegades com imatges estiguin contingudes dins el fitxer de text. És aquí on es crea la matriu “A” (en sentit de Matlab), que contindrà el valor de la intensitat que genera la imatge i les seves coordenades. Cal dir que per a la obtenció de la intensitat de la imatge hem hagut de convertir la nostre imatge de tres canals RGB en una imatge nivells de grisos; d’aquesta manera quan preнем el valor de la imatge de grisos i la dividim entre 255 obtenim el valor de la intensitat del reflector, normalitzada. Aquests valors (pixels) els sumem per obtenir la intensitat de llum final.

Un cop tenim emmagatzemades les intensitats amb les seves corresponents coordenades, hem de crear una gràfica per a poder estudiar els diferents punts d’intensitat que ofereix el reflector en les diferents coordenades. El codi és el següent:

**ALGORISME CrearGrafica2D(imatge)**

**ENTRADA:** Passarem per paràmetre la “A” que hem creat a l’algorisme anterior, la qual conté el nom de la imatge i les seves coordenades.
INICI
Count ← Longitud(imatge)
X ← Inicialitzarmatriu(l, count)
Y ← Inicialitzarmatriu(l, count)
Z ← Inicialitzarmatriu(l, count)
FOR i = 1 TO count
    X(l,i) ← imatge (i). Valorcoor(1)
    Y(l,i) ← imatge (i). Valorcoor(2)
    Z(l,i) ← imatge (i). ValorImatge
FI FOR
DibuixarGràfica(X, Y, Z)
FI

Tal i com hem vist a l’algorisme creem tres matrius (X, Y, Z) on ens serviran per guardar cadascun dels valors de X i de Y, i en la matriu Z col·locarem el valor de la intensitat que ens ha proporcionat la imatge. Una vegada tinguem totes les coordenades i valors carregats en les matrius, ja podem dibuixar la gràfica (veure figura 3.17).
4 IMPLEMENTACIÓ
4. Implementació

Dins d’aquest capítol es pretén aprofundir en els aspectes de disseny més importants. Doncs, farem una ampliació d’alguns aspectes del capítol de disseny, mostrant trossos de codi emprat en el projecte.

Mostrarem primer com carregar la boca del reflector, seguidament crearem el pla sota el reflector, el qual simularà la superfície del carrer. Inserirem un llum sota el reflector i finalment realitzarem la renderització de les imatges que volem obtenir mitjançant la càmera. La càmera es creadrà anteriorment a la renderització i es mourà sobre la superfície creada.

4.1. Lectura del fitxer DXF

Com ja hem comentat al capítol 3, Disseny, el fitxer DXF es tracta d’un fitxer ASCII, per tant és pot obrir i consultar el seu codi mitjançant qualsevol bloc de notes de qualsevol sistema operatiu, simplement ens cal cercar el què ens interessa. Recordem que estem buscant posicions de punts (vèrtexs) i, al mateix temps, saber si entre vèrtex i vèrtex existeix una comanda de línia o un arc.

```python
file = open(filename, "r")
continuar = "Cert"
while (continuar == "Cert"):
    Llegim_Parell_linies(file)
    index=index+1
    if (linea == "ENTITIES"):
        continuar = "False"
Llegim_Parell_linies(file)
print linea
index += 1
x=0
while ((linea!="ENDSEC") and (linea!="EOF")):
    if (linea!="ENDSEC"):
        if (linea == "LINE"):
            processLineEntity(file)
        elif (linea == "ARC"):
            processArcEntity(file)
        else:
            Llegim_Parell_linies(file)
            index += 1
print linea
file.close()
```

Com podem comprovar, en el petit fragment de codi que hem extret del procés de lectura d’un arxiu DXF, s’obre el fitxer mitjançant la funció `open` la qual li passem el nom del fitxer que volem obrir i el paràmetre “r”, que indica que obrim el fitxer com a lectura.
Una vegada tenim el fitxer obert podem llegir línia a línia el seu contingut així que amb la instrucció `Llegim_Parell_Linies(file)` obtenim la lectura de dues línies a la vegada, portant a terme la seva concatenaició, d’aquesta manera busquem la paraula `ENTITIES`, ja que a partir d’aquesta, les següents línies que la precedeixen contenen informació per a crear la boca del nostre reflector. Quan hem trobat la paraula `ENTITIES` sabem del cert que les dues següents línies han de portar la paraula `LINE` o `ARC` depenent de si conté una línia o un arc, respectivament. Podem trobar-nos el cas de què estiguem a final de fitxer o a final de secció (secció `Entities`), per això ho controlem amb la condicional iterativa “while ((linea!="ENDSEC") and (linea!="EOF”)): “. En el cas de què ens trobem una línia cridarem a la funció `ProcessLineEntity(file)` i la tractarem com a tal. En cas de què ens trobem un arc cridarem a la funció `ProcessArcEntity(file)`.

Si no es tracta d’una línia ni d’un arc continuarem llegint el fitxer DXF i així successivament fins que arribem a final de fitxer. Un cop al final del fitxer ens cal, simplement, tancar-lo amb la funció `file.close()`.

4.2. Preparació de l’algorisme de Graham Scan

L’algorisme de Graham Scan realitza el càlcul de l’envolvent convexa però, per aconseguir aquest càlcul, hem d’ordenar els elements d’aquesta envolvent seguint un criteri determinat i a més hem de partir de l’element més petit sobre l’eix y i més petit en l’eix x, que es troba dins de l’envolvent.

El primer pas que hem de fer per iniciar un algorisme de Graham Scan és cercar l’element més petit de l’envolvent convexa. A partir de la matriu que conté els elements que hem llegit del fitxer DXF hem de cercar quin d’ells és el més petit. Aquest és un pas important perquè depenent d’aquest vèrtex l’algorisme Graham Scan es desenvoluparà amb èxit o no.

```python
def FindLowest(matriu):
    n=matriu.element
    m=0
    i=1
    for i in range (1,n):
        if (matriu.Matriu_Vertex_Boca[i][1] < matriu.Matriu_Vertex_Boca[m][1]):
            Swap(0,i,matriu)
        else:
            if (matriu.Matriu_Vertex_Boca[i][1] == matriu.Matriu_Vertex_Boca[m][1]):
                if (matriu.Matriu_Vertex_Boca[i][0] < matriu.Matriu_Vertex_Boca[m][0]):
                    Swap(0,i,matriu)
                else:
                    Swap(0,m,matriu)
```

UdG 84
La funció *FindLowest(matriu)* és l’encarregada de realitzar la cerca d’aquest element. Li passem la matriu per paràmetre i recórrer tots els elements de la matriu, deixant l’element més petit a la primera posició. El paràmetre matriu és de la classe Matriu. Per accedir als elements que conté cal fer referència a la seva component Matriu_Vertex_Boca que es tracta d’una estructura que conté tots els vèrtexs de la boca del reflector. Recordem que cada vèrtex conté tres components, així doncs, per accedir al primer component del primer vèrtex utilitzariem matriu.Matriu_Vertex_Boca[0][0], a la segona component del primer vèrtex matriu.Matriu_Vertex_Boca[0][1], i a la tercera component del primer vèrtex matriu.Matriu_Vertex_Boca[0][2]. L’ordre de les components és x, y, z fent referència a 0, 1, 2 respectivament.

Si ens fixem al codi mostrat anteriorment, veurem que s’utilitza el bucle *for-in* per a recórrer tota la matriu i sempre fem comparacions amb el primer element de la matriu i amb l’element que estem posicionats en aquell moment. Per comprovar si l’element és més petit utilitzem la sentència condicional *if* (matriu.Matriu_Vertex_Boca[i][1] < matriu.Matriu_Vertex_Boca[m][1]) on m sempre fa referència a la posició 0 de la matriu. Primerament sempre tenim en compte l’eix y. En cas de què trobessim dos vèrtexs amb components y iguals hauríem de comprovar l’eix x i agafar el vèrtexs amb la x més petita dels dos. Una vegada hem trobat l’element més petit ens falta fer l’intercanvi d’aquest cap a la posició més petita de la matriu; aquest procés el realitzem mitjançant la funció *Swap(0,i,matriu)* on fa l’intercanvi del vèrtex que es troba a la posició “i” amb el vèrtex que es troba a la posició “0” de la matriu.

Quan ja tenim el vèrtex més petit situat a la primera posició de la matriu on hi guardem els vèrtexs que formen la boca del reflector (l’envolvent convexa), podrem començar a reordenar tots els nostres elements, tret del primer element de la matriu, segons l’angle que formen entre aquest i la línia horitzontal. La seva ordenació serà de menor a major.

```python
def Preparar_Unio_De_Vertex(matriu):
    LowestPoint = [[0,0,0]]
    LowestTangent = round(float(0),8)
    LowTangent = round(float(0),8)
    LowestPosition = 0
    IndexOrdenat=0

    NombreTotalVertex = matriu.element
    for i in range(1,NombreTotalVertex):
        LowestTangent = PolarComp(matriu.Matriu_Vertex_Boca[0],
                                matriu.Matriu_Vertex_Boca[i])
        LowestTangent = round(float(LowestTangent),8)
        LowestPosition = i

    for j in range(i+1,NombreTotalVertex):
        LowTangent = PolarComp(matriu.Matriu_Vertex_Boca[0],
                               matriu.Matriu_Vertex_Boca[j])
```

UdG 85
$$\text{LowTangent} = \text{round}(\text{float(LowTangent)},8)$$

$$\text{if (LowTangent} < \text{LowestTangent):}$$
$$\text{LowestTangent} = \text{LowTangent}$$
$$\text{LowestPosition} = j$$

$$\text{Swap(i,LowestPosition,matriu)}$$

La matriu com a paràmetre que li passem, tal i com observem en el codi anterior, ja conté l’element més petit a la primera posició. Com que necessitem recórrer tota la matriu utilitzem el bucle \textbf{for-in} que va des de la posició 1 fins al número total de vèrtexs entrats a la matriu; aquests els obtenim mitjançant \texttt{matriu.element} que ens retorna el número d’elements que conté la matriu.

Sempre estem comprovant resultats entre dos vèrtexs, entre el vèrtex situat a la primera posició de la matriu i la resta de vèrtexs. Primerament necessitem l’angle polar entre ells que ens ho calcula la funció \texttt{PolarComp(matriu.Matriu_Vertex_Boca[0], matriu.Matriu_Vertex_Boca[j])} on “j” ens indica la posició de la matriu on estem situats. Els valors resultants de la funció \texttt{PolarComp} són els que farem la comprovació i anirem guardant la posició del valor més petit així quan arribem al final del bucle haurem obtingut la posició del més petit. Llavors només ens caldrà intercanviar els vèrtexs amb la instrucció \texttt{Swap(i,LowestPosition,matriu)} i tornar a buscar el següent valor més petit que el precedeix i així successivament fins que tinguem tots els valors reordenats de la matriu.

Cal mostrar també el codi que amaga la funció \texttt{polarAngle} i que es crida des de la funció que hem anomenat abans \texttt{PolarComp}.

\texttt{def polarAngle(X,Y):}
\texttt{    if (X == 0) and (Y == 0):}
\texttt{        return (-1)}
\texttt{    if (X == 0):}
\texttt{        if (Y>0):}
\texttt{            return (90)}
\texttt{        elif (Y<=0):}
\texttt{            return (270)}
\texttt{        theta = Atan(X,Y)}
\texttt{        theta = theta * (360/(2*pi))}
\texttt{    if (X>0):}
\texttt{        if (Y>=0):}
\texttt{            return(theta)}
\texttt{        elif (Y<0): return (theta + 360)}
\texttt{    elif (X<=0): return(theta + 180)}

A la funció \texttt{polarAngle(X,Y)} simplement li passem el valor de X i el valor de Y obtinguts en el càlcul:

\[
X = \text{Vector}_B[0] - \text{Vector}_A[0] \\
\]
Quan diem Vector ens referim a un vèrtex pertanyent a la boca del nostre reflector. La probabilitat que la X i la Y ens donin 0 és molt baixa. Això voldria dir que els dos vèrtexs que comparem són iguals. En el cas de què la component X sigui igual entre els dos vèrtexs tindriem dues possibilitats: que l’angle format sigui 90° o que sigui 270°. El cas més normal que podem trobar-nos és haver de calcular l’arctangent de l’angle amb la funció predefinida pel mòdul math de Python Atan(X,Y) i realitzar el següent càlcul Atan(X,Y)∗(360/(2*pi)) on pi és una constant declarada en el mòdul math també. És aquí on podem donar diferents resultats depenen del quadrant on es trobi l’angle: return(theta), return (theta + 360), return(theta + 180); theta és el valor que hem obtingut del càlcul anterior.

4.3. Desenvolupament de la funció Graham Scan

Havent seguit els passos anteriors correctament, ja estem apunt per realitzar la crida a la funció Graham_Scan. En aquests moments ja tindrem la matriu ordenada segons els angles.

```python
def Graham_Scan (matriu_vella, matriu_nova):
    Crear_Matriu_Boca(matriu_nova, matriu_vella.Matriu_Vertex_Boca[0][0], matriu_vella.Matriu_Vertex_Boca[0][1], matriu_vella.Matriu_Vertex_Boca[0][2])
    Crear_Matriu_Boca(matriu_nova, matriu_vella.Matriu_Vertex_Boca[1][0], matriu_vella.Matriu_Vertex_Boca[1][1], matriu_vella.Matriu_Vertex_Boca[1][2])
    i=2
    while (i < matriu_vella.element):
        o = Cross_Product(matriu_nova.Matriu_Vertex_Boca[matriu_nova.element-2], matriu_nova.Matriu_Vertex_Boca[matriu_nova.element-1], matriu_vella.Matriu_Vertex_Boca[i])
        if (o > 0):
            Crear_Matriu_Boca(matriu_nova, matriu_vella.Matriu_Vertex_Boca[i][0], matriu_vella.Matriu_Vertex_Boca[i][1], matriu_vella.Matriu_Vertex_Boca[i][2])
            j=i
            while ((o > 0) and (j < matriu_vella.element)):
                o = Cross_Product(matriu_nova.Matriu_Vertex_Boca[matriu_nova.element-2], matriu_nova.Matriu_Vertex_Boca[matriu_nova.element-1], matriu_vella.Matriu_Vertex_Boca[j])
                j = j + 1
        elif (o < 0):
            Eliminar_Ultim_Vertex(matriu_nova)
```

UdG
Capítol 4. Implementació

\[ \text{Crear_Matriu_Boca}(\text{matriu_nova}, \text{matriu_vella} . \text{Matriu_Vertex_Boca}[i][0], \text{matriu_vella} . \text{Matriu_Vertex_Boca}[i][1], \text{matriu_vella} . \text{Matriu_Vertex_Boca}[i][2]) \]

\[ i = i + 1 \]

\[ o = \text{Cross_Product}(\text{matriu_nova} . \text{Matriu_Vertex_Boca}[\text{matriu_nova} . \text{element}-2], \text{matriu_nova} . \text{Matriu_Vertex_Boca}[\text{matriu_nova} . \text{element}-1], \text{matriu_vella} . \text{Matriu_Vertex_Boca}[\text{matriu_vella} . \text{element}-1]) \]

La funció \textit{Graham_Scan} retorna la matriu amb els únics vèrtexs necessaris per a realitzar la unió de cadascun d’ells i formar la boca del reflector. Així doncs, crearem una nova matriu a partir de la matriu que ja teníem.

Aquesta funció, tal i com podem comprovar en el codi anterior, passa els dos primers elements de la matriu vella cap a la matriu nova directament.

Abans de continuar mirant el codi de la funció \textit{Graham_Scan} fem un incís a la funció \textit{Cross_Product} que és cridada vàries vegades des de dins de \textit{Graham_Scan}.

\begin{verbatim}
def Cross_Product(P1,P2,P3):
    return ((P2[0] - P1[0]) * (P3[1] - P1[1]) - (P3[0] - P1[0])*(P2[1] - P1[1]))
\end{verbatim}

La funció \textit{Cross_Product} realitza el producte vectorial de la diferència de vector. Podem veure un clar exemple en la \textit{figura 4.1}.

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{cross_product.png}
  \caption{Es mostra la operació que realitza la funció Cross_Product.}
\end{figure}

L’algoritme de \textit{Graham Scan} considera que els punts es troben ordenats a la matriu de manera seqüencial. Per cada punt, es pot determinar, mitjançant els seus dos punts anteriors si hem de girar a l’esquerra o hem de girar a la dreta. Si girem a la dreta voldrà dir que l’últim punt no forma part de l’envolvent convexa i, per tant, es pot descartar. Si girem cap a l’esquerra, el punt serà acceptat i l’algoritme passarà al següent punt de la matriu. Si trobem dos punts que coincideixen en la posició podem optar en quedar-nos els dos punts o en descartar un punt. Nosaltres hem preferit quedar-nos amb només un punt coincident.

S’ha de determinar si els tres punts constitueixen un gir cap a l’esquerra o un gir cap a la dreta sense requerir calcular l’angle entre les dues línies del segment i poden ser guardats (veure \textit{figura 4.2}).
Pels tres punts \((P_1_x, P_1_y), (P_2_x, P_2_y)\) i \((P_3_x, P_3_y)\), simplement cal calcular el producte vectorial entre \((P_2_x - P_1_x)*(P_3_x - P_1_y) - (P_2_y - P_1_y)*(P_3_x - P_1_x)\) dels dos vectors definits pels punts \((P_1_x, P_1_y), (P_2_x, P_2_y)\) i \((P_2_x, P_2_y), (P_3_x, P_3_y)\). Si el resultat és zero, els punts coincidiran; si és positiu, el tercer punt constitueix un gir a l’esquerra, altrament constitueix un gir a la dreta.

Tornant a la funció **Graham_Scan** farem referència al resultat de la funció **Cross_Product** perquè si aquest resultat és positiu afegirem el vèrtex candidat a la nova matriu i en cas de què el resultat sigui negatiu voldrà dir que haurem de fer un pas enrera, eliminar l’últim element de la nova matriu i afegir-hi el nou vèrtex candidat a formar part de l’envolvent convexa.

### 4.4. Omlir l’interior de la boca del reflector amb punts

Amb el contorn de la boca del reflector no n’hi ha prou per poder completar el nostre reflector perquè necessitem una superfície on poder donar volum al nostre reflector. Per tant és important poder inserir punts a l’interior de la boca i així poder ajuntar tots aquests punts per crear una superfície.

```python
def CrearPuntsInteriors2(matriu):
    comptador=0
    matriu_nova = Matriu_Nova()
    pas= 100
    maximaY = int(Ymax(matriu))
    maximaX = int(Xmax(matriu))
    minimaY = int(Ymin(matriu))
    minimaX = int(Xmin(matriu))

    Limit_Esq_Sota= [minimaX, minimaY, 0]
    Limit_Esq_Sobre = [minimaX, maximaY, 0]
    Limit_Dreta_Sota = [maximaX, minimaY, 0]
    Limit_Dreta_Sobre = [maximaX, maximaY, 0]
```

}\textbf{Figura 4.2.} De A a B i de B a C van en sentit contrari a la agulles del rellotge, però de C a D, no. L’algorisme detecta aquesta situació i descarta qualsevol segment fins que no tornem a trabar cap segment en sentit contrari a les agulles del rellotge.
Capítol 4. Implementació

for y in range(minimaY, maximaY, pas):
    for x in range(minimaX, maximaX, pas):
        vector = [x,y,0]
        for i in range (0, matriu.element-1):
            if (InterseccioVertical(matriu.Matriu_Vertex_Boca[i],
                matriu.Matriu_Vertex_Boca[i+1],vector) == "True"):
                comptador=comptador+1

if (InterseccioVertical(matriu.Matriu_Vertex_Boca[0],
    matriu.Matriu_Vertex_Boca[matriu.element-1],vector)=="True"):
    comptador=comptador+1
if ((comptador % 2)==0):
    print "El vector ",vector," NO pertany a la boca."
else:
    if (Existeix_Vertex_Boca(matriu,vector[0],vector[1],vector[2] ) ==
        "False"):
        Crear_Matriu_Boca(matriu_nova,vector[0],vector[1],vector[2])

comptador=0
Natejar_Matriu(matriu)
Copiar_Matrius(matriu_nova,matriu)

A la funció li passem la matriu creada a partir de l’algorisme de Graham Scan, que conté els punts que componen el contorn de la boca del reflector i per tant tots els punts que hem de crear nous s’han de trobar a dins de la boca.

Tenim una variable anomenada comptador. Per saber si un punt es troba, o no, a l’interior de la boca del reflector haurem de tirar una línia vertical des del punt on comprovem si es troba a dins la boca. Si aquesta línia interseca amb el contorn de la boca un número parell de vegades o cap vegada voldrà dir que el punt es troba a fora de la boca (veure figura 4.3); en cas que la línia intersequi un número senar de vegades voldrà dir que el punt es troba a l’interior de la boca (veure figura 4.4). Així doncs la variable comptador serà l’encarregada de comptar quantes vegades interseca amb el contorn.

**Figura 4.3.** A partir del punt C tirem una línia vertical; si aquesta interseca amb el contorn de la boca un nombre senar de vegades voldrà dir que el punt pertany a l’interior de la boca.

**Figura 4.4.** A partir del punt C tirem una línia vertical; si aquesta interseca amb el contorn de la boca un nombre parell de vegades voldrà dir que el punt pertany a l’interior de la boca.
Eina pel desenvolupament de reflectors de conjunts òptics

A partir de la matriu que guarda els vèrtexs del contorn de la boca podem extreure els següents valors:

\[
\begin{align*}
\text{maximaY} & = \text{int}(Y_{\text{max}}(\text{matriu})) \\
\text{maximaX} & = \text{int}(X_{\text{max}}(\text{matriu})) \\
\text{minimaY} & = \text{int}(Y_{\text{min}}(\text{matriu})) \\
\text{minimaX} & = \text{int}(X_{\text{min}}(\text{matriu}))
\end{align*}
\]

D’aquesta manera aconseguim delimitar una àrea on col·locarem els punts que crearem. Una vegada tinguem aquests límits ja podem utilitzar dos bucles \texttt{for-in}; un serà l’encarregat de recórrer l’eix de les x i l’altre recorrerà l’eix de les y. La variable \texttt{pas} s’encarrega de deixar la distància entre punts creats.

A mesura que anem avançant amb el bucle \texttt{for-in} anem comprovant si existeix intersecció vertical utilitzant la instrucció \texttt{InterseccioVertical (matriu. Matriu_Vertex_Boca [i], matriu.Matriu_Vertex_Boca[i+1],vector) }. Com podem observar en el codi anterior, la comprovació de la intersecció vertical es realitza dins d’un tercer bucle \texttt{for-in} que recórrer tot el contorn de la boca contant el número d’interseccions que existeixen per a cada vector nou que vulguem crear. El final fem una comprovació sobre el valor del comptador i si aquest és parell voldrà dir que descartarem el vector, en cas contrari comprovarem que el vector no existeixi ja a la nova matriu i l’insereïm:

\[
\begin{align*}
\text{if } ((\text{comptador} \mod 2)==0): & \quad \text{print "El vector ",vector," NO pertany a la boca."} \\
\text{else:} & \quad \text{if (Existeix_Vertex_Boca(matriu,vector[0],vector[1],vector[2] ) == "False"): } \\
& \quad \text{Crear_Matriu_Boca(matriu_nova,vector[0],vector[1],vector[2])}
\end{align*}
\]

La funció que busca si existeix o no intersecció és la següent:

\[
\begin{align*}
def \text{InterseccioVertical}(P1,P2,P3):
& \quad \text{if } (P1[0]==P2[0]): \quad \text{return } (\text{"False"}) \\
& \quad \text{else:} \\
& \quad \quad t=(P3[0]-P1[0])/(P2[0]-P1[0]) \\
& \quad \quad \text{if } ((t<0) \text{ or } (t>1)): \quad \text{return } (\text{"False"}) \\
& \quad \quad \text{else:} \\
& \quad \quad \quad s = P1[1] \times (1-t) + P2[1] \times t - P3[1] \\
& \quad \quad \quad \text{if } (s>=0): \quad \text{return } (\text{"False"}) \\
& \quad \quad \text{else: } \quad \text{return } (\text{"True"})
\end{align*}
\]

Aquesta funció rep tres paràmetres. Cada paràmetre pertany a un vèrtexs. Els vèrtexs P1 i P2 formen part del contorn de la boca mentre que P3 és el vèrtex que volem inserir a l’interior de la boca. La funció retornarà cert o fals depenent de si existeix la intersecció o no, respectivament.
A continuació mostrem els càlculs matemàtics que hem utilitzat per saber si hi ha intersecció entre dues línies, tal i com es mostra en la figura 4.5.

\[
\]

\[
\begin{align*}
t & \in [0, 1] \\
S & \geq 0
\end{align*}
\]

\[
t = \frac{P_3(0) - P_1(0)}{P_2(0) - P_1(0)}
\]

\[\text{Figura 4.5. Procés per calcular si un vèrtex pertany a l’interior de la boca del reflector.}\]

### 4.5. Realització de la triangulació

La triangulació es realitza una vegada tenim tots els vèrtexs, tant els que formen el contorn de la nostra boca com els que es troben al seu interior en una sola matriu. A la funció, però, és necessari distingir els punts que formen el contorn de la boca i els punts que es troben a l’interior d’aquest contorn. Per això passarem com a paràmetres de la funció la matriu amb només el contorn de la boca i la matriu amb els tots els punts interiors.

```python
def Triangulacio (matriu_contorn, matriu_interior):
    llargadaN=0
    matriu_nova = Matriu_Nova()
    pas= 100
    maximaY = int(Ymax(matriu_contorn))
    maximaX = int(Xmax(matriu_contorn))
    minimaY = int(Ymin(matriu_contorn))
    minimaX = int(Xmin(matriu_contorn))
    Limit_Esq_Sota= [minimaX, minimaY, 0]
    Limit_Esq_Sobre = [minimaX, maximaY, 0]
    Limit_Dreta_Sota = [maximaX, minimaY, 0]
    Limit_Dreta_Sobre = [maximaX, maximaY, 0]
```

UdG
Eina pel desenvolupament de reflectors de conjunts òptics

Area_Triangulacio=Vector_Triangulacio()
Punts_Triangulacio=Matriu_Triangulacio()
Llista_Triangles=Matriu_Nova
Trident=Triangle()
posi=0

for y in range(minimaY, maximaY, pas):
    llargadaN=0
    for x in range(minimaX, maximaX, pas):
        llargadaN=llargadaN+1
        vector1 = [x,y,0]
        vector2 = [x+pas,y,0]
        vector3 = [x,y+pas,0]
        vector4 = [x+pas,y+pas,0]

        if (Existeix_Vertex_Boca(matriu_interior,x,y,0)=="True"):
            a=1
        else: a=0
        if (Existeix_Vertex_Boca(matriu_interior,x+pas,y,0)=="True"):
            b=1
        else: b=0
        if (Existeix_Vertex_Boca(matriu_interior,x,y+pas,0)=="True"):
            c=1
        else: c=0
        if (Existeix_Vertex_Boca(matriu_interior,x+pas,y+pas,0)=="True"):
            d=1
        else: d=0

    Area_Triangulacio=[a,b,c,d]

    if (Hi_Ha_Uns_Vector(Area_Triangulacio)>0):
        interseccions(vector1,vector2,vector3,vector4,matriu_contorn,matriu_interior,Punts_Triangulacio,Area_Triangulacio,Trident,Llista_Triangles)

    posi=posi+1
    print "Posicio principal: ",posi

Mostrar_Matriu_Boca(Punts_Triangulacio)
Dibuixar_Punts(Punts_Triangulacio)
print "Llargada N= ",llargadaN
print "Mostrem la llista de Triangles: 
Mostrar_Llista_Triangles(Llista_Triangles)
UnioTriangulacio(Llista_Triangles)
print "Mostrem la matriu_interior:
Mostrar_Matriu_Boca(matriu_interior)

Com es pot observar en el codi anterior, primerament es crea una nova matriu
matriu_nova = Matriu_Nova() i es delimita l’àrea de la boca del reflector:
Creem també un nou element de la classe `Vector_Triangulacio()` anomenat `Area_Triangulacio`. Aquesta `Area_Triangulacio` contindrà els vèrtexs que voldrem triangular. Algun d’ells pot ser que no pertanyin a l’interior de la boca però com que els vèrtexs els agafem de quatre en quatre podrem tenir setze combinacions. Aquesta variable conté quatre posicions on cada una d’aquesta posició indica si el vèrtex pertany o no pertany a la boca.

També creem un nou element de la classe `Matriu_Triangulacio()` anomenat `Punts_Triangulacio`. Dins aquesta variable hi guardarem tots els punts candidats a ser triangularitzats, es trobin, o no, a l’interior de la boca del reflector.

Així doncs utilitzarem dos bucles `for-in`, un imbricat amb l’altre perquè el què es tracta de fer és recórrer tota l’àrea limitada que hem creat anteriorment, així que un `for-in` recorrerà l’eix de les X i l’altre `for-in` recorrerà l’eix de les Y.

Una vegada ens trobem a dins del `for-in` interior, ja tenim el valor de X i el valor de Y, per tant ja podem crear els quatre vèrtexs candidats a ser triangularitzats:

```
vector1 = [x,y,0]
vector2 = [x+pas,y,0]
vector3 = [x,y+pas,0]
vector4 = [x+pas,y+pas,0]
```

La variable pas indica l’espai que deixem entre vèrtex i vèrtex creat. La component z pren sempre el valor 0. Un cop tinguem els quatre vèrtex creats hem de comprovar si es troben a la `matriu_interior`, és a dir, a la llista de vèrtex que es troben a l’interior de la boca del reflector: 

```
if (Existeix_Vertex_Boca (matriu_interior,x,y,0) == "True"):
```

La funció `Existeix_Vertex_Boca` simplement ens fa una cerca del vèrtex a la matriu; si aquest vèrtex existeix la funció ens retornarà `True`, en cas contrari ens retornarà `False`. Aquesta comprovació l’hem de realitzar per cadascun dels quatre vèrtexs i així formarem la `Area_Triangulació=[a,b,c,d]`. a, b, c i d contindran valors 0 en cas de que el vèrtex no es trobi a l’interior de la boca, o 1 en cas de que el vèrtex pertanyi a l’interior de la boca. D’aquesta manera aconseguim saber quin cas tenim davant a l’hora de fer la seva triangulació (veure el capítol 3 de disseny, 3.6.2. El procés de la triangulació). Si ens fixem els vèrtexs que agafem no vénem donats en ordre seguits, un darrera l’altre, sinó que es disposen en forma de quadrats tal i com podem veure en la figura 4.6.

![Figura 4.6](image)
Eina pel desenvolupament de reflectors de conjunts òptics

Observant la posició dels vèrtexs, tal i com es troba a la figura 4.6, veiem que els vèrtexs pertanyen a l’interior de la boca i per tant ens trobariem davant del cas 1111. En aquest cas podem extreure dos triangles.

Amb la instrucció if (Hi_Ha_Un_Vector(Area_Triangulacio)>0) comprovem si existeix cap “1” dins de la variable Area_Triangulacio. En cas afirmatiu haurem de buscar quin cas de triangulació tenim i realitzar la seva triangulació. Per tant ja podrem construir el triangle i això ho fem amb la funció Interseccions. En el codi on mostrem la funció de triangulació veiem la instrucció on fa referència a la crida de la funció Interseccions:

interseccions(vector1,vector2,vector3,vector4, matriu_contorn, matriu_interior, Punts_Triangulacio, Area_Triangulacio, Trident, Llista_Triangles)

Aquesta funció conté un codi molt extens així que el comentarem una vegada acabem de comentar la funció Triangulacio, en aquest mateix capítol d’implementació.

Finalment quan sortim del bucle for-in ja tenim tots els vèrtexs comprovats i tots els triangles calculats, ja sols ens queda dibuixar els vèrtexs a la superfície de Blender amb la instrucció Dibuixar_Punts(Punts_Triangulacio) i fer-ne la seva unió amb la instrucció UnioTriangulacio(Llista_Triangles).

Podem veure el codi amagat dins la funció Dibuixar_Punts seguidament:

def Dibuixar_Punts(matriu):
    for i in range(0,matriu.element):
        vertex=NMesh.Vert(matriu.Matriu_Vertex_Boca[i][0],
        matriu.Matriu_Vertex_Boca[i][1],matriu.Matriu_Vertex_Boca[i][2])
        poligon2.verts.append(vertex)

El codi que observem anteriorment recorre tota la matriu afegint cada vèrtex que conté a la interfície de Blender amb la funció poligon2.verts.append(vertex) on vertex és el punt amb la posició que volem inserir, poligon2 és el nom de l’objecte, verts fa referència a la classe i append és la funció per afegir vèrtexs.

Podem veure el codi dins la funció UnioTriangulacio seguidament:

def UnioTriangulacio(Llista_Triangles):
    for i in range (0,Llista_Triangles.element):
        cara=NMesh.Face()
        cara.append(polinog2.verts[Llista_Triangles.Matriu_Vertex_Boca[i][0]])
        cara.append(polinog2.verts[Llista_Triangles.Matriu_Vertex_Boca[i][1]])
        cara.append(polinog2.verts[Llista_Triangles.Matriu_Vertex_Boca[i][2]])
        polinog2.faces.append(cara)

El paràmetre que li passem a la funció conté la llista de triangles que s’han de formar. Els triangles es formen amb la unió de tres vèrtexs, així que com que ja tenim
els vèrtexs entrats ordenadament dins la variable `Llista_Triangles` simplement hem de recórrer tots els vèrtexs que estiguin continguts dins seu i formant-ne cares mitjançant la funció `append` de la classe `NMesh`.

Amb la instrucció `cara=NMesh.Face()` el què estem fent és crear una classe `Face` i així li podrem aplicar una sèrie de funcions determinades com és el cas de `append`. Quan tinguem la cara creada és necessari afegir-la al nostre objecte `polygon2` amb la instrucció `polygon2.faces.append(cara)`.

Ara veurem on realment es realitza el procés elaborat de la triangulació que es troba darrera la funció `interseccions`.

```python
def Interseccions (punt1,punt2,punt3,punt4,matriu_contorn, matriu_intern, matriunova, nombre_uns, Trident, Llista_Triangles):
    b1=[0,0,0]
    b2=[0,0,0]
    posicio=0
    for i in range (0,matriu_contorn.element-1):
        b1=matriu_contorn.Matriu_Vertex_Boca[i]
        b2=matriu_contorn.Matriu_Vertex_Boca[i+1]
        if (Hi_Ha_Uns_Vector(nombre_uns)==4):
            if ((Existeix_Vertex_Boca (matriu_intern,punt1[0],punt1[1], punt1[2])
            == "True") and (Existeix_Vertex_Boca (matriunova , punt1[0], punt1[1], punt1[2])
            == "False"):
                Crear_Matriu_Boca(matriunova,punt1[0],punt1[1],punt1[2])
            if ((Existeix_Vertex_Boca (matriu_intern, punt2[0], punt2[1], punt2[2])
            == "True") and (Existeix_Vertex_Boca (matriunova, punt2[0], punt2[1],
            punt2[2]) == "False"):
                Crear_Matriu_Boca(matriunova,punt2[0],punt2[1],punt2[2])
            if ((Existeix_Vertex_Boca (matriu_intern, punt3[0], punt3[1], punt3[2])
            == "True") and (Existeix_Vertex_Boca (matriunova, punt3[0], punt3[1],
            punt3[2]) == "False"):
                Crear_Matriu_Boca(matriunova,punt3[0],punt3[1],punt3[2])
            if ((Existeix_Vertex_Boca (matriu_intern, punt4[0], punt4[1], punt4[2])
            == "True") and (Existeix_Vertex_Boca (matriunova, punt4[0], punt4[1],
            punt4[2]) == "False"):
                Crear_Matriu_Boca(matriunova,punt4[0],punt4[1],punt4[2])
```

En aquest primer cas podem veure com el codi comprova la quantitat d’uns que apareixen a la variable `nombre_uns` per determinar de quin cas es tracta. Tal i com veiem a la figura del nostre cas trobem que els quatre vèrtexs es troben a l’interior de la boca del reflector.
En el següent cas ens trobem amb tres vèrtex pertanyents a l’interior de la boca del reflector i el vèrtex 1 pertanyent a l’exterior de la boca. La línia que talla la nostra figura correspon al contorn de la boca del reflector.

```python
if (Hi_Ha_Uns_Vector(nombre_uns)==3):
    if (area==[0,1,1,1]):
        interseccio=0

        if (HiHa_InterVertical(punt1,punt3,b1,b2)<"False"):
            t=HiHa_InterVertical(punt1,punt3,b1,b2)
            a_int_x = punt1[0] - punt1[0] * t + punt3[0] * t
            punt_inter1 = [a_int_x,a_int_y,0]
            puntvertical = punt_inter1

            if (Existeix_Vertex_Boca (matriunova, punt_inter1[0],
                punt_inter1[1], punt_inter1[2]) == "False"):
                Crear_Matriu_Boca(matriunova,punt_inter1[0],punt_inter1[1],punt_inter1[2])
                interseccio=1

        if (HiHa_InterHorizontal(punt1,punt2,b1,b2)<"False"):
            t = HiHa_InterHorizontal(punt1,punt2,b1,b2)
            a_int_x = punt1[0] - punt1[0] * t + punt2[0] * t
            punt_inter1 = [a_int_x,a_int_y,0]
            punthorizontal = punt_inter1

            if (Existeix_Vertex_Boca (matriunova, punt_inter1[0],
                punt_inter1[1], punt_inter1[2]) == "False"):
                Crear_Matriu_Boca(matriunova,punt_inter1[0],punt_inter1[1],punt_inter1[2])
                interseccio=1

        if ((Existeix_Vertex_Boca (matriu_intern, punt2[0], punt2[1],
                punt2[2]) == "True") and (Existeix_Vertex_Boca (matriunova ,punt2[0], punt2[1], punt2[2]) =="False") and (interseccio==1)):
            Crear_Matriu_Boca(matriunova,punt2[0],punt2[1],punt2[2])

        if ((Existeix_Vertex_Boca (matriu_intern, punt3[0], punt3[1],
            punt3[2]) == "True") and (Existeix_Vertex_Boca (matriunova ,punt3[0], punt3[1], punt3[2]) =="False") and (interseccio==1)):
            Crear_Matriu_Boca(matriunova,punt3[0],punt3[1],punt3[2])
```
if ((Existeix_Vertex_Boca (matriu intern, punt4[0], punt4[1], punt4[2]) =="True") and (Existeix_Vertex_Boca (matriu nova, punt4[0], punt4[1], punt4[2]) =="False") and (interseccio==1)):
    Crear_Matriu_Boca(matriu nova,punt4[0],punt4[1],punt4[2])

Els paràmetres que li passem a la funció són els quatre vèrtexs: punt1, punt2, punt3 i punt4; els vèrtexs que formen el contorn de la boca i es troben dins la matriu anomenada matriu_contorn; els vèrtexs que formen la part interior de la boca i es troben dins la matriu_intern. El paràmetre matriu nova contindrà només els vèrtexs que s’hauran de unir entre ells per a fer la triangulació. El paràmetre nombre_uns és un vector de quatre posicions i ens indica si el punt es troba, o no, a l’interior de la boca del reflector mitjançant un ‘1’ o un ‘0’, respectivament. El paràmetre Trident conté tres vèrtexs i aquests formen, conjuntament, un triangle entre ells. El paràmetre Llista_Triangles conté els triangles que haurem d’afegir i la posició de cada vèrtexs on es troba de la llista del polígon de Blender.

La funció recorre tot els elements que formen el contorn de la boca a la recerca d’interseccions. Com podem observar en el codi anterior, utilitzem una bucle for-in per recórrer tota la matriu i mitjançant la variable area comprovem en quin cas de la triangulació ens trobem, depenent de quins vèrtexs es troben a l’interior de la boca. La funció que cridem Hi_Ha_Uns_Vector(area) ens retorna el número de uns que conté el vector. En el primer cas comprovem si existeixen quatre uns, si és així, inserim un per un, cada vector dels quatre a la nova matriu que estem creant i passem seguidament a llegir quatre vèrtexs nous. Hem de tenir en compte de comprovar cada vèrtex si existeix a la nova matriu que creem i així afegir-lo, o no.

En el cas que trobem tres uns dins el vector nombre_uns se’ns presenten diferents possibilitats: [0,1,1,1]; [1,0,1,1]; [1,1,0,1]; [1,1,1,0].

En la primera possibilitat [0,1,1,1] buscarem la intersecció vertical entre el punt1 i el punt3 mitjançant la funció HiHa_InterVertical(punt1,punt3,b1,b2) que ens retornarà el punt d’intersecció o False depenent de si hi ha intersecció o no. Si existeix intersecció haurem de calcular el punt d’intersecció de x i el punt d’intersecció de y:

\[
\begin{align*}
a_{\text{int}}_x &= p\text{unt1}[0] - p\text{unt1}[0]*t + p\text{unt3}[0]*t \\
a_{\text{int}}_y &= p\text{unt1}[1] - p\text{unt1}[1]*t + p\text{unt3}[1]*t \\
p\text{unt1} &= [a_{\text{int}}_x,a_{\text{int}}_y,0] \\
p\text{untvertical} &= p\text{unt1}
\end{align*}
\]

Haurem de comprovar, també, que el punt_inter1 o puntvertical no existeixi en la nostra matriu nova; si no existeix el creem. Tal i com hem buscat la intersecció vertical, haurem de buscar la intersecció horitzontal mitjançant la funció HiHa_InterHoritzontal(punt1,punt2,b1,b2) . Aquesta funció també ens retorna un valor si existeix intersecció o False en cas de que no existeixi intersecció.
Eina pel desenvolupament de reflectors de conjunts òptics

Els dos vèrtexs que queden (punt3 i punt4) els afegirem a la matriu nova tal qual, assegurant-nos prèviament que no es troben ja a la matriu.

La funció \texttt{HiHa\_InterVertical} comprova si hi ha intersecció entre els punts i el contorn que li passem per paràmetre. D’igual manera es calcularia la intersecció horitzontal però canviant la component x per a la y i a la inversa.

\begin{verbatim}
def HiHa\_InterVertical(punta1,punta2,contorn1,contorn2):
    if (contorn2[0] - contorn1[0] <> 0):
        s = round(float((punta1[0] - contorn1[0]) / (contorn2[0] - contorn1[0])),4)
    else: return ("False")

    if (punta2[1]-punta1[1]<0):
    else: return ("False")

    if (s>=0 and s<=1 and t>=0 and t<=1):
        return (t)
    else:
        return("False")
\end{verbatim}

Aquesta funció ens retornarà el valor de ‘t’ si “s” i “t” es troben entre 0 i 1. Els punts que li passem a la funció són vèrtexs amb component x, y, z. Punt1 i Punt2 són els vèrtexs que formen el quadrat Punt1, Punt2, Punt3, Punt4 mentre que Contorn1 i Contorn2 són els vèrtexs que formen el contorn de la boca.

\begin{verbatim}
if (area==[1,0,1,1]):
    interseccio=0
    if ((Existeix\_Vertex\_Boca (matriu\_intern, punt1[0], punt1[1], punt1[2]) == "True") and (Existeix\_Vertex\_Boca (matriu\_nova ,punt1[0], punt1[1], punt1[2]) =="False")):
        Crear\_Matriu\_Boca\_Tri(matriu\_nova,punt1[0],punt1[1],punt1[2])

    if (HiHa\_InterHoritzontal(punt1,punt2,b1,b2)<>"False"):
        t = HiHa\_InterHoritzontal(punt1,punt2,b1,b2)
        a\_int\_x = punt1[0] - punt1[0]*t + punt2[0]*t
        a\_int\_y = punt1[1] - punt1[1]*t + punt2[1]*t
        punt\_inter2=[a\_int\_x,a\_int\_y,0]
        punthoritzontal=punt\_inter2
        interseccio=1

    if (Existeix\_Vertex\_Boca(matriu\_nova, punt\_inter2[0], punt\_inter2[1], punt\_inter2[2]) == "False"):
        Crear\_Matriu\_Boca(matriu\_nova,punt\_inter2[0],punt\_inter2[1],punt\_inter2[2])
\end{verbatim}
abacaba
La variable \textit{intersecció} ens indica amb un 1 que ja hem trobat el punt d’intersecció i ja no ens cal comprovar cap més punt dels que queden en el contorn del reflector.

Ara toca anar a buscar la intersecció vertical amb la funció \textit{HiHa\_InterVertical (punt2, punt4, b1, b2)} la qual si ens retorna diferent a \textit{False} afegirem el punt d’intersecció després de comprovar que no es troba ja a la nostra matriu.

Finalment ja podem afegir la resta de vèrtexs (vèrtex 3 i vèrtex 4) comprovant però que no existeixin a la matriu.

if (area==[1,1,0,1]):
    interseccio=0
    if ((Existeix\_Vertex\_Boca(matriu\_intern,punt1[0],punt1[1],punt1[2])="True")
    and (Existeix\_Vertex\_Boca( matriu\_nova, punt1[0], punt1[1], punt1[2]) == "False"):
        Crear\_Matriu\_Boca(matriu\_nova,punt1[0],punt1[1],punt1[2])
    if ((Existeix\_Vertex\_Boca(matriu\_intern,punt2[0],punt2[1],punt2[2])="True")
    and (Existeix\_Vertex\_Boca (matriu\_nova, punt2[0], punt2[1], punt2[2]) == "False"):
        Crear\_Matriu\_Boca(matriu\_nova,punt2[0],punt2[1],punt2[2])
    if (HiHa\_InterVertical(punt1,punt3,b1,b2)<"False"):
        t=HiHa\_InterVertical(punt1,punt3,b1,b2)
        a\_int\_x= punt1[0] - punt1[0]*t + punt3[0]*t
        a\_int\_y= punt1[1] - punt1[1]*t + punt3[1]*t
        punt\_inter3=[a\_int\_x,a\_int\_y,0]
        punt\_vertical=punt\_inter3
        interseccio=1
        if ((Existeix\_Vertex\_Boca (matriu\_nova, punt\_inter3[0], punt\_inter3[1],
        punt\_inter3[2]) == "False")):
            Crear\_Matriu\_Boca(matriu\_nova,punt\_inter3[0],punt\_inter3[1],punt\_inter3[2])

    if (HiHa\_InterHoritzontal(punt3,punt4,b1,b2)<"False"):
        t = HiHa\_InterHoritzontal(punt3,punt4,b1,b2)
        a\_int\_x = punt3[0] - punt3[0]*t + punt4[0]*t
        a\_int\_y = punt3[1] - punt3[1]*t + punt4[1]*t
        punt\_inter3=[a\_int\_x,a\_int\_y,0]
        punt\_horitzontal=punt\_inter3
        interseccio=1
        if (Existeix\_Vertex\_Boca (matriu\_nova, punt\_inter3[0], punt\_inter3[1],
        punt\_inter3[2]) =="False"):
Crear_Matriu_Boca(matriunova, punt_inter1[0], punt_inter1[1], punt_inter1[2])

if ((Existeix_Vertex_Boca(matriu_intern, punt1[0], punt1[1], punt1[2]) == "True")
    and (Existeix_Vertex_Boca(matriunova, punt1[0], punt1[1], punt1[2]) == "False")
    and interseccio == 1):
    Crear_Matriu_Boca(matriunova, punt1[0], punt1[1], punt1[2])

Els dos primers vèrtexs (vèrtex 1 i vèrtex 2) els afegirem si aquests no existeixen a la matriu mentre que el vèrtex 3 n’haurem de comprovar la intersecció vertical entre ell i el vèrtex 1 perquè el vèrtex 3 no es troba dins la boca del reflector HiHa_InterVertical(punt1, punt3, b1, b2). Comprovarem que existeix Intersecció Vertical i després de calcular el punt d’intersecció vertical l’afegirem a la matriu.

Després toca buscar la intersecció horitzontal entre el vèrtex 3 i el vèrtex 4 HiHa_InterHoritzontal(punt3, punt4, b1, b2) i afegir el punt d’intersecció a la matriu. El vèrtex 4 s’afegeix amb la funció Crear_Matriu_Boca(matriunova, punt4[0], punt4[1], punt4[2]).

if (area == [1, 1, 1, 0]):
    interseccio = 0
    if ((Existeix_Vertex_Boca(matriu_intern, punt2[0], punt2[1], punt2[2]) == "True")
        and (Existeix_Vertex_Boca(matriunova, punt2[0], punt2[1], punt2[2]) == "False")):
        Crear_Matriu_Boca(matriunova, punt1[0], punt1[1], punt1[2])
    if ((Existeix_Vertex_Boca(matriu_intern, punt3[0], punt3[1], punt3[2]) == "True")
        and (Existeix_Vertex_Boca(matriunova, punt3[0], punt3[1], punt3[2]) == "False")):
        Crear_Matriu_Boca(matriunova, punt3[0], punt3[1], punt3[2])
    if (HiHa_InterHoritzontal(punt3, punt4, b1, b2) <> "False"):
        t = HiHa_InterHoritzontal(punt3, punt4, b1, b2)
        a_int_x = punt3[0] - punt3[0]*t + punt4[0]*t
        a_int_y = punt3[1] - punt3[1]*t + punt4[1]*t
        punt_inter4 = [a_int_x, a_int_y, 0]
        punthoritzontal = punt_inter4
        interseccio = 1
        if ((Existeix_Vertex_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2]) == "False")):
Crear_Matriu_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2])

if (HiHa_InterVertical(punt2, punt4, b1, b2)<> "False"):
    t = HiHa_InterVertical(punt2, punt4, b1, b2)
    a_int_x = punt2[0] - punt2[0] * t + punt4[0] * t
    punt_inter4 = [a_int_x, a_int_y, 0]
    puntvertical = punt_inter4
    interseccio = 1

if ((Existeix_Vertex_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2]) == "False")):
    Crear_Matriu_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2])

Aquest és l’últim cas en què podem formar tres triangles amb els quatre vèrtexs on el vèrtex 4 es troba fora de la boca i els vèrtexs 1, 2 i 3 es troben a dins de la boca. Així doncs, després de comprovar si els tres vèrtexs ja existeixen dins la matriu Existeix_Vertex_Boca(matriu_nova, punt1[0], punt1[1], punt1[2]) == "False") els afegirem en cas de què no existeixin Crear_Matriu_Boca(matriunova, punt1[0], punt1[1], punt1[2]).

Buscarem si existeix intersecció horitzontal entre el vèrtex 3 i el vèrtex 4, trobarem el punt on s’intersequen en cas afirmatiu i l’afegirem a la nostra matriu en cas de que aquest no l’haguéssim afegit anteriorment:

if (HiHa_InterHoritzontal(punt3, punt4, b1, b2)<> "False"):
    t = HiHa_InterHoritzontal(punt3, punt4, b1, b2)
    a_int_x = punt3[0] - punt3[0] * t + punt4[0] * t
    punt_inter4 = [a_int_x, a_int_y, 0]
    punthoritzontal = punt_inter4
    interseccio = 1

if ((Existeix_Vertex_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2]) == "False")):
    Crear_Matriu_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2])

Realitzarem els mateixos passos per a buscar la intersecció vertical entre el vèrtex 4 i el vèrtex 2.

Ara anem a veure els casos en què ens trobem dos vèrtexs a l’interior de la boca del reflector i dos vèrtexs a l’exterior.

if (Hi_Ha_Uns_Vector(area)== 2):

if (area==[0,0,1,1]):
    interseccio=0

if (HiHa_InterVertical(punt1,punt3,b1,b2)!="False"):
    t=HiHa_InterVertical(punt1,punt3,b1,b2)
    a_int_x= punt1[0] - punt1[0]*t + punt3[0]*t
    a_int_y= punt1[1] - punt1[1]*t + punt3[1]*t
    punt_inter1=[a_int_x,a_int_y,0]
    puntvertical1=punt_inter1
    interseccio=1

if (Existeix_Vertex_Boca (matriunova, punt_inter1[0], punt_inter1[1], punt_inter1[2])== "False"):
    Crear_Matriu_Boca(matriunova,punt_inter1[0],punt_inter1[1],punt_inter1[2])

if (HiHa_InterVertical(punt2,punt4,b1,b2)!="False"):
    t=HiHa_InterVertical(punt2,punt4,b1,b2)
    a_int_x= punt2[0] - punt2[0]*t + punt4[0]*t
    a_int_y= punt2[1] - punt2[1]*t + punt4[1]*t
    punt_inter2=[a_int_x,a_int_y,0]
    puntvertical2=punt_inter2
    interseccio=1

if (Existeix_Vertex_Boca (matriunova, punt_inter2[0], punt_inter2[1], punt_inter2[2])== "False"):
    Crear_Matriu_Boca(matriunova,punt_inter2[0],punt_inter2[1],punt_inter2[2])

if (((Existeix_Vertex_Boca (matriu_intern, punt3[0], punt3[1], punt3[2]) =="True") and (Existeix_Vertex_Boca (matriunova, punt3[0], punt3[1], punt3[2]) == "False") and interseccio==1):
    Crear_Matriu_Boca(matriunova,punt3[0],punt3[1],punt3[2])

if (((Existeix_Vertex_Boca (matriu_intern, punt4[0], punt4[1], punt4[2]) == "True") and (Existeix_Vertex_Boca (matriunova, punt4[0],punt4[1],punt4[2]) == "False") and interseccio==1):
    Crear_Matriu_Boca(matriunova,punt4[0],punt4[1],punt4[2])

Aquest primer cas en trobar dos vèrtexs a l’interior (vèrtex 1 i vèrtex 2) i dos vèrtexs a l’exterior (vèrtex 3 i vèrtex 4) de la boca del reflector fa que haguem de buscar interseccions verticals i no horitzontals.

Com que el vèrtex 1 es troba a l’exterior de la boca, l’haurem de substituir pel punt d’intersecció vertical. Primerament comprovarem que hi hagi intersecció HiHa_InterVertical(punt1,punt3,b1,b2) . Si aquesta funció ens retorna diferent a False
prendem el punt d’intersecció i l’afegirem a la matriu, comprovant abans que no existeixi ja a la matriu.

Seguidament tornarem a comprovar si existeix intersecció vertical però entre els vèrtexs 2 i 4 \( HiHa_{\text{InterVertical}}(\text{punt2, punt4, b1, b2}) \), prendem el punt d’intersecció (en cas que existeixi) i l’afegirem a la nostra matriu.

\[
\begin{align*}
\text{if (HiHa_{\text{InterVertical}}(\text{punt2, punt4, b1, b2}) <= "False")}: \\
t &= \text{HiHa_{\text{InterVertical}}(\text{punt2, punt4, b1, b2})} \\
a\_\text{int} \_\text{x} &= \text{punt2}[0] - \text{punt2}[0] * t + \text{punt4}[0] * t \\
a\_\text{int} \_\text{y} &= \text{punt2}[1] - \text{punt2}[1] * t + \text{punt4}[1] * t \\
punt\_\text{inter} &= [a\_\text{int} \_\text{x}, a\_\text{int} \_\text{y}, 0] \\
punt\_\text{vertical} &= \text{punt\_inter} \\
\text{interseccio} &= 1
\end{align*}
\]

Finalment afegirem els vèrtexs 3 i 4:

\[
\text{Crear_Matriu_Boca(matriu nova, punt3[0], punt3[1], punt3[2])} \\
\text{Crear_Matriu_Boca(matriu nova, punt4[0], punt4[1], punt4[2])}
\]

El següent cas també conté dos vèrtexs que es troben a l’interior de la boca i dos vèrtexs que es troben al seu exterior.

\[
\begin{align*}
\text{if (area} &= [1, 1, 0, 0]): \\
\text{interseccio} &= 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{if ((Existeix_{\text{Vertex_Boca}}(matriu internal, punt1[0], punt1[1], punt1[2]) == "True")} \\
\text{and (Existeix_{\text{Vertex_Boca}} (matriu nova, punt1[0], punt1[1], punt1[2]) == "False")}): \\
\text{Crear_Matriu_Boca(matriu nova, punt1[0], punt1[1], punt1[2])}
\end{align*}
\]

\[
\begin{align*}
\text{if ((Existeix_{\text{Vertex_Boca}}(matriu internal, punt2[0], punt2[1], punt2[2]) == "True")} \\
\text{and (Existeix_{\text{Vertex_Boca}} (matriu nova, punt2[0], punt2[1], punt2[2]) == "False")}): \\
\text{Crear_Matriu_Boca_Tri(matriu nova, punt2[0], punt2[1], punt2[2])}
\end{align*}
\]

\[
\begin{align*}
\text{if (HiHa_{\text{InterVertical}}(\text{punt1, punt3, b1, b2}) <= "False")}: \\
t &= \text{HiHa_{\text{InterVertical}}(\text{punt1, punt3, b1, b2})} \\
a\_\text{int} \_\text{x} &= \text{punt1}[0] - \text{punt1}[0] * t + \text{punt3}[0] * t \\
a\_\text{int} \_\text{y} &= \text{punt1}[1] - \text{punt1}[1] * t + \text{punt3}[1] * t \\
punt\_\text{inter} &= [a\_\text{int} \_\text{x}, a\_\text{int} \_\text{y}, 0] \\
punt\_\text{vertical} &= \text{punt\_inter} \\
\text{interseccio} &= 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{if ((Existeix_{\text{Vertex_Boca}} (matriu nova, punt\_inter[0], punt\_inter[1], punt\_inter[2]) == "False")}): \\
\text{Crear_Matriu_Boca(matriu nova, punt\_inter[0], punt\_inter[1], punt\_inter[2])}
\end{align*}
\]
if (HiHa_InterVertical(punt2,punt4,b1,b2)<"False"):
    t=HiHa_InterVertical(punt2,punt4,b1,b2)
    a_int_x= punt2[0] - punt2[0]*t + punt4[0]*t
    a_int_y= punt2[1] - punt2[1]*t + punt4[1]*t
    punt_inter4=[a_int_x,a_int_y,0]
puntvertical2=punt_inter4
interseccio=1

if (Existeix_Vertex_Boca (matrionova, punt_inter4[0], punt_inter4[1], punt_inter4[2]) == "False"):
    Crear_Matriu_Boca(matrionova,punt_inter4[0],punt_inter4[1],punt_inter4[2])

Els dos primers vèrtexs 1 i 2 els inserim a la nostra matriu després de comprovar que no el repetim i que es troben a l’interior de la boca del reflector:

Crear_Matriu_Boca(matrionova,punt1[0],punt1[1],punt1[2])
Crear_Matriu_Boca(matrionova,punt2[0],punt2[1],punt2[2])

En canvi el vèrtex 3 es troba fora de la boca per tant haurem de comprovar si existeix intersecció vertical entre el vèrtex 1 i el vèrtex 3; una vegada trobat el punt on intersequen, l’entrarem a la nostra matriu, comprovant abans que no existeixi, així evitarem que els vèrtexs es repeteixin.

if (HiHa_InterVertical(punt1,punt3,b1,b2)<"False"):
    t=HiHa_InterVertical(punt1,punt3,b1,b2)
    a_int_x= punt1[0] - punt1[0]*t + punt3[0]*t
    a_int_y= punt1[1] - punt1[1]*t + punt3[1]*t
    punt_inter3=[a_int_x,a_int_y,0]
puntvertical1=punt_inter3
interseccio=1

El vèrtex 4 ens trobem en el mateix cas que en el vèrtex 3, per tant també haurem de buscar-ne la intersecció vertical però aquesta vegada amb els vèrtexs 2 i 4 i finalment inserirem el punt d’intersecció a la matriu mitjançant la funció Crear_Matriu_Boca(matrionova,punt_inter4[0],punt_inter4[1],punt_inter4[2]).

if (area==[0,1,0,1]):
    interseccio=0
if (HiHa_InterHoritzontal(punt1,punt2,b1,b2)<"False"):
    t=HiHa_InterHoritzontal(punt1,punt2,b1,b2)
    a_int_x= punt1[0] - punt1[0]*t + punt2[0]*t
    a_int_y= punt1[1] - punt1[1]*t + punt2[1]*t
    punt_inter1=[a_int_x,a_int_y,0]
punthoritzontal1=punt_inter1
interseccio=1
if (Existeix_Vertex_Boca (matriu_nova, punt_inter1[0], punt_inter1[1], punt_inter1[2]) == "False"):

    Crear_Matriu_Boca(matriu_nova, punt_inter1[0], punt_inter1[1], punt_inter1[2])

if ((Existeix_Vertex_Boca(matriu_intern, punt2[0], punt2[1], punt2[2]) == "True")
    and (Existeix_Vertex_Boca(matriu_nova, punt2[0], punt2[1], punt2[2]) == "False")
    and interseccio == 1):
    Crear_Matriu_Boca(matriu_nova, punt2[0], punt2[1], punt2[2])

if (HiHa_InterHoritzontal(punt3, punt4, b1, b2) <> "False"):
    
    t = HiHa_InterHoritzontal(punt3, punt4, b1, b2)
    a_int_x = punt3[0] - punt3[0] * t + punt4[0] * t
    punt_inter3 = [a_int_x, a_int_y, 0]
    punthoritzontal2 = punt_inter3
    interseccio = 1

if ((Existeix_Vertex_Boca (matriu_nova, punt_inter3[0], punt_inter3[1], punt_inter3[2]) == "False")):

    Crear_Matriu_Boca(matriu_nova, punt_inter3[0], punt_inter3[1], punt_inter3[2])

if ((Existeix_Vertex_Boca(matriu_intern, punt4[0], punt4[1], punt4[2]) == "True")
    and (Existeix_Vertex_Boca(matriu_nova, punt4[0], punt4[1], punt4[2]) == "False")
    and interseccio == 1):
    Crear_Matriu_Boca(matriu_nova, punt4[0], punt4[1], punt4[2])

Amb aquesta combinació de vèrtexs podem crear dos triangles, el primer triangle el creem mitjançant la intersecció horitzontal entre el vèrtex 1 i el vèrtex 2:

if (HiHa_InterHoritzontal(punt1, punt2, b1, b2) <> "False"):

    t = HiHa_InterHoritzontal(punt1, punt2, b1, b2)
    a_int_x = punt1[0] - punt1[0] * t + punt2[0] * t
    punt_inter1 = [a_int_x, a_int_y, 0]
    punthoritzontal1 = punt_inter1
    interseccio = 1

Amb la inserció del vèrtex 2 a la matriu Crear_Matriu_Boca (matriu_nova, punt2[0], punt2[1], punt2[2]) i amb la intersecció horitzontal del vèrtex 3 i el vèrtex 4:

if (HiHa_InterHoritzontal(punt3, punt4, b1, b2) <> "False"):

    t = HiHa_InterHoritzontal(punt3, punt4, b1, b2)
    a_int_x = punt3[0] - punt3[0] * t + punt4[0] * t
Capítol 4. Implementació

Per últim afegim el vèrtex 4 a la matriu. \textit{Crear_Matriu_Boca} (matriu nova, punt4[0], punt4[1], punt4[2]), comprovant, abans però, que no existeixi ja a la matriu.

\begin{verbatim}
if (area==[1,0,1,0]):
   interseccio=0

if ((Existeix_Vertex_Boca(matriu_intern,punt1[0],punt1[1],punt1[2])=="True")
   and (Existeix_Vertex_Boca (matriu nova, punt1[0], punt1[1], punt1[2]) ==
   "False")):
   Crear_Matriu_Boca(matriu nova, punt1[0], punt1[1], punt1[2])

if (HiHa_InterHoritzontal(punt1,punt2,b1,b2)<="False"):
   t = HiHa_InterHoritzontal(punt1,punt2,b1,b2)
   a_int_x = punt1[0] - punt1[0]*t + punt2[0]*t
   a_int_y = punt1[1] - punt1[1]*t + punt2[1]*t
   punt_inter2=[a_int_x,a_int_y,0]
   punthoritzontal1=punt_inter2
   interseccio=1

if (Existeix_Vertex_Boca (matriu nova, punt_inter2[0], punt_inter2[1],
   punt_inter2[2]) =="False"):
   Crear_Matriu_Boca(matriu nova, punt_inter2[0], punt_inter2[1],
   punt_inter2[2])

if ((Existeix_Vertex_Boca (matriu intern, punt3[0], punt3[1], punt3[2]) ==
"True") and (Existeix_Vertex_Boca (matriu nova, punt3[0], punt3[1], punt3[2])
== "False") and interseccio==1):
   Crear_Matriu_Boca (matriu nova, punt3[0], punt3[1], punt3[2])

if (HiHa_InterHoritzontal(punt3,punt4,b1,b2)<="False"):
   t = HiHa_InterHoritzontal(punt3,punt4,b1,b2)
   a_int_x = punt3[0] - punt3[0]*t + punt4[0]*t
   a_int_y = punt3[1] - punt3[1]*t + punt4[1]*t
   punt_inter4=[a_int_x,a_int_y,0]
   punthoritzontal2=punt_inter4
   interseccio=1

if (Existeix_Vertex_Boca (matriu nova, punt_inter4[0], punt_inter4[1],
   punt_inter4[2]) =="False" and interseccio==1):
   Crear_Matriu_Boca (matriu nova, punt_inter4[0], punt_inter4[1],
   punt_inter4[2])
\end{verbatim}
Arribem ja a l’últim cas on tenim dos vèrtexs interiors i dos vèrtexs exterior a la boca del reflector, concretament amb el vèrtex 1 i el vèrtex 3 a l’interior de la boca i el vèrtex 2 i el vèrtex 4 a fora del reflector.

Tal i com podem veure en el dibuix, es poden crear dos triangles, també. Primerament inserim el vèrtex 1 si aquest no es troba ja a la matriu **Crear_Matriu_Boca** (matriunova, punt1[0], punt1[1], punt1[2]) i a continuació busquem el punt d’intersecció horitzontal entre el vèrtex 1 i el vèrtex 2, si existeix; i en cas de què el punt d’intersecció trobat no existeixi a la matriu el crearem:

```
if (HiHa_InterHoritzontal(punt1,punt2,b1,b2)<"False"):  
t = HiHa_InterHoritzontal(punt1,punt2,b1,b2)  
a_int_x = punt1[0] - punt1[0]*t + punt2[0]*t  
a_int_y = punt1[1] - punt1[1]*t + punt2[1]*t  
punt_inter2=[a_int_x,a_int_y,0]  
punthoritzontal1=punt_inter2  
terseccio=1
```

El vèrtex 3 l’afegirem a la matriu mentre que en comptes d’afegir el vèrtex 4, com que aquest es troba fora del reflector buscarem un punt d’intersecció entre el vèrtex 3 i el vèrtex 4 i l’afegirem a la nostra matriu.

Seguidament mostrem els casos en què solament es troba un vèrtex a l’interior del reflector, els altres tres vèrtexs es trobaran a fora:

```
if (Hi_Ha_Uns_Vector(area)==1):  
if (area==[1,0,0,0]):  
terseccio=0

if ((Existeix_Vertex_Boca (matriu_intern, punt1[0], punt1[1], punt1[2]) == "True") and (Existeix_Vertex_Boca (matriunova, punt1[0], punt1[1], punt1[2]) == "False")):  
Crear_Matriu_Boca(matriunova,punt_inter2[0],punt_inter2[1],punt_inter2[2])
```

![Diagrama](image)
if (Existeix_Vertex_Boca (matriunova, punt_inter2[0], punt_inter2[1], punt_inter2[2]) == "False"):
    Crear_Matriu_Boca(matriunova, punt_inter2[0], punt_inter2[1], punt_inter2[2])

if (HiHa_InterVertical(punt1, punt3, b1, b2) != "False"):
    t = HiHa_InterVertical(punt1, punt3, b1, b2)
    a_int_x = punt1[0] - punt1[0]*t + punt3[0]*t
    a_int_y = punt1[1] - punt1[1]*t + punt3[1]*t
    punt_inter3 = [a_int_x, a_int_y, 0]
    puntvertical = punt_inter3
    interseccio = 1

    if (Existeix_Vertex_Boca (matriunova, punt_inter3[0], punt_inter3[1], punt_inter3[2]) == "False"):
        Crear_Matriu_Boca(matriunova, punt_inter3[0], punt_inter3[1], punt_inter3[2])

Com podem comprovar en el codi anterior mirem si els vèrtexs que estem apunt de tractar només n’hi ha un que es troba a l’interior del reflector: if (Hi_Ha_Uns_Vector(area) == 1). En cas afirmatiu passaríem a comprovar de quin cas es tracta. Concretament el nostra cas és 1000; això voldrà dir que el vèrtex 1 pertany a l’interior de la boca del reflector mentre que els altres tres es troben a fora.

Només hem d’afegir tres vèrtexs. El primer vèrtex que afegirem és el vèrtex 1 Crear_Matriu_Boca(matriunova, punt1[0], punt1[1], punt1[2]) mentre que els altres dos vèrtexs els haurem de calcular mitjançant el punt d’intersecció entre el vèrtex 1 i el vèrtex 2 els quals comprovarem que existeix una intersecció horitzontal HiHa_InterHoritzontal(punt1, punt2, b1, b2) i afegint aquest punt d’intersecció a la matriu Crear_Matriu_Boca(matriunova, punt_inter2[0], punt_inter2[1], punt_inter2[2]).

Seguidament comprovarem si existeix una intersecció vertical entre el vèrtex 1 i el vèrtex 3, buscarem el punt on intersequen i l’afegirem a la matriu, comprovant, abans, si aquest es troba ja a la matriu nova i si es tracta d’un punt de l’interior de la boca del reflector:

    if ((Existeix_Vertex_Boca (matriunova, punt_inter3[0], punt_inter3[1], punt_inter3[2]) == "False")):  
        Crear_Matriu_Boca(matriunova, punt_inter3[0], punt_inter3[1], punt_inter3[2])

Passem a mostrar el cas en què l’únic vèrtex que es troba a l’interior de la boca del reflector és el vèrtex 2:

    if (area == [0,1,0,0]):
        interseccio = 0
if (HiHa_InterHoritzontal(punt1, punt2, b1, b2) <> "False"):
    t = HiHa_InterHoritzontal(punt1, punt2, b1, b2)
    a_int_x = punt1[0] - punt1[0] * t + punt2[0] * t
    punt_inter1 = [a_int_x, a_int_y, 0]
    puntHoritzontal = punt_inter1
    interseccio = 1

    if (Existeix_Vertex_Boca (matriunova, punt_inter1[0], punt_inter1[1], punt_inter1[2]) == "False"):
        Crear_Matriu_Boca_Tri(matriunova, punt_inter1[0], punt_inter1[1], punt_inter1[2])

if ((Existeix_Vertex_Boca(matriu_intern, punt2[0], punt2[1], punt2[2]) == "True")
    and (Existeix_Vertex_Boca(matriunova, punt2[0], punt2[1], punt2[2]) == "False")
    and interseccio == 1):
    Crear_Matriu_Boca(matriunova, punt2[0], punt2[1], punt2[2])

if (HiHa_InterVertical(punt2, punt4, b1, b2) <> "False"):
    t = HiHa_InterVertical(punt2, punt4, b1, b2)
    a_int_x = punt2[0] - punt2[0] * t + punt4[0] * t
    punt_inter4 = [a_int_x, a_int_y, 0]
    puntvertical = punt_inter4
    interseccio = 1

    if ((Existeix_Vertex_Boca (matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2]) == "False") and (interseccio == 1)):
        Crear_Matriu_Boca(matriunova, punt_inter4[0], punt_inter4[1], punt_inter4[2])

Després de comprovar que es tracta del cas 0100 mirarem si existeix intersecció horitzontal entre el vèrtex 1 i el vèrtex 2. En cas afirmatiu buscarem el punt d’intersecció, tant la coordenada x com la seva coordenada y i afegirem el vèrtex a la matriu, sempre i quan aquest no es trobi a la matriu.

if (HiHa_InterHoritzontal(punt1, punt2, b1, b2) <> "False"):
    t = HiHa_InterHoritzontal(punt1, punt2, b1, b2)
    a_int_x = punt1[0] - punt1[0] * t + punt2[0] * t
    punt_inter1 = [a_int_x, a_int_y, 0]
    puntHoritzontal = punt_inter1
    interseccio = 1

    if (Existeix_Vertex_Boca (matriunova, punt_inter1[0], punt_inter1[1], punt_inter1[2]) == "False"):
El següent vèrtex que hem de tractar és el vèrtex 2. Com que aquest ja pertany a la boca del reflector simplement l’hem d’afegir a la matriu; abans comprovarem que aquest vèrtex no es trobi a dins.

\[
\text{Crear Matriu Boca (matriunova, punt2[0], punt2[1], punt2[2])}
\]

Ja només ens queda buscar el punt d’intersecció vertical entre el vèrtex 2 i el vèrtex 4, comprovant primer si existeix intersecció entre ells dos i seguidament comprovant que el punt resultant no es trobi dins la matriu a on el volem inserir.

\[
\begin{align*}
\text{if (area==[0,0,1,0]):} \\
\quad \text{interseccio}=0 \\
\text{if (HiHa InterVertical(punt1,punt3,b1,b2)<>"False"):} \\
\quad t=\text{HiHa InterVertical(punt1,punt3,b1,b2)} \\
\quad a\_int\_x=punt1[0] - punt1[0]*t + punt3[0]*t \\
\quad a\_int\_y=punt1[1] - punt1[1]*t + punt3[1]*t \\
\quad punt\_inter1=[a\_int\_x, a\_int\_y,0] \\
\quad puntvertical=punt\_inter1 \\
\quad interseccio=1 \\
\end{align*}
\]

\[
\begin{align*}
\text{if (Existeix Vertex Boca (matriunova, punt_inter1[0], punt_inter1[1], punt_inter1[2]) }==\text{"False"):} \\
\text{Crear Matriu Boca(matriunova,punt_inter1[0],punt_inter1[1],punt_inter1[2])} \\
\end{align*}
\]

\[
\begin{align*}
\text{if ((Existeix Vertex Boca(matriu_intern,punt3[0],punt3[1],punt3[2])=="True")} \\
\quad \text{and (Existeix Vertex Boca(matriunova,punt3[0],punt3[1],punt3[2])=="False")} \\
\quad \text{and (interseccio==1))}: \\
\quad \text{Crear Matriu Boca(matriunova,punt3[0],punt3[1],punt3[2])} \\
\end{align*}
\]

\[
\begin{align*}
\text{if (HiHa InterHoritzontal(punt3,punt4,b1,b2)<>"False"):} \\
\quad t=\text{HiHa InterHoritzontal(punt3,punt4,b1,b2)} \\
\quad a\_int\_x=punt3[0] - punt3[0]*t + punt4[0]*t \\
\quad a\_int\_y=punt3[1] - punt3[1]*t + punt4[1]*t \\
\quad punt\_inter4=[a\_int\_x, a\_int\_y,0] \\
\quad punthoritzontal=punt\_inter4 \\
\quad interseccio=1 \\
\end{align*}
\]

\[
\begin{align*}
\text{if ((Existeix Vertex Boca (matriunova,punt_inter4[0], punt_inter4[1], punt_inter4[2]) }== \text{"False") and (interseccio==1)):} \\
\text{Crear Matriu Boca(matriunova,punt_inter4[0],punt_inter4[1],punt_inter4[2])} \\
\end{align*}
\]
També ens trobem davant d’un cas on podem construir un únic triangle a partir de tres vèrtexs on el vèrtex 3 pertany a l’interior de la boca del reflector. Així doncs començarem per comprovar si existeix una intersecció vertical entre el vèrtex 1 i el vèrtex 3. En cas que existeixi buscarem el seu punt d’intersecció i afegirem a la matriu:

\[
\begin{align*}
  t &= \text{HiHa\_InterVertical}(\text{punt1, punt3, b1, b2}) \\
  a_{\text{int}}_x &= \text{punt1}[0] - \text{punt1}[0] * t + \text{punt3}[0] * t \\
  a_{\text{int}}_y &= \text{punt1}[1] - \text{punt1}[1] * t + \text{punt3}[1] * t \\
  \text{punt\_inter1} &= [a_{\text{int}}_x, a_{\text{int}}_y, 0] \\
  \text{punt\_vertical} &= \text{punt\_inter1} \\
  \text{interseccio} &= 1
\end{align*}
\]

Seguidament el vèrtex 3 afegirem a la matriu, comprovant abans que no es trobi ja a dins d’aquesta i finalment buscarem si existeix una intersecció horitzontal entre el vèrtex 3 i el vèrtex 4 i afegirem el punt d’intersecció a la nostra matriu:

\[
\begin{align*}
  \text{if} &\quad (\text{Existeix\_Vertex\_Boca} (\text{matriunova, punt\_inter4[0], punt\_inter4[1], punt\_inter4[2]})) == "\text{False}" \text{ and (interseccio==1))}:
  \text{Crear\_Matriu\_Boca(matriunova, punt\_inter4[0], punt\_inter4[1], punt\_inter4[2])}
\end{align*}
\]

Finalment arribem a l’últim cas de tots en què trobem el vèrtex 4 que pertany a l’interior de la boca del reflector i la resta de vèrtexs no. El seu codi és el següent:

\[
\begin{align*}
  \text{if} &\quad (\text{area} == [0,0,0,1]):
    \text{interseccio} = 0 \\
  \text{if} &\quad (\text{HiHa\_InterVertical(punt2, punt4, b1, b2)} <= "\text{False}"): \\
    t &= \text{HiHa\_InterVertical(punt2, punt4, b1, b2)} \\
    a_{\text{int}}_x &= \text{punt2}[0] - \text{punt2}[0] * t + \text{punt4}[0] * t \\
    a_{\text{int}}_y &= \text{punt2}[1] - \text{punt2}[1] * t + \text{punt4}[1] * t \\
    \text{punt\_inter2} &= [a_{\text{int}}_x, a_{\text{int}}_y, 0] \\
    \text{punt\_vertical} &= \text{punt\_inter2} \\
    \text{interseccio} &= 1 \\
  \text{if} &\quad (\text{Existeix\_Vertex\_Boca} (\text{matriunova, punt\_inter2[0], punt\_inter2[1], punt\_inter2[2]})) == "\text{False}"): \\
    \text{Crear\_Matriu\_Boca(matriunova, punt\_inter2[0], punt\_inter2[1], punt\_inter2[2])}
\end{align*}
\]

\[
\begin{align*}
  \text{if} &\quad (\text{HiHa\_InterHoritzontal(punt3, punt4, b1, b2)} <= "\text{False}"): \\
    t &= \text{HiHa\_InterHoritzontal(punt3, punt4, b1, b2)} \\
    a_{\text{int}}_x &= \text{punt3}[0] - \text{punt3}[0] * t + \text{punt4}[0] * t \\
    a_{\text{int}}_y &= \text{punt3}[1] - \text{punt3}[1] * t + \text{punt4}[1] * t \\
    \text{punt\_inter3} &= [a_{\text{int}}_x, a_{\text{int}}_y, 0] \\
    \text{punt\_horizontal} &= \text{punt\_inter3} \\
    \text{interseccio} &= 1
\end{align*}
\]
if ((Existeix_Vertex_Boca (matriunova, punt_inter3[0], punt_inter3[1], punt_inter3[2]) == "False")):

Crear_Matriu_Boca(matriunova,punt_inter3[0],punt_inter3[1],punt_inter3[2])

if ((Existeix_Vertex_Boca(matriu_intern,punt4[0],punt4[1],punt4[2])=="True")
and (Existeix_Vertex_Boca(matriunova,punt4[0],punt4[1],punt4[2])=="False")
and (interseccio==1)):

Crear_Matriu_Boca (matriunova, punt4[0], punt4[1], punt4[2])

Amb la funció HiHa_InterVertical(punt2,punt4,b1,b2) comprovem si existeix intersecció vertical entre el vèrtex 2 i el vèrtex 4. En cas d’haver-hi intersecció es buscarà el punt on hi ha intersecció i s’inserirà a la nostra matriu.

El següent punt d’intersecció es buscarà entre el vèrtex 3 i el vèrtex 4 mitjançant HiHa_InterHoritzontal(punt3,punt4,b1,b2) la qual ens dirà si existeix, o no, intersecció. A continuació, i en cas de què hi hagi intersecció, es buscarà el punt d’intersecció i s’afegirà a la matriu. Per últim s’inserirà el vèrtex 4 a la nostra matriu. Aquesta matriu on inserim els vèrtex serà la matriu resultant, la qual agafarem tots els seus vèrtexs que la formen i dibuixarem els vèrtexs a la interfície de Blender i en realitzarem la triangulació dels seus vèrtexs.

4.6. Creació del pla situat sota el reflector

Una vegada tenim el reflector construït cal simular el carrer on estarà situat el reflector mitjançant un pla. Aquest pla es trobarà per sota del nostre reflector i serà unes vàries vegades més gran que el reflector. Cal dir que el pla que creem serveix, simplement, perquè l’usuari es faci una idea de quina distància hi ha entre el reflector i la superfície, la qual pot simular un carrer o un terra pla. Nosaltres aprofitem la superfície per establir-hi la càmera i d’aquesta manera l’usuari no veu una càmera submergida a l’aire. La càmera és col·locada sobre aquesta superfície perquè ens interessa captar la intensitat de llum que es projecte sobre aquesta i és generada pel reflector. Aquesta càmera és utilitzada per obtenir les imatges de la renderització. Quantes més imatges obtinguem, més bona serà la nostra aproximació en la realitat, per això mourem la càmera en diferents posicions del pla. Per crear el pla executarem la funció CrearPla.

La funció CrearPla executa el següent codi:

def CrearPla(Plan,X,Y,Z):
    zeta=Z-distancia.val
    vertex = NMesh.Vert(X,Y,Z)
    Plan.verts.append(vertex)
    vertex= NMesh.Vert(maximaX, Y, Z)
    Plan.verts.append(vertex)
    vertex=NMesh.Vert(maximaX,maximaY,Z)
    Plan.verts.append(vertex)
    vertex=NMesh.Vert(X,maximaY,Z)
Eina pel desenvolupament de reflectors de conjunts òptics

Plan.verts.append(vertex)
cara=NMesh.Face()
cara.append(Plan.verts[0])
cara.append(Plan.verts[1])
cara.append(Plan.verts[2])
cara.append(Plan.verts[3])
Plan.faces.append(cara)
ObjectePla = Object.New('Mesh','ObjectePla')
ObjectePla.link(Plan)
ObjectePla.setLocation(0,0,0)
ObjectePla.setLocation(0,0,zeta)
escena = Scene.GetCurrent()
escena.link(ObjectePla)
Blender.Redraw()

Com hem dit anteriorment la distància entre el reflector i el pla l’especifica l’usuari i és guardada dins la variable \( zeta=Z\)-\( \text{distància}.\text{val} \) on Z és el valor que li passem per paràmetre i normalment és 0 per defecte. Així doncs, el primer vèrtex que creem les mateixes coordenades que l’objecte que enllaça amb el reflector però amb la component Z diferent \( vertex = NMesh.\text{Vert}(X,Y,Z) \). Un pla està compost per quatre vèrtex així que haurem de jugar amb els límits de l’àrea del reflector. Finalment, després de crear els quatre vèrtexs que formaran el pla i afegir-los a la llista de vèrtexs d’aquest objecte caldrà crear una sola cara formada per aquests vèrtexs amb la instrucció \( \text{cara.append(Plan.verts[0])} \)

Plan.verts.append(vertex)
vertex= NMesh.Vert(maximaX, Y, Z)
Plan.verts.append(vertex)
vertex=NMesh.Vert(maximaX,maximaY,Z)
Plan.verts.append(vertex)
vertex=NMesh.Vert(X,maximaY,Z)
Plan.verts.append(vertex)
cara=NMesh.Face()
cara.append(Plan.verts[0])
cara.append(Plan.verts[1])
cara.append(Plan.verts[2])
cara.append(Plan.verts[3])
Plan.faces.append(cara)

4.7. Creació de la càmera

La càmera la crearem damunt de la superfície creada anteriorment d’aquesta manera seguirem els límits del pla creat. Inicialment la càmera és inserida sobre el punt mig del pla i una vegada inicialitzem el render, aquesta es mourà equitativament en les diferents posicions que li pertoqui, ja que l’usuari haurà escollit anteriorment al render, el número d’imatges que vol capturar les renderitzacions. Quantes més imatges s’obtinguin més llarg serà el temps de càlcul.
def Crear_Camera()  
c = Camera.New('ortho')  
c.setMode("showLimits","showMist")  
c.scale = 500.0  
c.setDrawSize(500)  
cur = Scene.getCurrent()  
ob = Object.New('Camera')  
ob.link(c)  
ob.constraints.append(Constraint.Type.TRACKTO)  
const = ob.constraints  
c = const[0]  

c[Constraint.Settings.TARGET]=objecte2  
c[Constraint.Settings.TRACK]=Constraint.Settings.TRACKNEGZ  
c[Constraint.Settings.UP]=Constraint.Settings.UPY  
centre_X=round((minimaX+maximaX)/2,3)  
centre_Y=round((minimaY+maximaY)/2,3)  
Plan=Object.Get('ObjectePla')  
ZPlan=Plan.LocZ  

ob.setLocation(centre_X,centre_Y,ZPlan+2)  
cur.link(ob)  
cur.setCurrentCamera(ob)  
Blender.Redraw()  

Prenem l’objecte que identifica la boca del reflector i la seva localització sobre la nostre interfície:  

BocaReflector= Object.Get('ObjectePoligon2')  
BocaReflector.getLocation()  

Creem la càmera amb la instrucció \( c = \text{Camera.New('ortho')} \) i l’enllacem amb el seu corresponent objecte \( ob = \text{Object.New('Camera')}; \ ob\text{.link(c)} \). La càmera ha d’apuntar mentre fa les renderitzacions a la boca del reflector en tot moment, per això afegim restriccions a la càmera del tipus TRACKTO mitjançant la funció \( ob\text{.constraints.append (Constraint.Type.TRACKTO)} \) i seguidament l’indiquem l’objecte que ha d’apuntar, d’aquesta manera s’origina un vincle entre la boca del reflector i la càmera:  

\[
c[\text{Constraint.Settings.TARGET}]=\text{objecte2}  
c[\text{Constraint.Settings.TRACK}]=\text{Constraint.Settings.TRACKNEGZ}  
c[\text{Constraint.Settings.UP}]=\text{Constraint.Settings.UPY}  
\]

Finalment cal situar la càmera al centre del reflector abans d’inicialitzar el render; per fèr-ho utilitzarem els límits de l’àrea del pla i l’indicarem la posició de l’objecte càmera amb la funció \( \text{setLocation} \):  

\[
centre_X=\text{round}((\text{minimaX}+\text{maximaX})/2,3)  
centre_Y=\text{round}((\text{minimaY}+\text{maximaY})/2,3)  
Plan=\text{Object.Get('ObjectePla')}  
\]
4.8. Realització del fitxer de render

Una vegada tenim el nostre reflector creat, és interessant poder treure imatges de la renderització per poder mesurar la intensitat de llum que el reflector projecta sobre el terra o en una superfície del carrer. Per això cal construir una sèrie de fitxers que captin la intensitat de llum emesa pel reflector, en diferents posicions de la superfície, mitjançant una càmera. Parlarem doncs de la funció `FerRender()`.

def FerRender():
    global escena, renderitza
    escena = Scene.GetCurrent()
    obccamera = Object.Get('Camera')
    part = "001"
    x = minimaX
    parts = sqrt(int(round(int(NImatges.val)))))
    longitudx = maximaX-minimaX
    longitudy = maximaY-minimaY
    divisionsx = int(round(int(longitudx/parts)))
    divisionsy = int(round(int(longitudy/parts)))
    EliminarArxiu(NomArxiu.val+'.txt')
    EliminarArxiu(NomArxiu.val+'.txt.copia')
    for n in range (0, int(parts)):
        y = minimaY
        mitjax = (x+x+divisionsx)/2
        for m in range (0, int(parts)):
            mitjay = (y+y+divisionsy)/2
            obccamera.setLocation(mitjax, mitjay, obccamera.LocZ)
            context = escena.getRenderingContext()
            Render.EnableDispWin()
            context.extensions = True
            context.imageType = Render.JPEG
            context.renderPath = Directori.val + "/"
            filename = NomArxiu.val + part
            print filename
            context.render()
            context.saveRenderedImage(filename, 0)
            Render.CloseRenderWindow()
            CrearFitxerText(obccamera.LocX, obccamera.LocY, filename)
            numero = int(part)
            numero = numero + 1
            if (numero < 10):
                part = "0" + str(numero)
            elif (numero < 100) and (numero > 9):
Com podem comprovar en el codi, primerament movem la càmera a la punta del pla, concretament a la posició on x és mínima. Tenim el número d’imatges que volem aconseguir fent el render guardat a la variable `Nimatges.val`. Com que fem servir dos bucles `for-in` haurem de repartir aquestes imatges, per tant realitzem l’arrel quadrada `parts=sqrt(int(round(int(Nimatges.val))))` on ara `parts` contindrà la quantitat d’imatges que haurem de crear a cada volta del bucle `for-in`. Comprovem quina és la distància on haurem de posicionar la càmera sobre l’eix X i sobre l’eix Y:

```
longitudx=maximaX-minimaX
longitudy=maximaY-minimaY
divisionsx=int(round(int(longitudx/parts)))
divisionsy=int(round(int(longitudy/parts)))
```

El fitxer de sortida prendrà el nom que l’usuari li hagi donat i que es trobarà guardat dins la variable `nomarxiu.val` per tant el fitxer de render obtindrà aquest nom més la concatenació de tres caràcters els quals seguiran una successió de números des de ‘000’ fins a ‘999’ `filename=NomArxiu.val+part.`

La càmera l’anem posicionant amb la sentència `obcamera.setLocation (mitjax, mitjay, obcamera.LocZ)` on els valors que prenen `mitjax` i `mitjay` són:

```
mitjax=(x+x+divisionsx)/2
mitjay=(y+y+divisionsy)/2
```

Per obrir la pantalla de renderització de Blender fem servir la funció `Render.EnableDispWin( )`, indicant-li el tipus d’imatge que volem crear `context.imageType = Render.JPEG` i portant a terme el render utilitzant `context.render( )` i guardant la imatge a l’arxiu que nosaltres li hem indicat prèviament amb `context.saveRenderedImage(filename,0)`. Cal aclarir que el format JPG, en què guardem la imatge, té una pèrdua de qualitat despreciable. Finalment hem de tancar la finestra de renderització `Render.CloseRenderWindow()` i incrementar els tres darrers dígits de l’arxiu per poder crear la següent imatge que obtindrem el render:

```
numero=int(part)
nnumero=numero+1
if (numero<10):
  part="0"+str(numero)
elif (numero<100)and(numero>9):
  part="0"+str(numero)
e l s e :
  part=str(numero)
```
Si ens fixem hi ha una funció que no l’hem explicada, \texttt{CrearFitxerText(obcamera.LocX, obcamera.LocY,filename)}, la qual crea un fitxer de text on hi guardem la posició de la càmera (x, y) i el nom del fitxer que pren la imatge. Per exemple:

-264.921875 -200.60067749 Imatge001
-264.921875 -100.60067749 Imatge002
-264.921875 -0.600674510002 Imatge003
-264.921875 99.3993225098 Imatge004
-264.921875 199.39932251 Imatge005
-132.921890259 -200.60067749 Imatge006
(...)

\begin{Verbatim}
\textbf{def CrearFitxerText(posX,posY,nomfitxer):}

\texttt{path=Directori.val+"/"}
\texttt{nom_fitxer = NomArxiu.val+'.txt'}
\texttt{os.chdir(Directori.val)}

\texttt{if not exists(Directori.val):}
\texttt{os.mkdir(Directori.val)}
\texttt{os.chdir(Directori.val)}
\texttt{f1 = open(nom_fitxer,'w')}
\texttt{print "No existeix el directori"}

\texttt{else:}

\texttt{if not exists(nom_fitxer):}
\texttt{f1=open(nom_fitxer,'w')}\n\texttt{f1.close()}
\texttt{print "No existeix el fitxer"}

\texttt{path=Directori.val+"/"}
\texttt{nom_fitxer = NomArxiu.val+'.txt'}
\texttt{nova_linia = str(posX)+' '+str(posY)+' '+nomfitxer+' '}
\texttt{nom_copia = nom_fitxer + '.copia'}
\texttt{os.chdir(Directori.val)}
\texttt{f1 = open(nom_fitxer,'r')}
\texttt{f2 = open(nom_copia,'w')}
\texttt{for linia in f1:}
\texttt{    f2.write(linia)}

\texttt{f1 = open(nom_fitxer,'w')}
\texttt{f2.close()}
\texttt{f1.close()}

\texttt{f1 = open(nom_copia,'r')}
\texttt{f2 = open(nom_fitxer, 'w')}
\texttt{for linia in f1:}
\end{Verbatim}
A la funció li passem la posició x i la posició y de la càmera (PosX i PosY) i el nom que pren el fitxer de render.

Li indiquem el directori on volem que ens guardi el fitxer de text i el nom que tindrà aquest fitxer. Com que és molt provable que no estiguem situats al directori on hem de gravar l’arxiu utilitzarem la funció `os.chdir(Directori.val)` per a posicionar-nos-hi i un cop situats crearem el fitxer amb `f1 = open(nom_fitxer,'w')`. En cas que no existeixi el directori el crearem `os.mkdir(Directori.val)`. Les línies que volem que tingui aquest fitxer han de portar: Posició X, Posició Y, nom del fitxer. Per tant utilitzarem l’estructura `nova_linia = str(posX)+' '+str(posY)+' '+nomfitxer+' '` i l’escriurem en el fitxer mitjançant la funció `f2.write(nova_linia + '\n')`.

4.9. Obtenció de les corbes de nivell utilitzant el Matlab

El Matlab ens ofereix la possibilitat de crear gràfiques a partir de les dades que li entrem. Per tal d’adquirir aquestes dades, utilitzem la programació de Matlab per llegir les nostres imatges generades des de Blender amb la nostra aplicació, i les seves coordenades, que ens proporciona el fitxer de text que acompanya a aquestes imatges.

Començarem per mostrar el codi on indiquem el directori a on guardem les nostres imatges. La variable on guardem aquesta informació s’anomena `mydir`, la qual emmagatzema una cadena de caràcters.

```
mydir = 'C:\Render3\'
[A, count] = readImagesAndCoordinates(mydir);
plot2DValues(A)
```

La matriu `[A, count]` guarda tots els valors de les imatges i coordenades de la posició, que la funció `readImagesAndCoordinates` li passarà prèviament. `Count` és el nombre de dades llegides. La funció `plot2DValues` ens mostrarà la nostra gràfica a partir dels valors que hem obtingut a la matriu `A`.

Seguidament mostrem el codi de la funció `readImagesAndCoordinates`, on li passem per paràmetre la ruta del directori on es troba l’arxiu de text que guarda les coordenades:

```
function [A,count]=readImagesAndCoordinates(index)

indexFile = sprintf('%s\%s\', index, 'Imatge.txt');
 fid = fopen(indexFile, 'r');
 format = '%g %g %s\r';
 [names, count] = fscanf(fid,format);
 fclose(fid);
```
La funció `sprintf()` realitza una concatenació de dos caràcters. Concatenarem la ruta del directori on es troba l’arxiu de text amb el nom que pren l’arxiu de text. Obrim l’arxiu de text amb la funció `fopen()`, la qual ens retorna un identificador per a l’arxiu. Podrem llegir les línies que contenen aquest arxiu mitjançant el seu identificador.

`FScanf()` llegix línia a línia l’arxiu que hem obert anteriorment, seguint el format que li hem especificat on trobarà primer dos valors reals i per últim una cadena de caràcters. Tota aquesta informació l’emmagatzemem dins la matriu `[names, count]` i finalment tanquem el fitxer amb la funció `fclose()`.

```plaintext
realCount = count / 3;
rowSize = length(names)/realCount;
sizeName = length(names)/realCount - 2;
```

La variable `realCount` ens guarda el nombre d’imatges llegides. La divisió entre tres que realitzem, la fem perquè tenim tres columnes (x, y, nom de la imatge) i per tant a la variable `count` s’hi troba el nombre total d’elements llegits. Les variables `rowSize` i `sizeName` guarden la mida de la línia i la mida del nom de la imatge, respectivament.

```plaintext
for i=1:realCount,
    row = names((i-1)*rowSize+1:i*rowSize)';
    imgName = sprintf('%s%s', index, strcat(char(row(3:(2+sizeName)))));
    temp = rgb2gray(imread(imgName));
    A(i).imgVal = sum(temp(:)) / 255;
    A(i).coord = row(1:2);
end;
```

Arribem al bucle `FOR` de la funció, que és on realitzem les operacions més importants per a l’obtenció de les corbes de nivell. El bucle va des de la posició 1 fins al número total d’imatges que tenim. El què es tracta de fer, és llegir imatge per imatge. Per tant fem la concatenació de la ruta del directori on conté les imatges del reflector (la ruta ha de ser la mateixa que contenia l’arxiu de text) i d’aquesta manera carreguem la imatge amb la funció `imread(imgName)`. La funció `rgb2gray()` converteix la nostra imatge que conté tres canals RGB en una imatge de tons grisos. Aquesta imatge de tons grisos l’emmagatzemem en una variable anomenada `temp`. Finalment, de la imatge en tons grisos, n’extraïem el valor de la intensitat de la llum, fent la suma de tots els pixels de la imatge i fent la divisió entre 255 colors que conté, normalització. Aquest valor el guardarem dins la matriu `A(i).imgVal`, juntament amb les seves coordenades `A(i).coord`.

Ja només ens cal mostrar el codi que utilitzem per cridar la gràfica que volem estudiar. Abans, però, hem de generar X, Y, Z.

```plaintext
function plot2DValues(image)
    count=length(image);
    x=zeros(1,count);
    y=zeros(1,count);
    z=zeros(1,count);
```
El paràmetre que li passem a la funció `plot2Dvalues()` és la matriu $A$, creada a la funció `readImagesAndCoordinates`. Per tant guardem el valor de la longitud de $A$, és a dir, el nombre total de imatges llegides, a la variable `count`. Creem tres matrius que puguin contenir tants elements com imatges haguem llegit, amb la funció `zeros()`.

```matlab
for i=1:count,
  x(1,i)=image(i).coord(1);
  y(1,i)=image(i).coord(2);
  z(1,i)=image(i).imgVal;
end;
```

El bucle `FOR` va des de 1 fins al nombre que guarda la variable `count` (nombre de imatges llegides). Dins del bucle omplim les matrius $x$, $y$, $z$ amb les coordenades $x$ de cada imatge, les coordenades $y$, i el valor de la intensitat, respectivament.

```matlab
minx = min(x); maxx=max(x);
miny = min(y); maxy=max(y);
intx = minx:(maxx-minx)/100:maxx;
inty = miny:(maxy-miny)/100:maxy;
[XI,YI] = meshgrid(intx, inty);
ZI=griddata(x,y,z,XI,YI);
meshc(XI,YI,ZI), hold
contour(x, y, z);
```

Finalment agafem els valors mínims i màxims, tant de $x$ com de $y$ per crear el producte cartesià a partir de dos vectors ($\text{intx}$, $\text{inty}$) amb la funció `meshgrid()`. La funció `griddata()` fa una interpolació de dues dimensions, mentre que la funció `meshc()` crea una superfície de malla tal i com podem veure a la figura 4.7. Amb la última funció que trobem escrita en el codi, fa la crida de la gràfica final i poder-la visualitzar per pantalla.

![Figura 4.7](image)

**Figura 4.7.** Gràfica representativa de la intensitat de llum obtinguda a partir d’un reflector.
4.10. Creació de la interfície

La interfície (veure figura 4.8) està composta per una sèrie d'objectes procedents del mòdul de Blender Draw on inclou els objectes tipus Button, Toggle, Slide, etc.

La interfície (figura 4.8) està composta per una sèrie d'objectes procedents del mòdul de Blender Draw on inclou els objectes tipus *Button*, *Toggle*, *Slide*, etc.

**Figura 4.8.** Interfície d’usuari.

El codi de la interfície d’usuari el mostrem a continuació, amb ell hem arribat a crear tots els controls.

```python
def gui():
    global gzipmode
    global outdir, toggle1, toggle2, togglex, toggley, togglez
    global distancia, mymsg, distanciaevt
    global Llumpevt, buto1premut, Cameraevt, Renderevt
    global sortirevt, plaevt, Llumpspotevt, Llumplampevt, Eix_xevt, Eix_yevt, Eixzevt
    global Caixa_Aevt, NomArxiu, Director i, Caixa_Bevt, NImatges, Caixa_Cevt
    buto1premut="False"
    glRasterPos2i(20, 120)
    Text("Importació de la Boca del Reflector en format 'dxf' (ESC per sortir): ")
    Button("Importar Arxiu", exportevt, 60, 40, 110, 19)
    Button("Sortir", sortirevt, 20, 10, 80, 18)
    distancia = Slider("Distància: ", distanciaevt, 200,68,210,18,distancia.val 2,400.0, 2)
    Button("Crear Plà", Plaevt, 200, 40, 80, 18)
    Button("Inserir Llump", Llumpevt, 295, 40, 90, 18)
    Toggle("Lamp", Llumplampevt, 390, 40, 50, 18, toggle1)
    Toggle("Spot", Llumpspotevt, 390, 20, 50, 18, toggle2)
    Button("Inserir càmera", Cameraevt, 455, 40, 90, 18)
    NomArxiu = String("Nom Arxiu: ", Caixa_Aevt,560, 60,200,18, "Imatge",20, "Introdueix el nom de l’arxiu a renderitzar")
    Director i = String("Nom Directori: ",Caixa_Bevt,560,80,400,18,"c:/Render",80,"Introdueix directori 'c:/nomdirectori'")
    NImatges = String("NºImatges: ",Caixa_Cevt,660,40,100,18,"50",6,"Número de Renders")
    Button ("Renderització", Renderevt,560,40,90,18)
```

UdG  123
La funció `gui()` ens crea tot el què es pot apreciar a la nostra interfície, per exemple el botó d’importació d’arxius DXF `Button("Importar Arxiu", exportevt, 60, 40, 110, 19)` on `Importar Arxiu` (veure figura 4.9) és el text que podem visualitzar damunt del botó, `exportevt` és l’event a què fa referència aquest botó i la resta de paràmetres són les coordenades que agafa aquest botó. En l’apartat 4.10.1, *Utilització dels events*, explicarem com funcionen els events.

![Figura 4.9. Objecte Button.](image)

Un altre objecte interessant de la interfície és `distancia = Slider("Distància: ", distanciaevt, 200,68,210,18,distancia.val 2,400.0, 2)` . Aquest tipus d’objecte és la barra de desplaçament que ens permet triar la distància entre el reflector i el pla (veure figura 4.10).

![Figura 4.10. Objecte Slider.](image)

La funció `Toggle("Lamp", Llumplampevt, 390, 40, 50, 18, toggle1)` crea un botó similar al que mostrem a la figura 4.11. Permet mantenir el botó premut fins que no s’ha premut el botó que es veu sota seu.

![Figura 4.11. Objecte Toggle.](image)

I finalment, un altre tipus d’objecte que podem apreciar a la nostra interfície, és el camp on podem afegir-hi text, perquè l’usuari pugui entrar el nom del fitxer o el nom del directori on es volen guardar les imatges de renderització.

\[
\text{Director} = \text{String("Nom Directori: ",Caixa_Bevt, 560,80,400,18, "c:/Render", 80, "Introdueix directori ‘c:/nomdirectori’")}
\]

\[
\text{NImatges} = \text{String("NºImatges: ",Caixa_Cevt,660,40,100,18,"50",6,"Número de Renders")}
\]

![Figura 4.12. Objecte String.](image)

Podem observar que la funció `String()` retorna un valor que guardem en una variable. El valor que retorna són els caràcters que l’usuari entra dins l’objecte `String` (veure figura 4.12).
4.10.1. Utilització dels events

Quan l’usuari prem un botó de la nostra interfície o desplaça la barra de distància s’està originant uns determinats events, els quals desencadenaran un seguit d’accions diferents per a cada event. Seguidament mostrem els events que hem implementat; cadascun d’ells està explicat al final del codi.

```python
def bevent(evt):
    global buto1premut
    global Plan, objecte, obj, Llumpspotevt, Llumplampevt, Eix_xevt, Eix_yevt, Eix_zevt
global toggle1, toggle2, togglex, toggley, togglez, Renderevt, objecte2
global poligon, poligon2, distancia, Fitxerevt

    Aquest primer event fa referència al botó importar Arxiu:

    if (evt == exportevt):
        Window.FileSelector (my_callback, "Escull un fitxer")
        buto1premut="true"
        Blender.Redraw()

        Per referenciar-nos al botó de sortir utilitzem aquest event:

    if (evt == sortirevt): Exit()

    L’event Plaevt ens indica que s’ha premut el botó Crear Pla:

    if (evt == Plaevt):
        scene = Scene.getCurrent()
        Plan=NMesh.New('Superficia')
        CrearPla(Plan,round(minimaX,3),round(minimaY,3),0)
        Redraw()

        Per cridar l’event LlumpEvt hem de prémer el botó Crear Llum:

    if (evt == Llumpevt):

        if (toggle2==1):
            scene=Scene.GetCurrent()
            l = Lamp.New('Spot')
            l.setMode('Shadows','Square')
            ob = Object.New('Lamp')
            ob.link(l)
            scene.link(ob)
            scene.makeCurrent()
        elif (toggle1==1):
            scene=Scene.GetCurrent()
            l = Lamp.New('Lamp')
            l.setMode('Shadows','Square')
```

UdG  125
ob = Object.New('Lamp')
ob.link(l)

scene.link(ob)
scene.makeCurrent()
Blender.Redraw()

if (evt == Llumpspotevt):
toggle2 = 1 - toggle2
toggle1 = 0
Blender.Redraw()

if (evt == Llumplampevt):
toggle1 = 1 - toggle1
toggle2 = 0
Blender.Redraw()

Amb l’event Cameraevt carreguem una càmera a la nostra escena:

if (evt == Cameraevt):
    BocaReflector = Object.Get('ObjectePoligon2')
    print BocaReflector
    BocaReflector.getLocation()
    Crear_Camera(BocaReflector)
    Blender.Redraw()

L’últim event crida a la funció FerRender(), és l’event Renderevt:

if (evt == Renderevt):
    FerRender()

Seguidament explicarem cadascú dels events: quines funcions executen i quan s’activa l’event.

El primer event s’activa en prèmer el botó Importar Arxiu:

```
If (evt == exportevt):
    Window.fileselector(my_callback, "Escull un fitxer")
    Boto1premut="true"
    Blender.redraw()
```

La funció my_callback() s’executa una vegada hem seleccionat el fitxer DXF i crida a tot el codi que hem explicat anteriorment per carregar un arxiu DXF. La funció fileselector() fa aparèixer la següent finestra de selecció de fitxers:
Eina pel desenvolupament de reflectors de conjunts òptics

Cadascun d’aquests events li correspon un número i quan es produeix un event fem una comparació entre l’event produït i el numero que li correspon l’event a cada botó i si són iguals s’executa l’acció que el precedeix.

Si s’ha premut el botó de creació del pla s’executarà el codi que precedeix a l’event enllaçat amb aquest botó.

```python
if (evt == Plaevt):
    scene = Scene.getCurrent()
    Plan = NMesh.New('Superficia')
    CrearPla(Plan,round(minimaX,3),round(minimaY,3),0)
    Redraw()
```

La variable `scene` guardará l’escena que en aquests moments es trobi activada. Creem un nou pla amb `Plan=NMesh.New('Superficia')` on li assignarem les seves coordenades amb la funció `CrearPla(Plan,round(minimaX,3),round(minimaY,3),0)`. Que ja hem explicat a l’apartat 4.6, Creació del pla situat sota el reflector. La funció `Redraw()` modifica els canvis que hi han hagut a la pantalla.

Seguidament mostrem què passa quan seleccionem els tipus de llum que volem que s’insereixi sota el nostre reflector:

```python
if (evt == Llumpevt):
    if (toggle2==1):
        scene=Scene.GetCurrent()
        l = Lamp.New('Spot')
        l.setMode('Shadows','Square')
        ob = Object.New('Lamp')
        ob.link(l)
        scene.link(ob)
        scene.makeCurrent()
    elif (toggle1==1):
        scene=Scene.GetCurrent()
        l = Lamp.New('Lamp')
        l.setMode('Shadows','Square')
```

**Figura 4.13.** Quadre de selecció d’arxius
**Capítol 4. Implementació**

```python
ob = Object.New('Lamp')
ob.link(l)

scene.link(ob)
scene.makeCurrent()
Blender.Redraw()
```

Com que tenim dos botons tipus *Toggle*, quan no està premut un, està premut l’altre, tenim dos condicionals `if` que fa que s’afegeixi un tipus de llum `l = Lamp.New('Spot')` o un altre `l = Lamp.New('Lamp')` i al final sempre enllaçant el llum amb un objecte tipus llum, també,

```python
ob = Object.New('Lamp')
ob.link(l)
```

Pel què fa al botó que afgeix la càmera el seu event executa la crida de la funció `Crear_Camera` en la qual si passem l’objecte que conté la boca del reflector. El codi que conté la funció `Crear_Camera()` està explicat a l’apartat 4.7, *Creació de la càmera*.

```python
if (evt == Cameraevt):
    BocaReflector = Object.Get('ObjectePoligon2')
    BocaReflector.getLocation()
    Crear_Camera(BocaReflector)
```

L’últim event que ens queda per veure és l’event que enllaçar amb el botó de renderització, que crida la funció `FerRender()` i ja s’ha comentat la seva implementació en l’apartat 4.8, *Realització del Fíxer de Render*.

```python
if (evt == Renderevt):
    FerRender()
```
5 RESULTATS I
CONCLUSIONS
5. Resultats i conclusions

5.1. Resultats obtinguts

Des de l’inici del projecte existia l’objectiu primordial de poder realitzar l’estudi de la intensitat de llum que oferia un reflector determinat. D’aquesta manera els enginyers en luminotècnia aconsegueixen un millor treball amb una considerable disminució del temps utilitzat. Així doncs, el nostre procés era clar: intentar implementar una eina per tal de poder desenvolupar reflectors i, posteriorment, realitzar l’estudi mitjançant gràfiques de la seva intensitat de llum que genera. Per aconseguir-ho hem aprofitat l’entorn gràfic que ens ofereix Blender.

Partint de la boca del reflector generada amb qualsevol eina de CAD, en format DXF, la nostra aplicació permet carregar aquest arxiu i generar el volum que volem que tingui el reflector. Una vegada hem obtingut el reflector, podem introduir una làmpada per tal de poder veure el seu efecte en combinació amb el reflector, d’una manera senzilla i concisa, a través d’un botó de control.

Com que es tracten de reflectors encarregats d’il·luminar les zones vials, hem simulat una superfície plana on l’usuari pot escollir la distància entre el reflector i aquest pla. Per representar un carrer o carretera d’aquesta manera les imatges s’extreuen a partir d’una càmera situada damunt la superfície una vegada l’usuari ha premut el control de renderització. L’estudi d’aquestes imatges es realitza posteriorment, amb qualsevol eina capaç de processar-les i així poder calcular la intensitat de llum que es projecte sobre la superfície plana i que han captat les imatges. Una bona opció és el Matlab, i per això hem aprofitat el seu ús.

Una vegada amb el projecte finalitzat podem demostrar diferents resultats obtinguts, com per exemple:

**Resultat 1:**

- Mitjançant la nostra aplicació carreguem la boca de la figura 5.1 des de l’arxiu DXF.

**Figura 5.1.** Visualització de la boca del reflector mitjançant una eina CAD.
• Visualitzem la boca del reflector carregada des de l’arxiu DXF i obtenim la \textbf{figura 5.2}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figura52.png}
\caption{Visualització de la boca del reflector en la nostra aplicació, després de carregar l’arxiu DXF.}
\end{figure}

• Donem volum al reflector, aixecant els punts en sentit a l’eix z i obtenim la \textbf{figura 5.3}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figura53.png}
\caption{La boca del reflector després de crear el seu volum.}
\end{figure}
Renderitzem el reflector per obtenir-ne les 400 imatges que després processarem. Algunes de les imatges que hem obtingut del render són les \textit{figures 5.4 i 5.5}.

\textbf{Figura 5.4.} Imatge obtinguda del reflector després de la renderització. \textbf{Figura 5.5.} Imatge obtinguda del reflector després de la renderització.

\textbf{Figura 5.6.} Corbes de nivell que representen la intensitat de llum de les imatges obtingudes anteriorment. La distància entre el pla i el reflector és de 100 unitats.
Finalment, utilitzant el Matlab, obtenim la gràfica de la figura 5.6 que ens mostra la intensitat de llum que dibuixa el nostre reflector sobre la superfície pla o el terra. La figura 5.7 ens mostra les corbes de nivell en 2D.

**Resultado 2:**

- Mitjançant la nostra aplicació carreguem la boca de la figura 5.8 des de l’arxiu DXF.

**Figura 5.7.** Corbes de nivell que representen la intensitat de llum de les imatges obtingudes anteriorment. És el mateix resultat anterior però en 2D.

**Figura 5.8.** Visualització de la boca del reflector mitjançant una eina CAD.
• Visualitzem la boca del reflector carregada des de l’arxiu DXF obtenint la \textit{figura 5.9}.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{figura59}
\caption{Visualització de la boca del reflector en la nostra aplicació, després de carregar l’arxiu DXF.}
\end{figure}

• Donem volum al reflector, aixecant els punts en sentit a l’eix z i obtenim la \textit{figura 5.10}.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{figura510}
\caption{La boca del reflector després de crear el seu volum.}
\end{figure}
• Renderitzem el reflector per obtenir-ne les 400 imatges que després processarem. Algunes de les imatges que hem obtingut del render són les figures 5.11 i 5.12.

Figura 5.11. Imatge obtinguda del reflector després de la renderització.

Figura 5.12. Imatge obtinguda del reflector després de la renderització.

Figura 5.13. Corbes de nivell que representen la intensitat de llum de les imatges obtingudes anteriorment. La distància entre el pla i el reflector és de 100 unitats.

• Finalment, utilitzant el Matlab, obtenim la gràfica de la figura 5.13 que ens mostra la intensitat de llum que dibuixa el nostre reflector sobre la superfície plana o el terra.
Resultat 3:

- Mitjançant la nostra aplicació carreguem la boca de la figura 5.14 des de l’arxiu DXF.

![Figura 5.14. Visualització de la boca del reflector mitjançant una eina CAD.](image1)

- Visualitzem la boca del reflector carregada des de l’arxiu DXF obtenint la figura 5.15.

![Figura 5.15. Visualització de la boca del reflector en la nostra aplicació, després de carregar l’arxiu DXF.](image2)
• Donem volum al reflector, aixecant els punts en sentit a l’eix z i obtenim la figura 5.16.

Figura 5.16. La boca del reflector després de crear el seu volum.

• Renderitzem el reflector per obtenir-ne les 400 imatges que després processarem. Algunes de les imatges que hem obtingut del render són les figures 5.17 i 5.18.

Figura 5.17. Imatge obtinguda del reflector després de la renderització.

Figura 5.18. Imatge obtinguda del reflector després de la renderització.
Eina pel desenvolupament de reflectors de conjunts òptics

• Finalment, utilitzant el Matlab, obtenim la gràfica de la figura 5.19 que ens mostra la intensitat de llum que dibuixa el nostre reflector sobre la superfície plana o el terra.

5.2. Conclusions del projecte

Amb la implementació de la nostra aplicació s’ha aconseguit:

1) Obtenir l’estructura d’un reflector mitjançant el contorn de la boca d’una manera ràpida. Cosa que suposaria gran quantitat de temps per a obtenir els reflectors corresponents a les boques de la figura 5.20.

![Figura 5.20. Conjunt de boques de reflectors.](image)

2) Donar fluïdesa a l’hora de crear i donar volum dels reflectors com els de la figura 5.21.

![Figura 5.21. Conjunt de reflectors amb volum.](image)
3) Simular la intensitat de llum que el reflector projecte sobre al terra o en una superfície del carrer, aplicant un llum al seu centre.

4) Aconseguir tantes mostres com sigui possible de la intensitat de llum que produeix el reflector. És important saber que quantes més mostres capturem millor qualitat obtindrem en el resultat però més temps de càlcul necessitarem.

5) Facilitar l’estudi de la intensitat de llum que el reflector ofereix, a través de les gràfiques realitzades.

![Figura 5.22. Conjunt de gràfiques obtingudes a partir de les mostres de llum de la nostra aplicació.](image)

6) Agilitzar el procés de la construcció de reflectors mitjançant eines informàtiques, juntament amb un estalvi de pressupost.
6  TREBALLS FUTURS I AGRAÏMENTS
6. Treballs futurs i agraïments

6.1. Treballs futurs

Partint de la base realitzada amb l’eina pel desenvolupament de reflectors de conjunts òptics podriem destacar quatre grans projectes que es podrien portar a terme, algun d’ells complementaria aquest projecte mentre que algun altre simplement s’aprofitaria dels avantatges d’aquest.

**Integració del càlcul d’intensitats en Blender**

Aquesta millora complementaria el projecte aportant més funcionalitat a la interfície de Blender. Per fer-ho seria necessari estudiar la manera de com Blender interpreta la intensitat de llum en les imatges. Seria necessari poder llegir cada píxell de la imatge i així poder-la sumar en el càlcul d’intensitats.

**Obtenir el reflector a partir de la llum que es dibuixa sobre la superfície**

Aquest projecte seria realitzar el disseny invers al que hem realitzat nosaltres. Hauríem de començar llegint el càlcul d’intensitats i acabant obtenint un reflector amb volum, que tingués la capacitat de generar la mateixa intensitat de llum que la que partíem inicialment.

**Projecció de la llum en superfícies més complexes**

Aquesta millora complementaria el nostre projecte de manera que tingués unes aplicacions més concretes, ja que podríem crear superfícies les quals no fossin planes. Podríem representar reflectors aplicats en llocs de treball com oficines, on la zona de il·luminació es concentra sobre la taula, etc. Aquest projecte tindria per objectiu la il·luminació de punts més específics.

6.2. Agraïments

Per donar finalitzat aquest projecte voldria dedicar unes línies a totes aquelles persones que m’han ajudat i fet costat, d’alguna manera o altra, abans i durant la realització d’aquest:

- Especialment a la meva família, qui ha estat pacient, i qui ha suportat els meus canvis d’humor.

- Al meu tutor, *Gustavo Patow*, per fer que cada vegada que jo entrava desanimat pel seu despatx, m’aconseguia motivar per continuar el projecte endavant.
I a tots aquells que es mereixen està presents en alguna d’aquestes línies us dic:

La vida és com un estel, la veus néixer i amb un
obrir i tancar d’ulls acaba per desaparèixer.

Viu la vida amb la mateixa il·lusió que observes aquest estel.
QUADRE DE FIGURES I
TAULES
Quadre de figures i taules

<table>
<thead>
<tr>
<th>Figura</th>
<th>Pàgina</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>22</td>
</tr>
<tr>
<td>2.6</td>
<td>23</td>
</tr>
<tr>
<td>2.7</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>25</td>
</tr>
<tr>
<td>2.10</td>
<td>25</td>
</tr>
<tr>
<td>2.11</td>
<td>26</td>
</tr>
<tr>
<td>2.12</td>
<td>26</td>
</tr>
<tr>
<td>2.13</td>
<td>27</td>
</tr>
<tr>
<td>2.14</td>
<td>28</td>
</tr>
<tr>
<td>2.15</td>
<td>28</td>
</tr>
<tr>
<td>2.16</td>
<td>29</td>
</tr>
<tr>
<td>2.17</td>
<td>29</td>
</tr>
<tr>
<td>2.18</td>
<td>30</td>
</tr>
<tr>
<td>2.19</td>
<td>30</td>
</tr>
<tr>
<td>2.20</td>
<td>30</td>
</tr>
<tr>
<td>2.21</td>
<td>31</td>
</tr>
<tr>
<td>2.22</td>
<td>31</td>
</tr>
<tr>
<td>2.23</td>
<td>31</td>
</tr>
<tr>
<td>2.24</td>
<td>32</td>
</tr>
<tr>
<td>2.25</td>
<td>32</td>
</tr>
<tr>
<td>2.26</td>
<td>33</td>
</tr>
<tr>
<td>2.27</td>
<td>33</td>
</tr>
<tr>
<td>2.28</td>
<td>33</td>
</tr>
<tr>
<td>2.29</td>
<td>34</td>
</tr>
<tr>
<td>2.30</td>
<td>35</td>
</tr>
<tr>
<td>2.31</td>
<td>35</td>
</tr>
<tr>
<td>2.32</td>
<td>35</td>
</tr>
<tr>
<td>2.33</td>
<td>36</td>
</tr>
<tr>
<td>2.34</td>
<td>37</td>
</tr>
<tr>
<td>2.35</td>
<td>38</td>
</tr>
<tr>
<td>2.36</td>
<td>38</td>
</tr>
<tr>
<td>2.37</td>
<td>39</td>
</tr>
<tr>
<td>2.38</td>
<td>39</td>
</tr>
<tr>
<td>2.39</td>
<td>40</td>
</tr>
<tr>
<td>2.40</td>
<td>40</td>
</tr>
<tr>
<td>2.41</td>
<td>41</td>
</tr>
<tr>
<td>2.42</td>
<td>41</td>
</tr>
<tr>
<td>2.43</td>
<td>41</td>
</tr>
<tr>
<td>2.44</td>
<td>42</td>
</tr>
<tr>
<td>2.45</td>
<td>42</td>
</tr>
<tr>
<td>Figura</td>
<td>Página</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>2.46</td>
<td>42</td>
</tr>
<tr>
<td>2.47</td>
<td>42</td>
</tr>
<tr>
<td>3.1</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>58</td>
</tr>
<tr>
<td>3.3</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>64</td>
</tr>
<tr>
<td>3.5</td>
<td>66</td>
</tr>
<tr>
<td>3.6</td>
<td>67</td>
</tr>
<tr>
<td>3.7</td>
<td>67</td>
</tr>
<tr>
<td>3.8</td>
<td>68</td>
</tr>
<tr>
<td>3.9</td>
<td>68</td>
</tr>
<tr>
<td>3.10</td>
<td>69</td>
</tr>
<tr>
<td>3.11</td>
<td>69</td>
</tr>
<tr>
<td>3.12</td>
<td>74</td>
</tr>
<tr>
<td>3.13</td>
<td>74</td>
</tr>
<tr>
<td>3.14</td>
<td>74</td>
</tr>
<tr>
<td>3.15</td>
<td>76</td>
</tr>
<tr>
<td>3.16</td>
<td>78</td>
</tr>
<tr>
<td>3.17</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>88</td>
</tr>
<tr>
<td>4.2</td>
<td>89</td>
</tr>
<tr>
<td>4.3</td>
<td>90</td>
</tr>
<tr>
<td>4.4</td>
<td>90</td>
</tr>
<tr>
<td>4.5</td>
<td>92</td>
</tr>
<tr>
<td>4.6</td>
<td>94</td>
</tr>
<tr>
<td>4.7</td>
<td>122</td>
</tr>
<tr>
<td>4.8</td>
<td>123</td>
</tr>
<tr>
<td>4.9</td>
<td>124</td>
</tr>
<tr>
<td>4.10</td>
<td>124</td>
</tr>
<tr>
<td>4.11</td>
<td>124</td>
</tr>
<tr>
<td>4.12</td>
<td>124</td>
</tr>
<tr>
<td>4.13</td>
<td>127</td>
</tr>
<tr>
<td>5.1</td>
<td>131</td>
</tr>
<tr>
<td>5.2</td>
<td>132</td>
</tr>
<tr>
<td>5.3</td>
<td>132</td>
</tr>
<tr>
<td>5.4</td>
<td>133</td>
</tr>
<tr>
<td>5.5</td>
<td>133</td>
</tr>
<tr>
<td>5.6</td>
<td>133</td>
</tr>
<tr>
<td>5.7</td>
<td>134</td>
</tr>
<tr>
<td>5.8</td>
<td>134</td>
</tr>
<tr>
<td>5.9</td>
<td>135</td>
</tr>
<tr>
<td>5.10</td>
<td>135</td>
</tr>
<tr>
<td>5.11</td>
<td>136</td>
</tr>
<tr>
<td>5.12</td>
<td>136</td>
</tr>
<tr>
<td>5.13</td>
<td>136</td>
</tr>
<tr>
<td>5.14</td>
<td>137</td>
</tr>
<tr>
<td>5.15</td>
<td>137</td>
</tr>
<tr>
<td>5.16</td>
<td>138</td>
</tr>
<tr>
<td>5.17</td>
<td>138</td>
</tr>
<tr>
<td>5.18</td>
<td>138</td>
</tr>
</tbody>
</table>
Eina pel desenvolupament de reflectors de conjunts òptics

Figura 5.19 ..................................................................................................... 139
Figura 5.20 ..................................................................................................... 139
Figura 5.21 ..................................................................................................... 139
Figura 5.22 ..................................................................................................... 140
Figura A2.1 .................................................................................................... 170
Figura A2.2 .................................................................................................... 180
Figura A2.3 .................................................................................................... 180

Taula A2.1 ...................................................................................................... 163
Taula A2.2 ...................................................................................................... 163
Taula A2.3 ...................................................................................................... 163
Taula A2.4 ...................................................................................................... 164
Eina pel desenvolupament de reflectors de conjunts òptics
BIBLIOGRAFÍA
Bibliografia

1. Marzal, Andrés y Graci, Isabel. *Introducción a la programación con Python*. Departamento de Lenguajes y Sistemas Informáticos. Universitat Jaume I.


5. Sunday, Dan. *The Convex Hull of a 2D Point Set or Polygon*. En línia. Internet. Accessible a [http://softsurfer.com/Archive/algorithm_0109/algorithm_0109.htm#isLeft()](http://softsurfer.com/Archive/algorithm_0109/algorithm_0109.htm#isLeft())


APÈNDIXS
Apèndix 1. El programari lliure

1. Història del programari lliure

Entre els anys 60 i 70 del Segle XX, el programari no era considerat un producte sinó un afegit que els venedors dels grans computadors de l'època (els mainframes) aportaven als seus clients perquè aquests poguessin usar-los. En la dita cultura, era comú que els programadors i usuaris de programari compartissin lliurement els seus programes uns amb altres. Aquest comportament era particularment habitual en alguns dels majors grups d'usuaris de l'època, com DEC-VOS (grup d'usuaris de computadores DEC). A la fi dels 70, les companyies van iniciar l'hàbit d'imposar restriccions als usuaris, amb l'ús d'acords de llicència.

Allà pel 1971, quan la informàtica encara no havia patit el seu gran boom, les persones que feien ús d'ella, en àmbits universitaris i empresarials, creaven i compartien el programari sense cap tipus de restriccions.

Amb l'arribada dels anys 80 la situació va començar a canviar. Les computadores més modernes començaven a utilitzar sistemes operatius privatius, forçant els usuaris a acceptar condicions restrictives que impedien realitzar modificacions al dit programari.

El 1984 Richard Stallman va començar a treballar en el projecte GNU, i un any més tard va fundar la Free Software Foundation (FSF). Stallman va introduir una definició per a programari lliure i el concepte de copyleft, el qual va desenvolupar per donar als usuaris llibertat i per restringir les possibilitats d'apropiació del programari [1].

El terme free, traduït al castellà, significa tant de lliure com gratuït, per això moltes vegades solen confondre's el freeware amb el programari lliure, encara que entre ambdós hi ha notables diferències.

2. Llibertats del programari lliure

D'acord amb tal definició, el programari és lliure si garanteix les llibertats següents:

- "llibertat 0", executar el programa amb qualsevol propòsit (privat, educatiu, públic, comercial, militar, etc.)
- "llibertat 1", estudiar i modificar el programa (per a la qual cosa és necessari poder accedir al codi font)
- "llibertat 2", copiar el programa de manera que es pugui ajudar el veí o a qualsevol
- "llibertat 3", Millorar el programa i publicar les millores

És important assenyalar que les llibertats 1 i 3 obliguen que es tingui accés al codi font. La "llibertat 2" fa referència a la llibertat de modificar i redistribuir el programari lliurement llicenciat sota algun tipus de llicència de programari lliure que beneficii a la comunitat.
El terme programari no lliure s'empra per referir-se al programari distribuït sota una llicència de programari més restrictiva que no garanteix aquestes quatre llibertats. Les lleis de la propietat intel·lectual reserven la majoria dels drets de modificació, duplicació i redistribució per a l'amo del copyright; el programari disposat sota una llicència de programari lliure rescindeix específicament la majoria d'aquests drets reservats.

La definició de programari lliure no contempla l'afar del preu; un eslàgan sovint usat és *lliure com en llibertat, no com en cervesa gratis* o en anglès "Free as 'in' freedom, not as 'in' free beer" (al·ludint a l'ambigüitat del terme anglès "free"), i és habitual veure a la venda CDs de programari lliure com a distribucions Linux. No obstant això, en aquesta situació, el comprador del CD té el dret de copiar-ho i redistribuir-lo. El programari gratis va poder incloure restriccions que no s'adapten a la definició de programari lliure —per exemple, pot no incloure el codi font, pot prohibir explícitament als distribuïdors rebre una compensació a canvi, etc—.
Apèndix 2. Llenguatge de programació Python

1. Història de Python

Python va ser creat a la fi dels 1980 per Guido Van Rossum en CWI als Països Baixos com un successor del llenguatge de programació ABC, capaç de manejar excepcions i interactuar amb el sistema operatiu *Amoeba*.

Van Rossum és el principal autor de Python, i el seu continu rol central a decidir la direcció de Python és reconegut, referint-se a ell com a *Benèvol dictador vitalici* o *Benevolent Dictator for Life* (BDFL).

El 1991, Van Rossum va publicar el codi (versió 0.9.0) en *alt.sources*. En aquesta etapa del desenvolupament ja estaven present classes amb herència, maneig d'excepcions, funcions, i els tipus medul·lars: *list*, *dict*, *str* i així successivament. A més a més a aquest llançament inicial estava present un sistema de mòduls adoptat de Modula-3. El model d'excepcions en Python és semblant al de Modula-3, amb l'addició d'una clàusula *else*. L'any 1994 es va formar *comp.lang.python*, el fòrum de discussió principal de Python, marcant un punt destacat en el creixement del grup d'usuaris d'aquest llenguatge.

Python va aconseguir la versió 1.0 al gener de 1994. Un gran conjunt de característiques incloses en aquest llançament van ser les eines de la programació funcional: *lambda*, *map*, *filter* i *reduce*. Van Rossum va estipular que fa 12 anys endarrere, Python va adquirir *lambda*, *reduce()*, *filter()* i *map()*, cortesia d'un hacker de *Lisp* que les trobava a faltar i que va enviar *patches*. El donant va ser Amrit Prem; No es fa cap esment específic de qualsevol herència de *Lisp* en les notes de llançament.

L'última versió lliure provenint de CWI va ser Python 1.2. El 1995, Van Rossum va continuar el seu treball en Python en la *Corporation for National Research Initiatives* (CNRI) en Reston, Virginia on ell va llançar diverses versions del programari.

Durant l'estada de Van Rossum en CNRI, va llançar la iniciativa *Computer Programming for Everybody* (CP4E), a fi de fer la programació més accessible a més gent, amb un nivell de 'alfabetització' bàsic en llenguatges de programació, similar a l'alfabetització bàsica en anglès i habilitats matemàtiques necessàries per molts treballadors. Python va tenir un rol central en això: a causa del seu foc en una sintaxi neta, ja era idoni, i les metes de CP4E presentaven similituds amb el seu predecessor, ABC. El projecte va ser patrocinat per DARPA. A l'any 2007, el projecte CP4E està inactiu, i mentre Python intenti ser fàcil d'aprendre i no gaire arcaic en la seva sintaxis i semàntica, arribant als no-programadors, no és una preocupació activa.

El 2000, l'equip de desenvolupadors principal de Python es va canviar a BeOpen.com per formar l'equip *BeOpen PythonLabs*. CNRI va demanar que la versió 1.6 fora feta pública, continuant el desenvolupament fins al punt en què l'equip de desenvolupadors va abandonar CNRI. Python 2.0 va ser el primer i únic llançament de BeOpen.com. Després que Python 2.0 fora publicat per BeOpen.com, Guido Van Rossum i els altres desenvolupadors de *PythonLabs* es van unir en *Digital Creations*.

Python 2.0 va prendre una característica major del llenguatge de programació funcional *Haskell: list comprehensions*. La sintaxi de Python per a aquesta construcció és molt similar a la de *Haskell*, excepte per la preferència dels caràcters de puntuació en *Haskell*, i la preferència de Python per paraules claus alfabètiques. Python 2.0 va
introduir a més a més un sistema de recol·lecció d'escombraries capaces de recol·lectar cicles de referències.

Posterior a aquest doble llançament, i després que Van Rossum va deixar CNRI per treballar amb desenvolupadors de programari comercial, va quedar clar que l'opció d'usar Python amb programari disponible sota GPL era molt desitjable. La llicència usada llavors, la *Python License*, inclòa una clàusula estipulant que la llicència estava governada per l'estat de Virginia, que va fer que, sota l'òptica dels advocats de *Free Software Foundation* (FSF), incompatible amb GNU GPL. CNRI i FSF es van relacionar per canviar la llicència de programari lliure de Python per tal de fer-la compatible amb GPL. Aquest any (2001), Van Rossum va ser premiat amb *FSF Award for the Advancement of Free Software*.

Python 1.6.1 és essencialment el mateix que Python 1.6, amb alguns arreglaments de bugs, i amb una nova llicència compatible amb GPL.[10]

Python 2.1 va ser un treball derivat de Python 1.6.1, com també de Python 2.0. La seva llicència va ser anomenada: *Python Software Foundation License*. Tot el codi, documentació i especificacions afegides, des de la data del llançament de la versió alfa de Python 2.1, va tenir com a amo a *Python Software Foundation* (PSF), una organització sense fins de lucre formada l'any 2001, modelada basant-se en *Apache Software Foundation*.

Una innovació major en Python 2.2 va ser la unificació dels tipus en Python (tipus escrits en C), i classes (tipus escrits en Python) dins una jerarquia. Aquesta unificació va aconseguir un model d'objectes de Python pur i consistent. També van ser agregats els generadors que van ser inspirats pel llenguatge *Icon*.

L’abril de 2007, l'última versió de producció de Python és la 2.5.1

2. Característiques

Hi ha molts altres llenguatges de programació, Python presenta una sèrie d'avantatges que ho fan molt atractiu, tant per al seu ús professional com per l’aprenentatge de la programació. Entre les més interessants des del punt de vista didàctic tenim:

- Python és un llenguatge molt expressiu, és a dir, els programes Python són molt compactes: un programa Python sol ser bastant més curt que el seu equivalent en llenguatges com C. (Python arriba a ser considerat per a molts un llenguatge de programació de molt alt nivell.)
- Python és molt llegible. La sintaxi de Python és molt elegant i permet l'escriptura de programes on la seva lectura resulta més fàcil que si utilitzèssim altres llenguatges de programació.
- Python ofereix un entorn interactiu que facilita la realització de proves i ajuda a aclarir dubtes sobre certes característiques del llenguatge.
- L'entorn d'execució de Python detecta molts dels errors de programació que escapan al control dels compiladors i proporciona informació molt rica per detectar-los i corregir-los.
- Python pot usar-se com a llenguatge imperatiu procedimental o com a llenguatge orientat a objectes.
Eina pel desenvolupament de reflectors de conjunts òptics

- Posseeix un ric joc d'estructures de dades que es poden manipular de manera senzilla.

Aquestes característiques fan que sigui relativament fàcil traduir mètodes de càlcul a programes Python.

Python està en un procés de continu desenvolupament per una gran comunitat de desenvolupadors. Aproximadament cada sis mesos es fa pública una nova versió de Python. No és que cada mitjà any es canviï radicalment el llenguatge de programació, sinó que aquest s'enriquï mantenint en la mesura que es pugui la compatibilitat amb els programes escrits per a versions anteriors. Nosaltres explicarem característiques de la versió 2.3 de Python. Un avantatge fonamental de Python és la gratuïtat del seu intèrpret. Es pot descarregar l'intèrpret de la pàgina web http://www.python.org. L’intèrpret de Python té versions per a pràcticament qualsevol plataforma en ús: sistemes PC sota Linux, sistemes PC sota Microsoft Windows, sistemes Macintosh d'Apple, etc.

Perquè us aneu fent a la idea de quin aspecte presenta un programa complet en Python, us presentem el programa que calcula la mitjana de tres números que introduceixe per teclat l’usuari i mostra el resultat per pantalla:

```
# programa 2.3
import math

numerals = [1, 2, 4]
congress = [10, 20, 30]

# es desenreda la llista de numerals
numerals.sort()

# es desenreda la llista de congress
congress.sort()

# es calcula el producte de la llista de numerals

# es calcula el producte de la llista de congress

# es calcula el producte de la llista numerals i congress

# es calcula el resultat final
resultat = product * product1 * product2

print("El resultat final és: ", resultat)
```

3. Tipus de dades

3.1. Enters i flotants

Cada valor utilitzat per Python és d'un tipus determinat. Quan s'executa una operació, Python té en compte el tipus dels operants a l'hora de produir el resultat. Si els dos operants són de tipus enter, el resultat també és de tipus enter, així que la divisió entera entre els enters 3 i 2 produeix el valor enter 1.

Si desitgem obtenir resultats de tipus real, haurem d'usar operants reals. Els operants reals han de portar, en principi, una part decimal, encara que aquesta sigui nul·la.

Hi ha diferències entre enters i reals en Python més enllà que els primers no tinguin decimals i els segons si. El número 3 i el número 3.0, per exemple, són indistingibles en matemàtiques, però si són diferents en Python. Quina diferències hi ha?

- Els enters solen ocupar menys memòria.
- Les operacions entre enters són, generalment, més ràpides.
Així doncs, utilitzarem entres a menys que de debò necessitem números amb decimals.

Hem de precisar quelcom respecte a la denominació dels números amb decimals: la nomenclatura *reals* no és adequada, ja que indueix a pensar en els números reals de les matemàtiques.

En matemàtiques, els números reals poden presentar infinit decimals, i això és impossible en un computador. En treballar amb computadors haurem de conformar-nos amb aproximacions als números reals.

Cal recordar que a dins el computador apareixen seqüències de zeros i uns. Haurem, doncs, de representar internament amb ells les aproximacions als números reals. Per facilitar l’intercanvi de dades, tots els computadors convencionals utilitzen una mateixa codificació, és a dir, representen de la mateixa manera les aproximacions als nombres reals. Aquesta codificació es coneix com IEEE Standard 754 floating point (que es pot traduir per Estàndard IEEE 754 per a coma flotant), així que anomenarem nombres en format de coma flotant o simplement flotants als números amb decimals que podem representar amb l'ordinador.

Un número flotant ha d'especificar-se seguint certes regles. En principi, consta de dos parts: mantissa i exponent. L'exponent se separa de la mantissa amb la lletra ((e)) (o ((E))). Per exemple, el número flotant $2e^3$ (o 2E3) té mantissa 2 i exponent 3, i representa el número $2 \cdot 10^3$, és a dir, 2000.

L'exponent pot ser negatiu: $3.2e^{-3}$ és $3.2 \cdot 10^{-3}$, o sigui, 0.0032. S’ha de tenir en compte que si un número flotant no porta exponent ha de portar part fraccionària. Un parell de regles més: si la part entera del número és nul·la, el flotant pot començar directament amb un punt, i si la part fraccionària és nul·la, pot acabar amb un punt. Vegem un parell d'exemples: el número 0.1 es pot escriure també com .1; d'altra banda, el número 2.0 pot escriure's com 2., és a dir, en ambdós casos el zero és opcional. És possible barrejar en una mateixa expressió dades de tipus diferents.

### 3.2. Valors lògics

Des de la versió 2.3, Python ofereix un tipus de dades especials que permet expressar només dos valors: cert i fals. El valor cert s'expressa amb *True* i el valor fals amb *False*. Són els valors lògics o *booleans*. Aquest últim nom deriva del nom d'un matemàtic, Boole, que va desenvolupar un sistema algebraic basat en aquests dos valors i tres operacions: la conjunció, la disjunció i la negació. Python ofereix suport per a aquestes operacions amb els operadors lògics.

### 4. Operadors lògics i de comparació

Hi ha tres operadors lògics en Python: la *y lògica* o conjunció (*and*), la *o lògica* o disjunció (*or*) i el *no lògic* o negació (*not*).

L'operaor *and* dóna com a resultat el valor cert si, i només si, són certs els seus dos operants.
Aquesta és la seva taula de veritat:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>and</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>operandos</strong></td>
<td><strong>resultado</strong></td>
<td></td>
</tr>
<tr>
<td>izquierdo</td>
<td>derecho</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**Taula A2.1.** Taula de veritat de l’operador *and*.

L’operador *or* proporciona *True* si qualsevol dels seus operants és *True*, i *False* només quan tots dos operants són *False*. Aquesta és la seva taula de veritat:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>or</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>operandos</strong></td>
<td><strong>resultado</strong></td>
<td></td>
</tr>
<tr>
<td>izquierdo</td>
<td>derecho</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**Taula A2.2.** Taula de veritat de l’operador *or*.

L’operador *not* és unari, i proporciona el valor *True* si el seu operador és *False* i viceversa. Aquesta és la taula de veritat:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>not</strong></td>
<td></td>
</tr>
<tr>
<td><strong>operando</strong></td>
<td><strong>resultado</strong></td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

**Figura A2.3.** Taula de veritat de l’operador *not*.

Podem combinar valors lògics i operadors lògics per formar expressions lògiques. Aquí mostrem alguns exemples:
Hi ha una família d'operadors que tornen valors *booleans*. Entre ells tenim als operadors de comparació, que veurem en aquest apartat. Un d'ells és l'operador d'igualtat, que torna *True* si els valors comparats són iguals. L'operador d'igualtat s'escriu amb dos iguals seguits: `==`. Mirem el seu funcionament:

```
>>> True and False
False
>>> not True
False
>>> (True and False) or True
True
>>> True and True or False
True
>>> False and True or True
True
>>> False and True or False
False
```

<table>
<thead>
<tr>
<th>operador</th>
<th>comparación</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>es igual que</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>es distinto de</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>es menor que</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>es menor o igual que</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>es mayor que</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>es mayor o igual que</td>
</tr>
</tbody>
</table>

_Taula A2.4. Operadors de comparació_

A vegades desitgem que l'ordinador recordi certs valors per usar-los més endavant. Per exemple, suposem que desitgem efectuar el càlcul del perímetre i l'àrea d'un cercle de radi 1.298373 m. La fórmula del perímetre és $2 \times r$, on $r$ és el radi, i la fórmula de l'àrea és $\pi \times r^2$. (Aproximarem el valor de $\pi$ amb 3.14159265359.) Podem realitzar ambdós càlculs de la manera següent:

```
>>> 2 * 3.14159265359 * 1.298373
6.1579191568362176
>>> 3.14159265359 * 1.298373 ** 2
5.2960108355249037
```
Hem hagut d’introduir dues vegades els valors de $\pi$ i $r$ perquè en tenir tants decimals és molt fàcil cometre errors. Per solucionar aquest problema podem utilitzar variables.

En la primera línia hem creat una variable de nom $\pi$ i valor 3.14159265359. A continuació, hem creat una altra variable, $r$, i li hem donat el valor 1.298373. L'acte de donar valor a una variable es denomina assignació. En assignar un valor a una variable que no existeix, Python reserva un espai en la memòria, emmagatzema el valor en ell i crea una associació entre el nom de la variable i la direcció de memòria d’aquest espai. Podem representar gràficament el resultat d'aquestes accions així:

```
>>> pi = 3.14159265359
>>> r = 1.298373
>>> 2 * pi * r
8.1579181568392176
>>> pi + r ** 2
5.296010355249037
```

El nom d’una variable és el seu identificador. Hi ha unes regles precises per construir identificadors. Si no es segueixen, direm que l’identificador no és vàlid. Un identificador ha d’estar format per lletres minúscules, majúscules, dígits i/o el caràcter de subratllat (_), amb una restricció: que el primer caràcter no sigui un dígit.

Hi ha una norma més: un identificador no pot coincidir amb una paraula reservada o paraula clau. Una paraula reservada és una paraula que té un significat predefinit i és necessària per expressar certes construccions del llenguatge. Aquí tens una llista amb totes les paraules reservades de Python: `and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `yield`.

Hem de tenir presents que Python distingeix entre majúscules i minúscules, així doncs `area`, `Area` i `AREA` són tres identificadors vàlids i diferents.

5. Assignacions amb operador

Incrementar el valor d’una variable en una quantitat qualsevol és tan freqüent que existeix una forma compacta en Python. L’increment de i pot fer-se de la següent manera:

```
>>> i += 1
```

(No pot haver-hi espais entre el `+` i el `=`.) Es pot incrementar una variable amb qualsevol quantitat, fins i tot amb una que resulti d’haver evaluat una expressió.
Tots els operadors aritmètics tenen una assignació amb operador associada.

6. Variables no inicialitzades

En Python, la primera operació sobre una variable ha de ser l’assignació d’un valor. No es pot utilitzar una variable que no se li hagi assignat un valor prèviament.

L’assignació d’un valor inicial a una variable es denomina inicialització de la variable. Hem de dir, doncs, que en Python no és possible utilitzar variables no inicialitzades.

7. Tipus de dada cadena

Fins al moment hem vist que Python pot manipular dades numèriques de dos tipus: enters i flotants. Però Python també pot manipular altres tipus de dades. Mostrarem ara el tipus de dades que es denomina cadena. Una cadena és una seqüència de caràcters (lletres, números, espais, marques de puntuació, etc.) i en Python es distingeix perquè va tancada entre cometes simples o dobles. Per exemple, 'cadena', 'un altre exemple', "1, 2 1o 3", "...Python" són cadenes. Observa que els espais en blanc es mostren així en aquest text: " " . Ho fem perquè resulti fàcil comptar els espais en blanc quan hi ha més d’un seguit. Aquesta cadena, per exemple, està formada per tres espais en blanc: " " . Les cadenes poden usar-se per representar informació textual: noms de persones, noms de colors, matrícules de cotxe... Les cadenes també poden emmagatzemar-se en variables.

Es possible realitzar operacions amb cadenes. Per exemple, podem sumar cadenes afegint una en una altra.
Aquesta operació no s’anomena suma, sinó concatenació. El símbol utilitzat és +, el mateix que utilitzem quan sumem enters i/o flotants; però encara que el símbol sigui el mateix, no és igual sumar números que concatenar cadenes:

```
>>> 'a' + 'b'
'ab'
>>> nombre = 'Pepe'
>>> nombre + 'Canó'
'PepeCanó'
>>> nombre + 'u' + 'Canó'
'Pepe U Canó'
>>> apellido = 'Canó'
>>> nombre + 'u' + apellido
'Pepe Canó'
```

Per acabar aquesta secció, hi ha un operador de repetició de cadenes. El símbol que el denomina és *, el mateix que per a multiplicar enters i/o flotants. L’operador de repetició necessita dues dades: una de tipus cadena i una altra de tipus entera. El resultat és la concatenació de la cadena amb ella mateixa tantes vegades com indiqui el número enter:

```
>>> 'Hola' * 5
'HolaHolaHolaHolaHola'
>>> '1' * 60
'1'*60
>>> 60 * '1'
```

8. Funcions predefinides

Python proporciona funcions que podem utilitzar en les expressions. Aquestes funcions es diuen que estan predefinides.

La funció abs, per exemple, calcula el valor absolut d’un número. Podem utilitzar-la com en les següents expressions:

```
>>> abs(-3)
3
>>> abs(3)
3
```

El número sobre el que s’aplica la funció s’anomena argument. L’argument de la funció ha d’anar entre parèntesis.

Existeixen moltes funcions predefinides, alguna d’elles són:
• **Float**: conversió a flotant. Si rep un número enter com a argument, retorna el mateix número convertit en un flotant equivalent.

```python
>>> float(3)
3.0
```

La funció `float` també accepta arguments de tipus cadena. Quan se li passa una cadena, `float` la converteix en el número flotant que representa. Però si la cadena no representa un flotant, es produeix un error de tipus `ValueError`.

• **Int**: conversió a enter. Si rep un número flotant com a argument, retorna l’enter que s’obté eliminant la part fraccionària.

```python
>>> int(2.1)
2
>>> int(-2.0)
-2
```

• **Str**: conversió a cadena. Rep un número i retorna una representació d’aquest com a cadena.

```python
>>> str(2.1)
'2.1'
>>> str(234E47)
'2.34e+49'
```

La funció `str` també pot rebre com a argument una cadena, però en aquest cas retorna com a resultat la mateixa cadena.

• **Round**: arrodonir. Pot utilitzar-se amb un o dos arguments. Si s’utilitza amb un sol argument, arrodoneix el número a flotant més pròxim amb part decimal nul·la.

```python
>>> round(2.1)
2.0
>>> round(2.9)
3.0
>>> round(-2.9)
-3.0
>>> round(2)
2.0
```

Aquestes funcions poden formar part d’expressions i els seus arguments poden ser, a la vegada, expressions.

9. Funcions definides en mòduls

Python també proporciona funcions trigonomètriques, logaritmes, etc. Però no estan directament disponibles quan iniciem una sessió. Abans d’utilitzar-les hem
d’indicar a Python que volem utilitzar-les. Per fer-ho, importarem cada funció d’un mòdul.

9.1. El mòdul *math*

Per exemple, volem importar la funció sinus del mòdul matemàtic (*math*):

```python
>>> from math import sin
```

Ara ja podem utilitzar la funció en els nostres càlculs:

```python
>>> sin(0)
0.0
>>> sin(1)
0.841470984808
```

Podem observar que l’argument de la funció sinus s’ha d’expressar en radiants.
Inicialment Python no sap calcular la funció sinus. Quan importem una funció, Python aprèn la seva definició i ens permet utilitzar-la. Les definicions de funcions es troben en mòduls. Les funcions trigonomètriques es guarden en el mòdul matemàtic. Per exemple, la funció cosinus, seria desconeguda per a Python si aquesta no la importéssim, també.

10. Mètodes

Les dades de cert tipus permeten invocar unes funcions especials: els anomenats mètodes. Dels que ja n’hem parlat, només les cadenes permeten invocar mètodes.

Un mètode permet, per exemple, obtenir una versió en minúscules de la cadena sobre a la que es crida.

```python
>>> cadena = 'Un Ejemplo de Cadena'
>>> cadena.lower()
'un ejemplo de cadena'
>>> 'OTRO Ejemplo'.lower()
'otro ejemplo'
```

La sintaxis és diferent a la d’una cria a la funció convencional. El primer que apareix és l’objecte sobre el que es fa la crida. El nom del mètode es separa de l’objecte amb un punt. Els parèntesis oberts i tancats al final són obligatoris.

Existeix una mètode, *upper* (que en anglès vol dir majúscules) que passa tots els caràcters a majúscules.

Un altre mètode, *title*, que passa la inicial de cada paraula a majúscules.

Alguns mètodes accepten parèntesis. El mètode *replace*, per exemple, rep com a argument dues cadenes: un patró i un substitut. El mètode busca el patró dins la cadena sobre la que es cridarà el mètode i substituirà totes les seves aparicions per la cadena substituta.
11. Programes

Fins ara us hem explicat Python en un entorn interactiu: hem mostrat com introduir expressions (i assignacions a variables) i Python les ha avaluat i ens ha proporcionat immediatament els seus respectius resultats.

Però si haguéssim d’utilitzar el sistema únicament d’aquesta manera ens limitaria bastant la nostra capacitat de treball. En Python també es poden crear programes, és a dir, una seqüència d’expressions i assignacions en un fitxer de text i demanar a Python que les executi una rera l’altra. Anomenarem programa al contingut del fitxer de text.

Es poden generar els fitxers en qualsevol editor de text. Nosaltres hem utilitzat un entorn de programació: l’entorn PythonG (veure figura A2.1). Un entorn de programació és un conjunt d’eines que faciliten el treball del programador.

![Figura A2.1. Entorn PythonG](image)

11.1. Lectura de dades del teclat

Hi ha una funció predefinida, raw_input (en anglès significa entrada en brut), que fa el següent: para l’execució del programa i espera a que l’usuari escrigui un text i premi la tecla de retorn de carro; en aquest moment segueix l’execució i la funció retorna una cadena amb el text que ha teclejat l’usuari.

Les cadenes poden utilitzar-se també per a mostrar texts per pantalla en qualsevol moment a través de sentències print.
Una sentència `print` pot mostrar més d’un resultat en una mateixa línia: és necessari separar amb comes to ts els valors que desitgem mostrar. Cada una de les comes es tradueix en un espai de separació

11.2. Comentaris

Una manera facilitar la lectura dels nostres programes consisteix en intercalar comentaris que expliquin la seva funcionalitat. Com que aquests comentaris només tenen per objectiu facilitar la lectura dels programes pels programadors, es poden escriure en l’idioma que vulguem. Quan l’intèrpret Python veu un comentari no fa res amb ell. Els comentaris Python s’inicen amb el símbol `#`: qualsevol text des de `#` fins al final de la línia es considera comentari.

12. Estructures de control

Fins ara hem vist el codi Python com una sèrie de línies que s’executaven una rera l’altre, des de la primera fins a la última i seguint un ordre en el que apareix en el fitxer: el flux d’execució del programa és estrictament seqüencial.

No obstant, és possible alterar el flux d’execució dels programes per fer que:

- Prenguin decisions a partir de les dades i/o resultats intermitjos i, en funció d’aquestes decisions, executin certes sentències i altres no.
- Prenguin decisions a partir de les dades i/o resultats intermitjos i, en funció d’aquestes decisions, executin algunes sentències més d’una vegada.

El primer tipus d’alteració del flux de control es fa amb sentències condicional o de selecció i el segon tipus amb sentències iteratives o de repetició. Les sentències que permeten alterar el flux d’execució s'engloben en les denominades estructures de control de flux (que abreugem amb la terminació estructures de control).

Veurem una forma addicional d’alterar el flux de control que permet senyalar, detectar i tracta els errors que es produeixen en executar un programa: les sentències d’emissió i captura d’excepcions.

12.1. Sentències condicionals

12.1.1. Sentència condicional `if`

La sentència condicional `if` ha d’anar seguida d’una condició.

En Python s’escriu de la següent forma:

```
If condición:
    acció
    acció
    ...
    acció
```
En anglès *if* significa *si*. Per tant, la sentència condicional *if* es traduiria com: si es compleix la condició llavors executa la següent acció (o accions).

### 12.1.2. Sentències condicionals imbricades

Per entendre què són les sentències condicionals imbricades us mostrarem un exemple:

```python
if a != 0:
    x = -b/a
    print 'Solución: ', x
if a == 0:
    if b != 0:
        print 'La ejecución no tiene solución.'
    if b == 0:
        print 'La ejecución tienen infinitas soluciones.'
```

Ens fixarem en les línies que es troben més a la dreta; això voldrà dir que cap d’elles s’executarà si no es compleix la condició que es troba per sobre d’elles, per tant direm que l’execució de la línia que conté la instrucció `print` depèn de la línia que conté la sentència *if*.

Doncs, ara ja hem vist com les estructures de control poden imbricar-se, és a dir, aparèixer unes dins de les altres.

### 12.1.3. En cas contrari (else)

Tal i com hem vist en l’apartat anterior, la sentència condicional *if* ens portava a executar una acció si es complia una determinada condició. Ara podrem executar una altra acció en cas de què la condició del *if* no es compleixi.

```python
if condición:
    acciones
else:
    otras acciones
```

La paraula *else* significa, en anglès, *sinó* o *en cas contrari*. És molt important que respectem les tabulacions: les accions sempre han d’anar una mica a la dreta i el *if* i el *else* alineats a la mateixa columna.

### 12.1.4. Una forma compacta per a estructures condicionals múltiples (elif)

Python permet una forma compacta per expressar fragments de codi de la següent forma:
if condició:
    ...
else:
    if outra condició:
        ...

Un else immediatament seguit per un if pot escriure’s així:

if condició:
    ...
elif outra condició:
    ...

12.2. Sentències iteratives

Python permet indicar que volem la repetició d’un determinat tros de programa de dues formes diferents: mitjançant la sentència while i mitjançant la sentència for. La primera d’elles és més general.

12.2.1. La sentència while

En anglès la paraula while significa mentre. La sentència while s’utilitza així:

while condició:
    acció
    acció
    ...
    acció

Permet expressar en Python accions on el seu significat és:
“mentre es compleixi aquesta condició, repeteix aquesta acció.”

Les sentències que mostren repetició es denominen bucles.

Alerta però amb els bucles sense fi:

Els bucles són molt útils a l’hora de confeccionar programes, però també són perilllosos si no anem en compte: és possible que no finalitzin mai, com mostrem en el següent exemple:

```
i = 0
while i < 10
    print i
```
Com podem comprovar el valor de i sempre val 0 i per tant la condició del while sempre es complirà perquè sempre la i serà més petit que 10. Observem que la i no s’incrementa mai.

12.2.2. El bucle FOR-IN

Hi ha un altre tipus de bucle en Python: el bucle for-in, que es pot traduir com “per a tot element d'una sèrie, fer...” Un bucle for-in presenta el següent aspecte:

```
For variable in serie de valors:
    acció
    acció
    ... 
    acció
```

Un exemple seria:

```
for nombre in ['Pepe', 'Ana', 'Juan']:
    print ('Hola, %s.' % nombre)
```

Ens mostraria per pantalla tres valors de la variable “nombre” perquè la seva sèrie hi ha tres valors.

12.2.3. Ruptura de bucles: BREAK

A vegades ens interessa no arribar al final del bucle perquè ja hem trobat la solució que buscàvem, per tant podem avortar el bucle while.

Hi ha una sentència que permet avortar l’execució d’un bucle des de qualsevol punt del bucle: break (en anglès significa trencar).

Aquesta sentència és una comoditat que ens ofereix el llenguatge. La sentència break també s’utilitza en el bucle for-in. Però cal dir que no és aconsellable abusar del break, a vegades una condició ben expressada en la primera línia del bucle while fa més entenedor un programa.

12.2.4. Imbricament d’estructures

Podem tenir bucles imbricats (un for-in dins d’un altre for-in, etc.), on cada un d’ells farà unes determinades accions. Mostrem un exemple d’imbricament d’estructures:
13. Captura i tractament d’excepcions

Ja sabem que en alguns programes poden aparèixer errors en temps d’execució, és a dir, poden generar excepcions: divisions per zero, intents de calcular arrels quadrades de valors negatius, problemes en operar amb tipus incompatibles (com en sumar una cadena i un enter), etc. Hem presentat l’estructura de control if com un mitjà per controlar aquests problemes i oferir un tractament especial quan convingui. La detecció de possibles errors amb if resulta un tant pesada, perquè modifica sensiblement l'aspecte del programa en omplir-ho de comprovacions.

Hi ha una estructura de control especial per a la detecció i tractament d’excepcions: try-except. La seva forma bàsica d'ús és aquesta:

```
try:
    acció potencialment errónea
    acció potencialment errónea
    ...
    acció potencialment errónea
except:
    acció para tratar el error
    acció para tratar el error
    ...
    acció para tratar el error
```

Podem expressar la idea fonamental de la següent manera:

“Intenta executar aquestes accions i, si es produeix un error, executa immediatament aquestes altres accions”.

14. Tipus estructurats: seqüències

De moment hem vist dades de tres tipus diferents: enters, flotants i cadenes. Els dos primers són tipus de dades escalars. Les cadenes, en canvi, són tipus de dades seqüencials. Un dada de tipus escalar és un element únic, atòmic. En canvi, una dada de

```python
limite = int(raw_input('Dame un número: '))
cor num in range (1, limite+1):
    creo_que_es_primo = True
    for divisor in range(2,num):
        if num % divisor == 0:
            creo_que_es_primo = False
            break
    if creo_que_es_primo:
        print num
```
tipus seqüencial es composa d'una successió d'elements i una cadena és una successió de caràcters. Les dades de tipus seqüencial són dades estructurades. En Python és possible manipular les dades seqüencials de diferents maneres, facilitant així l'escriptura de programes que manipulen conjunts o sèries de valors.

Més endavant mostrarem les llistes. Una llista és una successió d'elements de qualsevol tipus. Finalment, veurem com definir i manipular matrius: disposicions bidimensionals d'elements. Python no incorpora un tipus de dades natives per a matrius, així que les construirem com llistes de llistes.

14.1. Cadenes

Una cadena és una successió de caràcters tancada entre cometes (simples o dobles). Python ofereix una sèrie d'operadors i funcions predefinits que manipulen cadenes o tornen cadenes com a resultat. Algunes funcions són:

- **Operador +** (concatenació de cadenes): accepta dues cadenes com a operants i torna la cadena que resulta d'unir la segona amb la primera.
- **Operador ** (repetició de cadena): accepta una cadena i un enter i torna la concatenació de la cadena amb si mateixa tantes vegades com indica el enter.
- **Operador %** (substitució de marques de format): accepta una cadena i una o més expressions (entre parèntesis i separades per comas) i torna una cadena en què les marques de format (sequències com %d, %f, etc.) es substitueixen pel resultat d'avaluar les expressions.
- **Int**: rep una cadena on el contingut de la qual és una seqüència de dígits i torna el número enter que descriu.
- **Float**: accepta una cadena on el contingut de la qual descriu un flotant i torna el flotant en qüestió.
- **Str**: se li passa un enter o flotant i torna una cadena amb una representació del valor com a seqüència de caràcters.
- **Ord**: accepta una cadena composta per un únic caràcter i torna la seva codificació ASCII (un enter).
- **Chr**: rep un enter (entre 0 i 255) i torna una cadena amb el caràcter que té aquest enter com a codi ASCII.

Podem manipular cadenes mitjançant mètodes que els són propis:

- **a.lower()** (pas a minúscules): torna una cadena amb els caràcters convertits en minúscules.
- **a.upper()** (pas a majúscules): torna una cadena amb els caràcters convertits en majúscules.
- **a.capwords()** (pas a paraules amb inicial en majúscula): torna una cadena en què tota paraula comença en majúscula.

14.1.1. Longitud d’una cadena

Per conèixer la longitud d’una cadena existeix la funció **len** (abreviatura en anglès, que vol dir longitud), és a dir, retorna el número de caràcters que està formada la cadena. Es tracta d’una funció predefinida, així que podem utilitzar-la directament:
La cadena ‘ ‘ s’anomena cadena buida i té longitud zero. No hem de confondre la cadena buida ‘ ‘, amb la cadena que conté un espai en blanc, ‘ _ ‘, perquè encara que s’assemblin no són iguals. La cadena que conté un espai en blanc ens retornarà longitud 1.

14.1.2. Indexació

Podem accedir a cadascun dels caràcters d’una cadena utilitzant un operador d’indexació. L’índex de l’element que volem accedir ha de tancar-se entre claudàtors. Si a és una cadena, a[i] és el caràcter que ocupa la posició i+1. Hem de tenir en compte que el primer element té índex zero.

14.1.3. Recorregut de cadenes

Una propietat interessant de les dades seqüencials és que es poden recórrer d’esquerra a dreta amb un bucle for-in. Per exemple, el següent bucle recórrer els caràcters d’una cadena d’un en un i d’esquerra a dreta:

Tenim una forma alternativa de recórrer els elements d’una cadena: recorrent el rang de valors que pren el seu índex i indexant cada un d’ells.

14.1.4. Referències a cadenes

Direm que a apunta al valor 2 i que b apunta al valor 3.25. La fletxa rep el nom de punter o referència.
Amb les cadenes representarem els valors desglossant cada un dels seus caràcters en una caixa individual com un índex associat. El resultat d’una assignació com aquesta:

```
>>> c = 'Una_cadena'
```

es representarà de la següent manera:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>n</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>e</td>
<td>n</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
```

Direm que la variable `c` apunta a la cadena “Una cadena”, que és una seqüència de caràcters. La cadena buida no ocupa cap casella de memòria i la representem gràficament d’una manera especial.

Hem de tenir en compte que una assignació únicament altera el valor d’un punter. Però altres operacions amb cadenes cometen la reserva de memòria nova. Podem prendre com a exemple l’operador de concatenació. La concatenació pren dues cadenes i forma una cadena nova resultant de la unió de les dues, és a dir, reserva memòria per a una nova cadena.

### 14.2. Llistes

El concepte de seqüència és molt potent i no es limita a les cadenes. Python ens permet definir seqüències de valors de qualsevol tipus. Per exemple, podem definir seqüències de números entiers o flotants, o fins i tot de cadenes. Parlem llavors de llistes. En una llista podem, per exemple, registrar les notes dels estudiants d’una classe, l’evolució de la temperatura hora a hora, els coeficients d’un polinomi, la relació de noms de persones assistents a una reunió, etc.

Python segueix una notació especial per representar les llistes. Els valors d’una llista deuen estar tancats entre claudàtors i separats per comes.

```
>>> [1, 2, 3]
[1, 2, 3]
```

Els elements que formen una llista també poden ser cadenes i també podem utilitzar expressions per a calcular el valor de cada element d’una llista.

Python guarda les llistes de la mateixa manera que les cadenes: mitjançant referències (punters) a la seqüència d’elements. Per això, l’assignació a una variable del contingut d’una altra variable que guarda una (referència a una) llista suposa la còpia de, únicament, la seva referència, així que totes dues acaben apuntat a la mateixa zona de memòria.
Un avantatge de Python és que proporciona operadors i funcions similars per treballar amb tipus de dades semblants. Les cadenes i les llistes tenen quelcom en comú: ambdues són seqüències de dades, així doncs, molts dels operadors i funcions que treballen sobre cadenes també ho fan sobre llistes. Per exemple, la funció \texttt{len}, aplicada sobre una llista, ens diu quants elements la componen:

\begin{verbatim}
>>> a = [1, 2, 3]
>>> len(a)
3
>>> len([0, 1, 10, 5])
4
>>> len([10])
1
\end{verbatim}

La longitud de la llista buida és 0. L’operador \texttt{+} concatena llistes i l’operador \texttt{*} repeteix un número determinat de vegades una llista.

Hem de tenir en compte que tant l’operador \texttt{+} com \texttt{*} generen noves llistes, sense modificar les originals.

### 14.2.1. Comparació de llistes

Els operadors de comparació també treballen amb llistes. Sembla clar com es comportaran operadors com el d’igualtat (==) o el de desigualtat (!=):

- si les llistes són de talla diferent, resolent que les llistes són diferents;
- i si mesuren el mateix, comparant element a element d’esquerra a dreta i resolent que les dues llistes són iguals si tots els seus elements són iguals, i diferents si hi ha algun element diferent.

Fem un parell de proves amb l’intèrpret de Python:

\begin{verbatim}
>>> [1, 2, 3] == [1, 2]
False
>>> [1, 2, 3] == [1, 2, 3]
True
>>> [1, 2, 3] == [1, 2, 4]
False
\end{verbatim}

Els operadors <, >, <= i >= també funcionen amb llistes. Com? De la mateixa manera que amb les cadenes, perquè al cap i a la fi, tant cadenes com llistes són seqüències. Agafem com a exemple l’operador <, en comparar les llistes \([1, 2, 3]\) i \([1, 3, 2]\), és a dir, en avaluar l’expressió \([1, 2, 3] < [1, 3, 2]\). Es comença per comparar els primers elements d’ambdues llistes. Com no és cert que \([1 < 1\), passem a comparar els respectius següents elements. Com \(2 < 3\), el resultat és \texttt{True}, sense necessitat d’efectuar cap comparació addicional.
14.2.2. Afegir elements en una llista

Podem afegir elements a una llista, això és, fer-la créixer. Per afegir un element a la nostre llista, aquest element ha de pertànyer a una llista i per fer-ho podem utilitzar l’operador + (operador de concatenació).

```
>>> a = a + [4]
>>> a
[1, 2, 3, 4]
```

Existeix un altre mètode efectiu per afegir elements en una llista: mitjançant el mètode `append` (que en anglès vol dir *afegir*).

```
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

Hi ha una diferència fonamental entre utilitzar l’operador de concatenació + i utilitzar `append`: la concatenació crea una nova llista copiant els elements de les llistes que participen com a operants i `append` modifica la llista original.

14.2.3. Eliminar elements d’una llista

També podem eliminar elements d’una llista. Per fer-ho utilitzarem la sentència `del` (abreviatura de delete, que en anglès significa esborrar). S’ha d’indicar quin element es desitja eliminar immediatament després de la paraula `del`.

```
>>> del a[1]
```

La sentència `del` no produeix una còpia de la llista sense la casella esborrada, sinó que modifica directament la llista sobre la que opera.

**Figura A2.2.** Elements pertanyents a ‘a’ abans d’executar la paraula `del`.

**Figura A2.3.** Elements pertanyents a ‘a’ després d’executar la paraula `del`.

14.3. Passar de cadenes a llistes i viceversa

Moltes vegades voldrem convertir cadenes en llistes i viceversa. Python ens ofereix una sèrie de utilitats que val la pena conèixer si volem estalviar-nos moltes hores de programació.

Una acció freqüent consisteix en obtenir una llista amb totes les paraules d’una cadena. Mostrem un exemple per saber com es fa:

```
>>> 'uno, dos, tres'.split()
['uno', 'dos', 'tres']
```
14.4. Matrius

En l’àmbit matemàtic, una matriu és una taula rectangular de nombres o, més generalment, una taula consistent en quantitats abstractes que poden sumar-se i multiplicar-se. Les línies horitzontals d’una matriu es denominen files i les línies verticals es denomen columnes. Una matriu amb m files i n columnes s’anomena matriu de m x n, on ‘m’ i ‘n’ són les seves dimensions. Les dimensions d’una matriu sempre es donen amb el nombre de files primer i el nombre de columnes després.

Les matrius són disposicions bidimensionals de valors. En notació matemàtica, una matriu es denota tancant entre parèntesis els valors, que es disposen en files i columnes. Mostrem com és una matriu M:

\[
M = \begin{pmatrix}
1 & 2 & 3 \\
2 & 12 & 6 \\
1 & 0 & -3 \\
0 & -1 & 0
\end{pmatrix}
\]

Aquesta matriu té 4 files i 3 columnes, amb la qual cosa abreviem dient que és una matriu de dimensió 4x3.

Les llistes permeten representar sèries de dades en una sola dimensió. Amb una llista de números no es pot representar directament una matriu, però sí amb una llista de llistes.

En la notació matemàtica l’element que ocupa la fila i-èssima i la columna j-èssima de la matriu M es representa amb la \( M_{i,j} \). Per exemple, l’element d’una matriu que ocupa la cel·la de la fila 1 i la columna 2 es denota amb \( M_{1,2} \). Però si desitgem accedir a aquest element en el matriu Python M, hem de tenir en compte que Python sempre compte des de zero, així que la fila tindrà índex 0 i la columna tindrà índex 1.

14.4.1. Creació de matrius

Crear una matriu consisteix, doncs, en crear una llista de llistes. Si desitgem crear una matriu nul·la (una matriu on els components sigui tots iguals a 0) de mida 2x2, serà suficient escriure:

\[
M = [ [1, 2, 3], [2, 12, 6], [1, 0, -3], [0, -1, 0] ]
\]

\[
M = \begin{pmatrix}
1 & 2 & 3 \\
2 & 12 & 6 \\
1 & 0 & -3 \\
0 & -1 & 0
\end{pmatrix}
\]

Però si desitgem accedir a aquest element en el matriu Python M, hem de tenir en compte que Python sempre compte des de zero, així que la fila tindrà índex 0 i la columna tindrà índex 1.

14.4.1. Creació de matrius

Crear una matriu consisteix, doncs, en crear una llista de llistes. Si desitgem crear una matriu nul·la (una matriu on els components sigui tots iguals a 0) de mida 2x2, serà suficient escriure:

\[
M = [ [0, 0], [0, 0] ]
\]
14.4.2. Lectura de matrius

Si desitgem llegir una matriu de mida determinada, podem crear una matriu nul·la com hem vist en l’apartat anterior i, a continuació, omplir cada un dels seus components:

```python
# Creamos una matriz nula...
M = []
for i in range(m):
    M.append([0] * n)
# ... y leemos su contenido de teclado
for i in range(m):
    for j in range(n):
        M[i][j] = float(input('Dame el componente (%d,%d): '%(i, j)))
```

14.5. Funcions

Algunes funcions es troben predefinides (abs, assault, etc.) mentre que d’altres tenen d'importar-se de mòduls abans de poder ser utilitzades (per exemple, sin i cos s'importen del mòdul math). En aquest apartat mostrarem com definir les nostres pròpies funcions. Definint noves funcions estarem ensenyant a Python a fer càlculs que inicialment no sap fer i, d'alguna manera, adaptant el llenguatge de programació al tipus de problemes que desitgem resoldre, enriquint perquè el programador pugui executar accions complexes d'una manera senzilla: cridant a funcions des del seu programa.

Ja hem utilitzat vist com utilitzar fitxers que contenen funcions i variables de valor predefinit que podem importar en els nostres programes. En aquest apartat mostrarem com crear els nostres propis mòduls, de manera que reutilitzar les nostres funcions en diversos programes resultarà extremadament senzill: només caldrà importar-les.

14.5.1. Utilització de funcions

Denominarem activar, invocar o cridar a una funció a l'acció d'utilitzar-la. Les funcions que hem aprés a invocar reben zero, un o més arguments separats per comas i tancats entre un parell de parèntesis i poden tornar un valor o no tornar res.

```python
>>> abs(-3)
3
>>> abs(round(2.45, 1))
2.5
>>> from sys import exit
>>> exit()
```

Podem cridar a una funció des d'una expressió. Com que el resultat té un tipus determinat, hem d'estar atents que aquest sigui compatible amb l'operació i tipus dels operants amb què es combina.
14.5.2. Definició de funcions

Veurem el mode en què podem definir (i utilitzar) les nostres pròpies funcions Python. Veurem en primer lloc com definir i cridar a funcions que tornen un valor i passarem, després, a veure els denominats procediments: funcions que no tornen cap valor.

14.5.2.1. Definició i utilització de funcions amb un sol paràmetre

Començarem definint una funció senzilla, una que rep un número i retorna el quadrat d’aquest número. El nom que donarem a la funció serà “cuadrado”. Observem el següent exemple:

```python
def cuadrado(x):
    return x ** 2
```

Amb aquest exemple acabem de definir la funció “cuadrado” que s’aplica sobre un valor al que anomenem x i retorna un número: el resultat d’elevar x al quadrat. En el programa apareixen dues noves paraules reservades: `def` i `return`. La paraula `def` és abreviatura de `define` i `return` significa retorna en anglès. Podríem llegir el programa anterior com “defineix quadrat de x com el valor que resulta d’elevar x al quadrat”.

Les regles per donar nom a les funcions i als seus paràmetres són les mateixes que seguim per donar nom a les variables: només es poden usar lletres (de l’alfabet anglès), dígits i el caràcter de subratllat; la primera lletra del nom no pot ser un número; i no es poden usar paraules reservades. Però no s’ha de donar el mateix nom a una funció i a una variable. En Python, cada nom ha d’identificar clarament un únic element: una variable o una funció.

En definir una funció “cuadrado” és com si haguéssim creat una màquina de calcular quadrats. Des de la òptica del seu ús, podem representar la funció com una caixa que transforma un dada d'entrada en un dada de sortida:
14.5.2.2. Definició i ús de funcions amb més d’un paràmetre

No totes les funcions tenen un sol paràmetre. Anem a definir ara una funció amb dos paràmetres: una funció que retorna el valor de l’àrea d’un rectangle donada la seva altura i la seva amplada:

$$\text{area_rectangulo}$$

\[
\begin{array}{c}
\text{altura} \\
\text{anchura} \\
\end{array} \rightarrow \text{altura}\times\text{anchura}
\]

Observem que els diferents paràmetres d’una funció s’han de separar per comes. En utilitzar la funció, els arguments també s’han de separar per comes.

```python
def area_rectangulo(altura, anchura):
    return altura * anchura
print(area_rectangulo(3, 4))
```

14.5.2.3. Procediments: funcions sense retornar valor

No totes les funcions tornen un valor. Una funció que no torna un valor es denomina procediment. Una funció que no torna res pot mostrar missatges o resultats per pantalla. Mostrar un missatge per pantalla és un efecte secundari.

14.5.3. Variables locals i variables globals

Podem observar que en el cos de les funcions és possible definir i utilitzar variables. Ara veurem algunes propietats de les variables definides en el cos d’una funció i en què es diferencien de les variables que definim fora de qualsevol funció, és a dir, en el què anomenem programa principal.

```python
def area_triangulo(a, b, c):
    s = (a + b + c) / 2.0
    return sqrt(s * (s - a) * (s - b) * (s - c))
```

Si ens fixem en l’exemple veurem que la variable s només existeix en el cos de la funció. Fora d’aquest cos, s no està definida. Si volem accedir a s des del programa principal ens provocarà un error.

Les variables que només existeixen en el cos d’una funció es denominen variables locals. A la resta de variables es diuen variables globals. També els paràmetres formals d’una funció es consideren variables locals, així que no podem accedir al seu valor fora del cos de la funció.
En acabar l’execució de la funció, les variables locals deixen d’existir.

14.5.4. Accés a variables globals desde funcions

De moment podríem pensar que en el cos d’una funció només es poden utilitzar variables locals però no és cert. Dins d’una funció també podem consultar i modificar variables globals. Python utilitza una regla simple per decidir si una variable utilitzada en una funció és local o global: si se li assigna un valor, és local; sinó, és global.

15. Tipus estructurats: registres

El conjunt de tipus de dades Python que hem vist es divideix en tipus escalars (enters i flotants) i tipus seqüencials (cadenes i llistes). En aquest tema veurem com definir i utilitzar tipus de dades definides per nosaltres mateixos agregant tipus de dades de diferent o igual naturalesa. Per exemple, podrem definir un nou tipus que reuneixi un enter i dues cadenes o un de diferent amb una llista i un flotant. Les dades d'aquests nous tipus reben el nom de registres. Els registres ens permeten modelar objectes del món real que han de descriure’s mitjançant una col·lecció d’informacions, com a persones (descrites per nom, cognoms, DNI, edat, etc.), cançons (descrites per títol, autor, intèrpret, estil, etc.), dates (descrites per dia, mes i any), etc.

15.1. Associant dades relacionals

Suposem que en un programa utilitzéssim el nom, el DNI i l’edat de dues persones. En principi necessitariem tres variables per guardar les dades de cada persona: dues variables amb valors de tipus cadena (el nom i el DNI) i una altra amb un valor de tipus enter (l’edat). Cada dada es trobaria guardada en una variable diferent i completament independent de les altres. El programador hauria de recordar en cada moment quina variable està relacionada entre elles per utilitzar-les coherament. Existeix una alternativa per treballar amb un grup de tres variables independents per persona: definir una “persona” com una llista amb tres elements. En cada element de la llista guardarem un dels seus valors, sempre en el mateix ordre:

```
juan = ['Juan Paz', '12345678Z', 19]
ana = ['Ana Mir', '23456789D', 18]
```

Treballar d’aquesta manera permet que les dades de cada persona estiguin agrupades. Però també se’ns fa incómodo el seu ús perquè hem de recordar l’índex 0 accedeix al nom, l’índex 1 al DNI i l’índex 2 a l’edat.
15.2. Registres

Si declarem una variable de tipus “persona” agruparà les tres informacions d’una persona (el seu nom, dni i l’edat) de manera semblant a com ho fa una llista. La diferència es trobarà en la forma amb la què accedim a cada informació: ara utilitzarem la notació “persona.nom” per accedir al nom de la persona. Fixem-nos que quan diem que persona és un tipus de dades no parlem de variables. Per facilitar la distinció entre tipus de dades i variables, podem utilitzar sempre inicials en majúscules pels identificadors del tipus de dades. Només és una convenció nostra.

15.3. Definició de nous tipus de dades

La definició d’un nou tipus de dades és equivalent, en cert sentit, a la definició d’una nova funció. La definició d’un tipus registre ensenyà a Python com construir objectes d’un nou tipus de dades, però no constitueix en si mateixa la construcció d’un d’aquests objectes. Vegem com definir el tipus Persona:

```python
Class persona(record)
   nombre = ''
   dni = ''
   edad = 0
```

Observeu que la definició comença amb la paraula `class` (en anglès, `classe`), i segueix amb l’identificador del nou tipus (persona) i la paraula `record` entre parèntesis. La primera línia acaba amb dos punts, així que les següents línies de la definició apareixen més a la dreta. Cada línia indica el nom d’un dels camps o atributs del registre i, mitjançant una assignació, el seu valor per defecte.
Podem representar els registres gràficament: