```
                (**CODE SOURCE complet en Caml pour le TIPE sur le Rubik's
Cube**)

        (*** 1) Structure de groupe, loi de composition, mouvements
ÈlÈmentaires *)

type permutation == int vect;; (*une permutation de Sn est reprÈsentÈ par
le tableau d'entiers [sigma(1); ... ; sigma(n)] *)
type indices == int vect ;;

let (rond : permutation -> permutation -> permutation) = fun sc1 sc2 ->
(*loi de composition rond pour permutations*)
let n = vect_length sc1 in let sc = make_vect n 0 in
for i=0 to (n-1) do
sc.(i) <- sc1.(sc2.(i)-1) (*dÈcalage d'indicage des tableaux sous Caml*)
done;
sc;;

(*composition des indices de rotation des coins : IRC*)
let (irc_comp : indices -> indices -> permutation -> indices) = fun im1 im2
sig ->
let im12 = make_vect 8 0 in
for i=0 to 7 do
let a=(im1.(i)+im2.(sig.(i)-1)) in
(*on teste en premier le cas le plus courant, pour la rapiditÈ ; en effet,
le cas abs a = |a| < 2 se produit dans 7 cas sur 9*)
if (abs a) < 2 then
im12.(i) <- a
else
if a > 0 then
im12.(i) <- -1 else im12.(i) <- 1
done; im12;;

(*composition des indices de rotation des milieux : IRM*)
let (irm_comp : indices -> indices -> permutation -> indices) = fun im1 im2
sig ->
let im12 = make_vect 12 0 in
for i=0 to 11 do
im12.(i) <- ((im1.(i)+im2.(sig.(i)-1)) mod 2)
done;
im12;;

type rubik =  { sc : permutation; sm : permutation ; irc : indices ; irm :
indices };;
(* Une position du cube est codÈe par 4 tableaux, sous la forme :
{SigmaC;SigmaM;IRC;IRM}*)

(* 'x' compose deux positions du cube en appliquant la formule de
composition*)
let (x : rubik -> rubik -> rubik) = fun p1 p2 ->
{sc = rond p2.sc p1.sc ; sm = rond p2.sm p1.sm ; irc = irc_comp p1.irc
p2.irc p1.sc ;
irm = irm_comp p1.irm p2.irm p1.sm};;

(* on rend la fonction de composition infixe pour plus de commoditÈ*)
#infix "x";;

(*V, W, O.... reprÈsentent les mouvements ÈlÈmentaires d'un quart de tour
trigonomÈtrique d'une face de chaque couleur. Je les ai recherchÈ
manuellement en regardant comment la rotation de chaque face affecte les
cubes ÈlÈmentaires (coins et milieux), avec mes notations et conventions*)
let V = {sc = [|2;3;4;1;5;6;7;8|]; sm = [|2;3;4;1;5;6;7;8;9;10;11;12|]; irc
= [|0;0;0;0;0;0;0;0|];
irm = [|0;0;0;0;0;0;0;0;0;0;0;0|]} and
W = {sc = [|5;1;3;4;6;2;7;8|]; sm = [|1;5;3;4;10;2;7;8;9;6;11;12|]; irc =
[|-1;1;0;0;1;-1;0;0|];
irm = [|0;0;0;0;0;1;0;0;0;1;0;0|]} and
O = {sc = [|1;6;2;4;5;7;3;8|]; sm = [|1;2;6;4;5;11;3;8;9;10;7;12|]; irc =
[|0;-1;1;0;0;1;-1;0|];
irm = [|0;0;0;0;0;0;1;0;0;0;1;0|]} and
J = {sc = [|1;2;3;4;8;5;6;7|]; sm = [|1;2;3;4;5;6;7;8;12;9;10;11|]; irc =
[|0;0;0;0;0;0;0;0|];
irm = [|0;0;0;0;0;0;0;0;0;0;0;0|]} and
R = {sc = [|4;2;3;8;1;6;7;5|]; sm = [|8;2;3;4;1;6;7;9;5;10;11;12|]; irc =
```

```
[|1;0;0;-1;-1;0;0;1|];
irm = [|0;0;0;0;1;0;0;0;1;0;0;0|]} and
B = {sc = [|1;2;7;3;5;6;8;4|]; sm = [|1;2;3;7;5;6;12;4;9;10;11;8|]; irc =
[|0;0;-1;1;0;0;1;-1|];
irm = [|0;0;0;0;0;0;0;1;0;0;0;1|]};;
(*r, w, o.... reprÈsentent les mouvements ÈlÈmentaires d'un quart de tour
horaire d'une face de chaque couleur ;
R2, W2, O2...  reprÈsentent les mouvements ÈlÈmentaires d'un demi-tour
d'une face de chaque couleur ;
id reprÈsente le cube fait, c-‡-d l'identitÈ*)
let r = R x R x R and
w = W x W x W and
j = J x J x J and
o = O x O x O and
b = B x B x B  and
v = V x V x V and
W2 = W x W and
R2 = R x R and
J2 = J x J and
O2 = O x O and
B2 = B x B and
V2 = V x V and
id =W x W x W x W;;

(*compo_list compose dans l'ordre une sÈrie de mouvements ÈlÈmentaires
contenus dans une liste*)
let rec (compo_list: rubik list -> rubik) = fun
[mouvement] -> mouvement
|(mouvement::suite) -> mouvement x (compo_list suite)
|[] -> invalid_arg "Liste vide dans compo_list";;

(* [compo pos n] compose une position ëposí n fois avec elle mÍme. Le temps
d'exÈcution est en O(ln n).
Le principe utilisÈ est analogue ‡ celui de l'exponentiation rapide*)
let rec compo = fun
| _ 0 -> id
| mvt n  when n mod 2 = 0 -> let k = compo mvt (n/2) in k x k
| mvt n -> mvt x (compo mvt (n-1));;

(*je me suis rendu compte par la suite des confusions apportÈes par ma
notation qui dÈpend de la disposition des couleurs sur les diffÈrents
Rubik's Cube ; la 2Ëme convention adoptÈe est la suivante: H pour "haut", P
pour "bas" (car B est dÈja occupÈ, prononcer "bas" d'une voix nasale), D
pour "droite", G pour "gauche", A pour "arriËre", F pour "face" ; les
correspondances entre les 2 notations ont ÈtÈ Ètablies par rapport ‡ la
position de rÈfÈrence du Cube, face verte en haut et face blanche face ‡
nous *)
let H = V and P = J and D = O and G = R and A = B and F = W;;
let h = v and p = j and d = o and g = r and a = b and f = w;;
let H2 = V2 and P2 = J2 and D2 = O2 and G2 = R2 and A2 = B2 and F2 = W2;;

let tab = [|W;J;B;V;O;R;w;j;b;v;o;r|];; (* Èquivalence avec la deuxiËme
notation : [|F;P;A;H;D;G;f;p;a;h;d;g|] *)



        (*** 2) Fonctions mathÈmatiques *)

type cycle == int vect ;;
type transposition == int * int ;;

(*cette fonction donne l'ordre d'une position, qui est infÈrieur ou Ègal ‡
1260 pour une position accessible.*)
let ordre position =
let rec ord2 = fun
pos when pos = id -> 1
|pos -> 1 + (ord2 (pos x position))
in
ord2 position;;

(*cette fonction donne l'inverse d'une position*)
let inverse2 p = compo p 166319;; (*version peu performante mais simple,
rÈsultat thÈorique mis en pratique.*)
```

```
(*donne la signature d'une permutation de Sn reprÈsentÈe par sa matrice ;
on utilise la formule avec la dÈcomposition en cycles disjoints ; on aurait
pu faire une version plus performante, mais le programme ne nÈcessite pas
un coˆt particuliËrement faible et l'on a donc mis en avant la simplicitÈ*)
let signature (tab : permutation) = let s = ref 1 and l = vect_length tab
in
for i = 0 to (l-2) do
let cyc = ref 1 and courant = ref (tab.(i) - 1) in
(*attention aux dÈcalages de 1, l'indexation d'un tableau sous CAML
commence ‡ 0*)
while !courant > i do
(*on compte une seule fois chaque cycle en commenÁant par son ÈlÈment le
plus ‡ gauche dans le tableau ; si lorsque l'on parcourt le cycle, on
rencontre un ÈlÈment plus petit que celui par lequel on a commencÈ, c'est
que le cycle a dÈj‡ ÈtÈ comptÈ et l'on abandonne donc le calcul*)
let k = tab.(!courant) - 1 in
if k >= i then begin courant:=k;cyc:= - (!cyc) end
else begin cyc:=1; courant:=(i-1) end
done;
s:=(!s)*(!cyc)
done;
!s;;

(*donne la dÈcomposition en produit de cycles ‡ supports disjoints d'une
permutation de Sn reprÈsentÈe par sa matrice*)
let (cycles : permutation -> cycle list) = fun tab ->
let s = ref [] and l = vect_length tab in
for i = 0 to (l-2) do
let cyc = ref [i+1] and courant = ref (tab.(i) - 1) in
(*attention aux dÈcalages de 1, l'indexation d'un tableau sous CAML
commence ‡ 0*)
while !courant > i do
(*on compte une seule fois chaque cycle en commenÁant par son ÈlÈment le
plus ‡ gauche dans le tableau, selon le principe prÈcÈdemment Ètabli*)
let k = tab.(!courant) - 1 in
if k >= i then begin cyc:=(!cyc)@[!courant+1];courant:=k end
else begin cyc:=[]; courant:=(i-1) end
done;
if list_length !cyc > 1  then
s:=(!s)@[!cyc] done;
map vect_of_list !s;;

(* 'cycles2' est une version en O(n) mais plus lourde du point de vue de la
programmation ; on marque les valeurs par lesquelles on est dÈj‡ passÈ. La
mÍme astuce peut Ítre utilisÈ dans 'signature' mais ne prÈsente pas
d'intÈrÍt *)
let (cycles2 : permutation -> cycle list) = fun tab -> let s = ref [] and
tab_courant = copy_vect tab and l = vect_length tab in
for i = 0 to (l-2) do
if tab_courant.(i) <> 0 then begin
let cyc = ref [i+1] and courant = ref (tab.(i) - 1) in
(*attention aux dÈcalages de 1, l'indexation d'un tableau sous CAML
commence ‡ 0*)
while !courant > i do
(*on compte une seule fois chaque cycle en commenÁant par son ÈlÈment le
plus ‡ gauche dans le tableau, selon le principe prÈcÈdemment Ètabli*)
tab_courant.(!courant) <- 0;
let k = tab.(!courant) - 1 in
if k >= i then begin cyc:=(!cyc)@[!courant+1];courant:=k end
else begin cyc:=[]; courant:=(i-1) end;
done;
if !cyc <> [] then
s:=(!s)@[!cyc];
tab_courant.(i) <- 0;
end
done;
map vect_of_list !s;;
let (cycle2transpo : cycle -> transposition list) = fun perm ->
(*transforme un cycle en un produit de transpositions*)
let l = ref [] in for i = 0 to (vect_length perm - 2) do
l:=(perm.(i),perm.(i+1))::(!l) done; !l;;
let (transpo2_3cycles : transposition -> transposition -> cycle list) = fun
```

```
(a,b) (c,d) -> (*transforme 2 transpositions en un produit de 3-cycles*)
if b = c then [[|d;b;a|]] else
if a = d then [[|c;d;b|]] else
[ [|a;b;d|] ; [|d;a;c|] ];;

(*dÈcompose une permutation paire en un produit de 3-cycles*)
let en_3cycles (perm : permutation) = let  k = vect_of_list (flat_map
cycle2transpo (cycles2 perm)) and l = ref [] in
for i = 0 to (vect_length k / 2 - 1) do
l:=(!l)@(transpo2_3cycles k.(2*i) k.(2*i+1))
done;
(!l : cycle list);;

let c3_2perm8 (c : cycle) = let k = [|1;2;3;4;5;6;7;8|] in (*transforme un
3-cycle en une permutation*)
k.(c.(0)-1) <- c.(1);
k.(c.(1)-1) <- c.(2);
k.(c.(2)-1) <- c.(0); (k : permutation);;

let c3_2perm12 (c : cycle) = let k = [|1;2;3;4;5;6;7;8;9;10;11;12|] in
(*transforme un 3-cycle de S12 en une permutation de S12*)
k.(c.(0)-1) <- c.(1);
k.(c.(1)-1) <- c.(2);
k.(c.(2)-1) <- c.(0); (k : permutation);;

(*donne la trace d'une position du Rubik's Cube sous la forme d'une liste
de 3 entiers ; cette position est accessible ssi sa trace vaut [0;0;0]*)
let trace pos =
let sumic = ref 0 and sumim = ref 0 in
for i = 0 to 7 do
sumic:=!sumic+(pos.irc).(i)
done;
for i = 0 to 11 do
sumim:=!sumim+(pos.irm).(i)
done;
[signature pos.sc - signature pos.sm ; !sumim mod 2 ; !sumic mod 3];;

(*donne la paritÈ d'une position, 1 si elle est paire, -1 si elle est
impaire. Renvoie 0 si la position n'a pas de paritÈ,
i.e. sign(sm)<>sign(sc) *)
let paritÈ pos = let m = signature pos.sc in if m <> signature pos.sm then
0 else m;;

(* 2 mouvements d'ordre 1260, le premier trouvÈ par Butler, le second est
un exemple des dizaines de mouvements d'ordre 1260 que j'ai trouvÈ par
informatique *)
let butler = compo_list [j;B2;W;r;W] and m1260 = compo_list [b;o;W;v;W;b];;

(*le deuxiËme ÈlÈment du centre du groupe, avec l'identitÈ*)
let damier = compo_list
[d;g;P;P;A;G;G;F;F;D;D;H;p;D;p;p;F;A;P;f;f;P;d;d;H;f;f;P];;

let in_1260 pos = (*dÈtermine si une position donnÈe 'pos' engendre un sous
groupe de G d'ordre 1260*)
if (compo pos 420)<>id&(compo pos 630)<>id&(compo pos (1260/5))<>id&(compo
pos (1260/7))<>id&
(compo pos 1260)=id
then true else false;;

(*dÈtermine si une position donnÈe appartient ‡ ALGr de la face jaune. On
ne s'intÈresse pas aux positions triviales.*)
let in_ALGr (pos:rubik) =
((sub_vect pos.sm 0 8)=[|1;2;3;4;5;6;7;8|])&((sub_vect pos.sc 0 4)=[|
1;2;3;4|])
& ((sub_vect pos.irc 0 4)=[|0;0;0;0|])&((sub_vect pos.irm 0 8)=[|
0;0;0;0;0;0;0;0|])
& pos<>id & pos<>j & pos<>J & pos<>J2;;
        (*** 3) Compression des positions *)

type zip == int vect;;
(*une position compressÈe est un tableau de 4 entiers, chacun reprÈsentant
la compression de sc, sm, irc, irm*)
```

```
(*dÈclarations gÈnÈrales de variables et de fonctions auxiliaires pour ne
pas dÈfinir dans chaque fonction des variables temporaires, d'oˇ un gain
de temps puisqu'il s'agit de fonctions que l'on appellera des milliers de
fois par seconde...*)
(*fonction factorielle*)
let rec fac = fun
0->1
|n-> n*(fac (n-1));;

(*fonction puissance*)
let rec pow a = fun
0 -> 1
|n -> a*(pow a (n-1)) ;;

let var1 = ref 0
and var2 = ref 0
and var3 = ref 0
and variable = ref 0
and auxiliaire = ref [|1;2;3;4;5;6;7;8|]
and compteur = ref 0;;


(* 'rang' calcule le rang dans l'ordre lexicographique d'une permutation de
Sn, c‡d produit une bijection de Sn dans
[|0;n!-1|] ; on l'utilise pour n = 8 ou 12. Similitude avec Hˆrner pour la
mise en oeuvre et avec l'ordre lexicographique pour la mÈthode de
transformation*)
let rang (p : permutation) =
let n = vect_length p in
var1:=0;
for i = 0 to (n-2) do
var1:=!var1*(n-i);
for j = (i+1) to (n-1) do
if p.(j)<p.(i) then incr var1
done;
done; !var1;;

(*une remarque expÈrimentale: une permutation de S8 ou S12 est paire ssi
son rang est congru ‡ 0 ou 3 modulo 4*)

(*antirang(n) est la bijection rÈciproque de rang pour Sn*)
let antirang n r =
let rÈponse = make_vect n 0 in
rÈponse.(n-1) <- 1;
var1:= r;
for i = (n-2) downto 0 do
rÈponse.(i) <- 1+ (!var1 mod (n-i));
var1:= !var1 / (n-i);
for j = i+1 to (n-1) do
if rÈponse.(j)>= rÈponse.(i) then rÈponse.(j) <- rÈponse.(j) + 1
done;
done;
rÈponse;;

(*supertab est un tableau contenant la correspondance entre une permutation
de S8 et son rang ; il est plus rapide de calculer une fois pour toute ce
tableau plutÙt que d'utiliser la fonction inverse8 ‡ chaque fois ; pour des
raisons de mÈmoire, cela n'est pas possible avec S12 pour les permutations
des milieux*)

let (supertab : permutation vect) = make_vect 40320 [||];;
for i =0 to 40319 do
supertab.(i) <- (antirang(8) i)
done;;
(*on dÈfinit ici une fois pour toutes un tableau contenant les puissances
de 2 utiles ‡ la conversion d'entiers en base 2*)
let base2 =[|2048;1024;512;256;128;64;32;16;8;4;2;1|];;

(* 'enbase2' convertit un tableau IRM d'entiers (0 et 1) en un entier dont
l'Ècriture en base 2 est la suite de 0 et 1 du tableau *)
let enbase2 (tab : indices)=
var2:=0;
for i = 0 to 11  do
var2:=!var2+(tab.(i)*(base2.(i)));
```

```
done;
!var2;;

(*on dÈfinit ici une fois pour toutes un tableau contenant les puissances
de 3 utiles ‡ la conversion d'entiers en base 3*)
let base3=[|pow 3 7;pow 3 6;pow 3 5;pow 3 4;pow 3 3;9;3;1|];;

(* 'enbase3' convertit un tableau IRC d'entiers (0, 1 et -1 que l'on
transforme en 2) de longueur 8 en un entier dont l'Ècriture en base 3 est
cette suite d'entiers *)
let enbase3 (tab : permutation)=
var2:=0;
for i = 0 to 7  do
let k = tab.(i) in
if k<0 then
var2:=!var2+(2*base3.(i))
else
var2:=!var2+(k*(base3.(i)));
done;
!var2;;

(* ce tableau contiendra le rÈsultat de la fonction 'debase2' *)
let (debaseur2 : permutation) = make_vect 12 0;;

(* debase2 est la fonction rÈciproque de la fonction 'enbase2', il fournit
le tableau IRM de longueur 12 d'entiers (0 et 1) correspondant ‡ l'entier
qu'il reÁoit en paramËtre*)
let debase2 n =
var2:=n;
for i=0 to 11 do
let k = (!var2)/(base2.(i)) in
debaseur2.(i) <- k;
var2:=!var2 - (base2.(i)*k)
done;
(debaseur2 : permutation);;

(* ce tableau contiendra le rÈsultat de la fonction 'debase3' *)
let (debaseur3 : permutation) = make_vect 8 0;;

(* debase3 est la fonction rÈciproque de la fonction 'enbase3', il fournit
le tableau IRC de longueur 8 d'entiers 0, 1 ou -1 correspondant ‡ l'entier
qu'il reÁoit en paramËtre*)
let debase3 n =
var2:=n;
for i=0 to 7 do
let k = (!var2)/(base3.(i)) in
if k = 2 then
debaseur3.(i) <- -1
else
debaseur3.(i) <- k;
var2:=!var2 - (base3.(i)*k)
done;
(debaseur3 : permutation);;

(*cette fonction contracte une position du Rubik's Cube, c‡d la transforme
en une liste de 4 entiers qui est moins gourmande en mÈmoire*)
let contracte p= ([|rang p.sc;rang p.sm;enbase3 p.irc;enbase2 p.irm|] :
zip);;
let detracte (pos : zip) = (*cette fonction dÈtracte une position du
Rubik's Cube, c‡d transforme la liste de 4 entiers en une position que l'on
peut composer facilement*)
{sc = supertab.(pos.(0)); sm = antirang(12) pos.(1) ; irc = debase3 pos.
(2); irm = debase2 pos.(3)};;

(*ce programme tire au sort une position accessible du Rubik's Cube*)
let rand_pos () = let k = ref (detracte [|0;0;0;1|]) in
while trace !k <> [0;0;0] do
k:=detracte [|rand__int 40320;rand__int (fac 12);rand__int (pow 3
8);rand__int (pow 2 12)|]
done; !k ;;

(*ce programme tire au sort une position QUELCONQUE du Rubik's Cube*)
let rand_pos2 () = detracte [|rand__int 40320;rand__int (fac 12);rand__int
(pow 3 8);rand__int (pow 2 12)|];;
```

```
        (*** 4) Positions isomorphes *)

(*bloc de fonctions pour inverser rapidement une position*)
let inverse_perm (perm : permutation) = let n = vect_length perm in let inv
= make_vect n 0 in
for i = 0 to (n-1) do
inv.(perm.(i) - 1) <- i + 1
done;(inv : permutation);;

let inverse_indices8 (indices : indices) (perm : permutation) = let res =
make_vect 8 0 in for i = 0 to 7
do
res.(perm.(i)-1) <- - indices.(i)
done; res;;

let inverse_indices12 (indices : indices) (perm : permutation) = let res =
make_vect 12 0 in for i = 0 to 11
do
res.(perm.(i)-1) <- indices.(i)
done; res;;

let inverse pos = {sc = inverse_perm pos.sc; sm = inverse_perm pos.sm; irc
= inverse_indices8 pos.irc pos.sc;
irm = inverse_indices12 pos.irm pos.sm};;

(* 'spotH', 'spotG' et 'spotF' sont 3 positions inaccessibles du Rubik's
Cube que l'on utilise avec le principe de conjugaison pour trouver les
diffÈrentes orientations du cube*)
let spotH =  { sc = [|2; 3; 4;1;  6; 7; 8;5|]; sm = [| 2; 3; 4;1; 6; 7;
8;5; 10; 11; 12;9|]; irc = [|0; 0; 0; 0; 0; 0; 0; 0|];
irm = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]};;
let spotG =  { sc = [|4;3;7;8;1;2;6;5|]; sm = [|
8;4;7;12;1;3;11;9;5;2;6;10|]; irc = [|1; -1; 1; -1; -1; 1; -1; 1|];
irm = [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0;1|]};;
let spotF =  {sc = [|5;1;4;8;6;2;3;7|]; sm = [|
9;5;1;8;10;2;4;12;11;6;3;7|]; irc = [|-1;1; -1; 1 ; 1; -1; 1; -1|];
irm = [|1; 0; 1; 1; 0; 1; 0; 1; 1; 1;1; 0|]};;

(*on dÈfinit ici les 24 orientations spatiales possibles du Rubik's Cube*)
let orientations = make_vect 24 id;;
let k = ref spotH and l = ref spotG in
for i = 0 to 3 do
for j = 0 to 3 do
orientations.(4*i+j) <-  !l x !k ;
l:= !l x spotG done;
k:= !k x spotH done;;
let k = ref spotF and l = ref spotG in
for i = 0 to 1 do
for j = 0 to 3 do
orientations.(4*i+j+16) <- !l x !k ;
l:= !l x spotG
done;
k:= !k x spotF x spotF
done;;

(*on prÈcalcule les orientations inverses pour plus de rapiditÈ par la
suite dans les calculs*)
let orientations_inv = map_vect inverse orientations;;

(*cette fonction 'symÈtrie' renvoie le symÈtrique d'une position par
rapport au plan (spotG) *)
let sym = { sc = [|2; 1; 4; 3; 6; 5; 8; 7|]; sm = [|3; 2; 1; 4; 6; 5; 8; 7;
11; 10; 9; 12|]; irc = [|0; 0; 0; 0; 0; 0; 0; 0|];
irm = [|0; 0; 0; 0; 1; 1; 1; 1; 0; 0; 0; 0|]};;
let sym_inv = inverse sym;;
let symÈtrie p = let k = sym x p x sym_inv in { sc = k.sc ; sm = k.sm ; irc
= map_vect (function y -> - y) k.irc ;
irm = k.irm };;

(* [listomorph pos]  renvoie la liste des positions isomorphes 2 ‡ 2
distinctes de 'pos' *)
let listomorph p = let psym = symÈtrie p in let liste = ref [] in
```

```
for i = 0 to 23 do
let k = orientations_inv.(i) x p x orientations.(i) and ks =
orientations_inv.(i) x psym x orientations.(i) in
liste:= it_list union [] [!liste;[k];[ks];[inverse k];[inverse ks]]
done; !liste;;

(* [morph pos] renvoie le nombre de positions isomorphes 2 ‡ 2 distinctes
d'une position 'pos' ;
le morph d'une position est compris entre 1 et 96 *)
let morph p = list_length (listomorph p);;

(* cette fonction dÈtermine si 2 positions sont isomorphes *)
let is_omorph p1 p2 = mem p1 (listomorph p2);;

(* cette fonction dÈtermine le plus petit ÈlÈment d'une liste dans l'ordre
total lexicographique. Elle est polymorphe *)
let min_lexico (a::l) = it_list min a l;;

(* 'pos_ref' renvoie la position de rÈfÈrence associÈe ‡ un reprÈsentant
d'une famille d'isomorphe, et le nombre d'ÈlÈments de la famille
d'isomorphes*)
let pos_ref (p:rubik) = let k = map contracte (listomorph p) in
min_lexico k,list_length k;;

        (*** 5) Fonctions de recherche de positions *)

(* force_brute prend en argument un programme 'condition'  de type (rubik -
> bool) sÈlectionnant certaines positions particuliËres (par exemple
in_1260 ou in_ugroup), et dÈtermine toutes les positions du cube engendrÈes
par au plus 'profondeur' mouvements ÈlÈmentaires (le 1er mouvement
ÈlÈmentaire effectuÈ sur 'position' est diffÈrent de
'interdit') sur 'position' vÈrifiant 'condition'  *)

let force_brute (condition: rubik -> bool) (position:rubik)
(profondeur:int) (interdit:rubik) =
let affichage = [|"W";"J";"B";"V";"O";"R";"w";"j";"b";"v";"o";"r"|] in (*on
dÈfinit un tableau de rÈfÈrence utile par la suite*)
let rec test condition profondeur interdit pos historique= (*on dÈfinit le
programme rÈcursif qui va appliquer le principe de force brute*)
if condition pos then (*si une position vÈrifie la condition, alors on
affiche la sÈrie de mouvements correspondants*)
begin
do_list print_string historique;print_newline()
end;
match profondeur with
0 -> () (*on s'arrÍte lorsque la profondeur est nulle, c'est logique*)
|n -> if n>=4 then (*pour une profondeur assez ÈlevÈe, on affiche
l'avancement du programme pour avoir un contrÙle visuel de la progression
des recherches*)
begin
if interdit <>id then (*si le mouvement interdit est l'identitÈ, cela
signifie forcemment que c'est le premier passage du programme car il n'a
encore effectuÈ,
aucun mouvement sur 'position' ; on n'affichera donc rien ‡ ce passage-l‡*)
begin
do_list print_string (list_of_vect((make_vect (n-3) "-" )));
print_int (index (compo interdit 3) (list_of_vect tab));
print_newline () end end;
let i_interdit = (index interdit (id::(list_of_vect tab)) - 1 ) in (*on
dÈtermine l'indice du mouvement interdit dans le tableau 'tab', il vaut -1
pour l'identitÈ qui signifie que l'on interdit aucun mouvement particulier,
lors du premier passage du programme*)
for i = 0 to 11 do (*et l'on teste toutes les combinaisons des autres
mouvements sur 'pos'*)
if i<>i_interdit then
begin
let new_pos = pos x (tab.(i)) and mouv_interdit = compo (tab.(i)) 3 in
test condition (n-1) mouv_interdit new_pos (historique@[affichage.(i)])
(*passage rÈcursif ‡ une profondeur infÈrieure pour chaque nouvelle
position, avec mÈmorisation du mouvement effectuÈ, pour l'affichage, dans
'historique', et gÈnÈration du nouveau mouvement interdit qui est l'inverse
du dernier mouvement effectuÈ*)
end
```

```
done
in test condition profondeur interdit position [];
print_string "Le programme a terminÈ les recherches.";;

(* fonctions pour le calcul de Pos(n) *)

(*donnÈes contient les rÈsultats des programmes de recherche de Pos(n).
[donnÈes.(n).(i)] renvoie la liste des positions compressÈes de profondeur
n, dont le rang de la permutation des coins vaut i. On se limite ‡ n<=10*)
let (donnee:zip list vect vect) = [|[||];[||];[||];[||];[||];[||];[||];
[||];[||];[||];[||]|];;

(* on remplit 'donnee' pour n = 0 et n = 1, et l'on l'initialise pour les
autres valeurs de n*)
for i = 0 to 10 do
donnee.(i) <- make_vect 40320 [];
done;;
donnee.(0).(0) <- [contracte id];;
for i = 0 to 11 do
donnee.(1).(rang  tab.(i).sc) <- [contracte tab.(i)]
done;;

(*cette fonction compose un mouvement contractÈ avec un mouvement dÈtractÈ,
ce qui est utile pour la fonction 'nb_positions' qui suit.*)
let comp_tract p1 p2 =
contracte (detracte p1 x p2);;

(*version primaire de Pos(n)*)
let nb_positions n =
let tabn = donnee.(n-1) and verif = donnee.(n-2) in
for i = 0 to 40319 do
(*print_int (1000*i/40319);
print_newline();*)
let tabi = vect_of_list (tabn.(i)) in
for j = 0 to (vect_length tabi  - 1 ) do
for k = 0 to 11 do
let pos = comp_tract tabi.(j) tab.(k) in
let sg = pos.(0) in
if not (mem pos (verif.(sg) ) ) then
if not (mem pos ( (donnee.(n)).(sg) )) then
donnee.(n).(sg)<-(pos::(donnee).(n).(sg));
done;
done;
done;
print_string "Le programme a terminÈ les recherches.";
let data_n = donnee.(n) in
compteur:= 0 ; for i = 0 to 40319 do
compteur:=!compteur+(list_length data_n.(i) )
done;!compteur;;
(*on lance le calcul pour n=2 mouvements et l'on vÈrifie que l'on retrouve
bien le rÈsultat thÈorique: 12x11-18 = 114*)
nb_positions 2;;

(*[recherche n] cherche les positions de profondeur n ‡ partir des
positions de profondeur n-1, sans mÈmoriser les nouvelles positions, d'oˇ
un gain de mÈmoire considÈrable ; 'recherche' ne compare pas toutes les
positions entre elles*)
let recherche n =
compteur:=0;
let tabn = donnee.(n-1) and tab2 = map_vect (function y -> (inverse2
y).sc ) tab and verif = donnee.(n-2) in
for jj = 0 to 40319 do
(*pour chaque permutations des coins*)
if jj mod 40 = 0 then begin
print_int (1000*jj/40319);
print_newline() end;
let temp = ref [] in
for k = 0 to 11 do
let temp2 = ref [] in
(*pour chaque mvt ÈlÈmentaire*)
let i = rang (rond tab2.(k) supertab.(jj)) in
(*on dÈtermine l'unique permutation des coins qui donne celle recherchÈe
qui donne celle recherchÈe*)
let tabi = vect_of_list (tabn.(i)) in
```

```
for j = 0 to (vect_length tabi  - 1 ) do
let pos = comp_tract tabi.(j) tab.(k) in
if not (mem pos (verif.(jj) ) ) then
temp2:=pos::(!temp2)
done;
temp:=union !temp2 !temp
done;
compteur:= !compteur + list_length (!temp)
done;
print_string "Le programme a terminÈ les recherches.";
!compteur;;

(*on va utiliser certaines propriÈtÈs des listes triÈes pour rechercher
rapidement des ÈlÈments et fusionner les listes*)
#open "sort";;
let cp z y = if z = y then invalid_arg "cp" else z>=y;;

let mem22 a l = try merge cp [a] l; false
with _ -> true;;

(*version optimisÈe de recherche des positions de profondeur n*)
let Pos n = donnee.(n) <- make_vect 40320 [];compteur:=0;
let rajoute pos sg m =
try donnee.(n).(sg) <- merge cp [pos] donnee.(n).(sg);m
with _ -> 0
in
let tabn = donnee.(n-1) and verif = donnee.(n-2) in
let ii = ref 1 in
for i = 40319 downto 0 do
ii:=((!ii+1333) mod 40320);
if i mod 40 = 0 then begin
print_int (1000*(40319 - i)/40319);
print_newline()
end;
let tabi = vect_of_list (tabn.(!ii)) in
for j = 0 to (vect_length tabi  - 1 ) do
for k = 0 to 11 do
let machin = (detracte tabi.(j)) in
let pos,m = pos_ref ( machin x tab.(k)) in
let sg = pos.(0) in
(*mem22 est mem pour des listes triÈes*)
if not (mem22 pos (verif.(sg) ) ) then
compteur:= !compteur + (rajoute pos sg m);
let pos,m = pos_ref ( tab.(k) x machin) in
let sg = pos.(0) in
if not (mem22 pos (verif.(sg) ) ) then
compteur:= !compteur + (rajoute pos sg m);
done;
done;
done;
print_string "Le programme a terminÈ les recherches : = ";!compteur;;

(*la fonction, pour sauvegarder les rÈsultats sur le disque dur, dÈcoupe
les tableaux car Caml ne gËre pas l'Ècriture de donnÈes trop volumineuses*)
let sauvegarder m =
for n = 1 to m do
for i = 0 to 402 do
let k = open_out_bin ("donnee" ^ (string_of_int n) ^ "_" ^ (string_of_int
i)  ^ ".dat") in
let l = make_vect 100 [] in
for j = 0 to 99 do
l.(j) <- donnee.(n).(i*100+j)
done;
output_value k l;
close_out k
done;
let k = open_out_bin ("donnee" ^ (string_of_int n) ^ "_403.dat") in
let l = make_vect 20 [] in
for j = 40300 to 40319 do
l.(j-40300) <- donnee.(n).(j)
done;
output_value k l;
close_out k; done;;
```

```
let rÈcupÈrer m =
for n = 1 to m do
for i = 0 to 402 do
let k = open_in_bin ("donnee" ^ (string_of_int n) ^ "_" ^ (string_of_int i)
^ ".dat") in
let l = input_value k in
for j = 0 to 99 do
donnee.(n).(i*100+j) <- l.(j)
done;
close_in k done;
let k = open_in_bin ("donnee" ^ (string_of_int n) ^ "_403.dat") in
let l = input_value k in
for j = 0 to 19 do
donnee.(n).(40300+j) <- l.(j)
done;
close_in k; done;;


        (*** 6) Mouvements isomorphes *)
(*on utilisera l'isomorphisme sur les mouvements pour la rÈsolution du
Rubik's Cube*)

type mouvements == rubik list;; (*suite de mouvements ÈlÈmentaires*)
type dÈcomposition == int list;; (*dÈcomposition en mouvement
ÈlÈmentaires : suite de chiffres indiquant l'indice dans 'tabu' du
mouvement ÈlÈmentaire effectuÈ*)

let tabu = [|F;P;A;H;D; G; f; p;a;h; d;  g|] and tabu2 = [|f;p;a;h;g; d; F;
P;A;H; G;  D|] and
tabu_string = [|"W-";"J-";"B-";"V-";"O-";
"R-";"w-";"j-";"b-";"v-";"o-";"r-"|];;

(*d'un mouvement reprÈsentÈ sous la forme d'une liste de mouvements
ÈlÈmentaires, renvoie la liste de leur numÈro par rapport ‡ tabu, c'est ‡
dire la dÈcomposition du mouvement. Utile pour ne pas avoir ‡ mÈmoriser le
nombre associÈ ‡ chaque mvt, il suffit de faire dÈcompose [P;H;G;G;D]*)
let rec (dÈcompose: mouvements -> dÈcomposition ) = function
[] -> []
|(s::suite) -> (index s (list_of_vect tabu))::(dÈcompose suite);;

(*'effectue' donne la position correspondant ‡ une sÈrie de mouvements*)
let rec (effectue : dÈcomposition -> rubik) = function
[] -> id
|[s] -> tabu.(s)
|(s::suite) -> tabu.(s) x (effectue suite);;

let rond2 (sc1 : permutation) (sc2 : permutation) = let sc = make_vect 12 0
in
for i=0 to 11 do
sc.(i) <- sc1.(sc2.(i))
done;
(sc : permutation);;

(*on dÈtermine les 24 orientations possibles du cube dans l'espace et les
permutations des noms de face correspondant*)
let (liste_orientations : permutation list) = let ox =[| 3;0;1; 2;4;  5;
9;6;7;8 ;10;11|] (*permutations des faces lors d'une rotations vers soi,
suivant ox*) and oz = [|4;1;5;3;2;0;10;7;11;9;8;6|] and oy = [|0;5;2;4  ;1;
3; 6; 11;8;10; 7; 9|] and
liste = ref [] in let perm0x = ref ox and perm0z= ref oz and perm0y= ref oy
in
for i = 1 to 4 do
for j = 1 to 4 do
liste:=(rond2 !perm0z !perm0x)::(!liste);
perm0x:=rond2 !perm0x ox
done;
perm0z:=rond2 !perm0z oz
done;
for i = 1 to 2 do
for j = 1 to 4 do
liste:=(rond2 !perm0y !perm0x)::(!liste);
perm0x:=rond2 !perm0x ox
done;
```

```
permOy:=rond2 oy (rond2 oy oy)
done; !liste ;;

(* c'est la matrice des permutations correspondant ‡ effectuer un mouvement
de maniËre symÈtrique par rapport ‡ la couronne centrale entre les faces F
et A *)
let (listp : permutation) = vect_of_list (dÈcompose (list_of_vect tabu2));;

(*sÈries de 4 fonctions effectuant le symÈtrique, ou l'inverse... d'une
sÈrie de mouvements*)
let rec mouv  perm = map (function i -> perm.(i)) ;;

let rec mouv_sym  perm = map (function i -> listp.(perm.(i))) ;;

let rec mouv_inv perm = function
[] -> []
|(s::suite) -> (mouv_inv perm suite) @ [(6+perm.(s)) mod 12];;

let rec mouv_sym_inv perm = function
[] -> []
|(s::suite) -> (mouv_sym_inv perm suite) @ [listp.((6+perm.(s)) mod 12)];;

(*donne les dÈcompositions distinctes correspondant au rÈsultat de
listomorph ; certaines dÈcompositions diffÈrentes peuvent donner le mÍme
mouvement. Elles sont simplement symÈtriques ou 'rotatives' l'une par
rapport ‡ l'autre, ou encore inverses*)
let listomorph_mouv (p : dÈcomposition) = let rec aux = function
[] -> []
|(perm::suite) -> it_list union []
[[mouv perm p];[mouv_inv perm p];[mouv_sym perm p];[mouv_sym_inv perm p];
(aux suite)]
in aux liste_orientations;;




        (*** 7) RÈsolution du Rubik's Cube *)

(*mouvements effectuant des 3-cycles sur les coins*)
let c31 = dÈcompose [g;P;D;p;G;P;d;p];;
let c32 = dÈcompose [o;b;O;g;P;D;p;G;P;d;p;o;B;O];;
let c33 = dÈcompose [W;g;P;D;p;G;P;d;p;w];;
(*liste de tous les mouvements effectuant des 3-cycles sur les coins*)
let liste_mouvements =
vect_of_list (it_list union [] [listomorph_mouv c31;listomorph_mouv
c32;listomorph_mouv c33]);;

(*recherche du mouvement effectuant un certain 3-cycle des coins sur une
certaine position*)
let coins_3cycle cycle position   =
let k = rond (c3_2perm8 cycle) position.sc and c = ref (-1) in
for i = 0 to 287 do
if (position x (effectue liste_mouvements.(i))).sc = k then c:=i
done;
liste_mouvements.(!c),(position x (effectue liste_mouvements.(!c)));;

(*mouvements effectuant des 3-cycles sur les milieux*)
let m31 = dÈcompose [g;a;p;A;P;a;p;A;P;G];;
let m32 = dÈcompose [b;g;a;p;A;P;a;p;A;P;G;B];;
let m33 = dÈcompose [b;b;g;a;p;A;P;a;p;A;P;G;b;b];;
let m34 = dÈcompose [v;b;b;g;a;p;A;P;a;p;A;P;G;b;b;V];;
let m35 = dÈcompose [j;b;J;g;a;p;A;P;a;p;A;P;G;j;B;J];;
let m36 = dÈcompose [V;o;o;g;a;p;A;P;a;p;A;P;G;o;o;v];;
let m37 = dÈcompose [R;g;a;p;A;P;a;p;A;P;G;r];;
let m38 = dÈcompose [r;g;a;p;A;P;a;p;A;P;G;R];;
let m39 = dÈcompose [R;v;v;b;b;g;a;p;A;P;a;p;A;P;G;b;b;v;v;r];;
(*liste de tous les mouvements effectuant des 3-cycles sur les milieux*)
let liste_mouvements2 = vect_of_list (it_list union []
[listomorph_mouv m31;listomorph_mouv m32;listomorph_mouv
m33;listomorph_mouv m34;listomorph_mouv m35;
listomorph_mouv m36;listomorph_mouv m37;listomorph_mouv m38;listomorph_mouv
m39]);;
```

```
(*recherche du mouvement effectuant un certain 3-cycle des milieux sur une
certaine position*)
let milieux_3cycle cycle position   = let k = rond (c3_2perm12 cycle)
position.sm and c = ref (-1) in
for i = 0 to 863 do
if (position x (effectue liste_mouvements2.(i))).sm = k then c:=i
done; liste_mouvements2.(!c),(position x (effectue liste_mouvements2.(!
c)));;

(*effectue une liste de 3-cycles des coins sur une position et renvoie la
dÈcomposition nÈcessaire ‡ les effectuer et la position finale*)
let rec coins_perm position = function
[] -> [],position
|(xx::l) -> let (j, jj)= coins_3cycle xx position in let s,ss=(coins_perm
jj l) in
j@s,ss ;;

(*effectue une liste de 3-cycles des milieux sur une position et renvoie la
dÈcomposition nÈcessaire ‡ les effectuer et la position finale*)
let rec milieux_perm position = function
[] -> [],position
|(xx::l) -> let (j, jj)= milieux_3cycle xx position in let
s,ss=(milieux_perm jj l) in
j@s,ss ;;

(*rÈsout une position paire pour les permutations*)
let perm_pos pos = let xx,y = (milieux_perm pos  ( en_3cycles (inverse_perm
pos.sm) ) )
in let a,b= (coins_perm y ( en_3cycles (inverse_perm pos.sc)  ) ) in
xx@a,b;;

(*rÈsout une position quelconque pour les permutations*)
let rÈsoudre_perm position = if paritÈ position = 1 then perm_pos position
else
let a,b=(perm_pos (position x W)) in (0::a),b;;
(*liste des mouvements faisant tourner 2 coins adjacents et rien d'autre*)
let tourne_coins = vect_of_list (listomorph_mouv (dÈcompose
[g;a;G;f;g;A;G;F;g;a;D;A;G;a;d;A]));;
(*liste des mouvements faisant tourner 2 milieux adjacents et rien
d'autre*)
let tourne_milieux = vect_of_list (listomorph_mouv (dÈcompose
[G;d;f;g;D;P;P;G;d;f;D;P;g;P;G;P;P;g;a;p;A;p;a;p;p;A]));;

let succ = [|1;2;3;7;5;6;7;-1|];; (*dÈtermine les couples de coins que l'on
fait tourner : 0-1 puis 1-2 puis 2-3 puis 3-7 puis 4-5... puis 6 - 7, le
dernier ne sert ‡ rien*)
let succ2 = [|1;2;3;7;8;9;10;8;9;10;11;-1|];; (*mÍme principe, mais pour
les milieux*)

(*rÈsout une position dont les permutations sont ‡ l'identitÈ mais dont les
indices de rotations ne sont pas tous nuls ; renvoie la dÈcomposition y
parvenant*)
let analyse_tourneur position= let p = ref position and l = ref [] and
compteur = ref true in for i = 0 to 6 do
compteur:=true;let m = !p.irc.(i) and c = ref (-1) in
if m<> 0 then
begin for j = 0 to 95 do
if !compteur then
let k = !p x (effectue tourne_coins.(j)) in if k.irc.(i)= 0 & k.irc.(succ.
(i))<> !p.irc.(succ.(i)) then begin c:=j;p:=k;compteur:=false end
done;
if !c>=0 then l:=tourne_coins.(!c)@(!l)
end
done;
for i = 0 to 10 do
compteur:=true;
let m = !p.irm.(i) and c = ref (-1) in
if m<> 0 then
begin for j = 0 to 95 do
if !compteur then let k = !p x (effectue tourne_milieux.(j)) in if k.irm.
(i)= 0 & k.irm.(succ2.(i))<> !p.irm.(succ2.(i)) then begin
c:=j;p:=k;compteur:=false end
done;if !c>=0 then l:=tourne_milieux.(!c)@(!l)
end
```

```
done;!l;;

(*rÈsout une position et renvoie le nombre de coups nÈcessaires et la
dÈcomposition permettant la rÈsolution*)
let rÈsoudre pos= let a,b= rÈsoudre_perm pos in let k = a@(analyse_tourneur
b) in list_length k,k;;

(*convertit une dÈcomposition en une chaine de caractËres affichant les
mouvements*)
let rec en_string = function
[] -> ""
|[l] -> tabu_string.(l)^"
"
|(l::suite) -> tabu_string.(l)^(en_string suite);;

(*affiche explicitement la solution d'une position*)
let solution position = let k,kk = rÈsoudre position in
print_string ("une solution en: "^(string_of_int k)^"
mouvement(s)."^(en_string kk));;



        (*** 8) Affichage du cube en 2D *)

#open "graphics";;
let orange = 0xFF8000;;
let gris = 0x909090;;

type coin == color*color*color;;
type milieu == color*color;;
type point3D == int * int * int;;
type point == int * int;;
type face == int;;

let rotation (xx,y,z) i= if i = 0 then xx,y,z else if i = 1 then z,xx,y
else y,z,xx;;
let rotation2 (xx,y) i= if i = 0 then xx,y else y,xx;;
(*chaque coin est un triplet de couleurs, dans le sens trigo, le haut ou
bas est la premiere couleur*)
let (couleur_coins:coin vect) =
[|(green,red,white);(green,white,orange);(green,orange,blue);
(green,blue,red);(yellow,white,red);(yellow,orange,white);
(yellow,blue,orange);(yellow,red,blue)|];;

let (couleur_milieux:milieu vect) = [|(green,red);(green,white);
(green,orange);(green,blue);(red,white);(white,orange)
;(orange,blue);(blue,red);(yellow,red);(yellow,white);(yellow,orange);
(yellow,blue)|];;

let (plan_coins : (point3D*point3D*point3D) vect) =
[|(1,1,0),(2,1,1),(3,0,1)
;
(1,1,1),(3,1,1),(6,1,0)
;
(1,0,1),(6,0,0),(4,0,0)
;
(1,0,0),(4,0,1),(2,0,1)
;
(5,1,1),(3,0,0),(2,1,0)
;
(5,1,0),(6,1,1),(3,1,0)
;
(5,0,0),(4,1,0),(6,0,1)
;
(5,0,1),(2,0,0),(4,1,1)|];;

let (plan_milieux : (point3D*point3D) vect) = [|
(1,1,0),(2,1,2);
(1,2,1),(3,1,2);
(1,1,2),(6,1,0);
(1,0,1),(4,0,1);
(2,2,1),(3,0,1);
(3,2,1),(6,2,1);
(6,0,1),(4,1,0);
```

```
(4,1,2),(2,0,1);
(5,1,2),(2,1,0);
(5,2,1),(3,1,0);
(5,1,0),(6,1,2);
(5,0,1),(4,2,1)
|];;

let faces_par_coin = [|[1;2;3];[1;3;6];[1;4;6];[1;2;4];[2;3;5];[3;5;6];
[4;5;6];[2;4;5]|];;
let coins_par_face = [|[|4;1;2;3|];[|8;5;1;4|];[|5;6;2;1|];[|3;7;8;4|];[|
7;6;5;8|];[|3;2;6;7|]|];;
let milieux_par_face = [| [|1;2;3;4|];[|9;5;1;8|];[|10;6;2;5|];[|
7;12;8;4|];[|11;10;9;12|];[|3;6;11;7|] |];;
let couleur_face = [|green;red;white;blue;yellow;orange|];;

let pas = ref 26;;
let b1 = 750;; let b2 = 350;;
let moveto2 a b = moveto (a+b1) (b+b2);;
let lineto2 a b = lineto (a+b1) (b+b2);;
let fill_poly2 t = fill_poly (map_vect (function (a,b) -> (a+b1,b+b2) )
t);;
let fill_rect2 a b c d = fill_rect (a+b1) (b+b2) c d;;

let (emplacement_cases : point vect ) =
[|3*(!pas),9*(!pas);3*(!pas),6*(!pas);6*(!pas),6*(!pas);0,3*(!pas);3*(!
pas),3*(!pas);3*(!pas),0|];;

let draw_coins pc irc =
let drawc couleur (n,xx,y) =
set_color couleur; let (x0,y0) = emplacement_cases.(n-1) in
fill_poly2 [|x0+2*(!pas)*xx,y0+2*(!pas)*y;x0+2*(!pas)*xx+(!pas),y0+2*(!
pas)*y;
x0+(!pas)+2*(!pas)*xx,y0+(!pas)+2*(!pas)*y;x0+2*(!pas)*xx,y0+(!pas)+2*(!
pas)*y|]
in
for i = 0 to 7 do
let couleur1,c2,c3 = rotation couleur_coins.(i) irc.(i) and
destination1,d2,d3 = plan_coins.(pc.(i)-1) in
drawc couleur1 destination1;
drawc c2 d2;
drawc c3 d3
done ;;


let draw_milieux pm irm =
let drawm couleur (n,xx,y) =
set_color couleur; let (x0,y0) = emplacement_cases.(n-1) in
fill_poly2 [|x0+(!pas)*xx,y0+(!pas)*y;x0+(!pas)*xx+(!pas),y0+(!pas)*y;
x0+(!pas)+(!pas)*xx,y0+(!pas)+(!pas)*y;x0+(!pas)*xx,y0+(!pas)+(!pas)*y|]
in
for i = 0 to 11 do
let couleur1,c2 = rotation2 couleur_milieux.(i) irm.(i) and destination1,d2
= plan_milieux.(pm.(i)-1) in
drawm couleur1 destination1;
drawm c2 d2
done ;;

let dessine_structure () =
let couleurs =[|green;red;white;blue;yellow;orange|] in
for i = 0 to 5 do
set_color couleurs.(i);
let (x0,y0)=emplacement_cases.(i) in
fill_rect2 (x0+(!pas)) (y0+(!pas)) (!pas) (!pas)
done;
set_color black;
let ppas=3*(!pas) in
moveto2 (2*ppas) ppas;lineto2 0 ppas;lineto2 0 (2*ppas);lineto2 (3*ppas)
(2*ppas);lineto2 (3*ppas) (3*ppas);
lineto2 ppas (3*ppas);lineto2 ppas 0;lineto2 (2*ppas) 0;
lineto2 (2*ppas) (4*ppas);lineto2 ppas (4*ppas);lineto2 ppas
(3*ppas);moveto2 (4*(!pas)) 0;
lineto2 (4*(!pas)) (12*(!pas));moveto2 (5*(!pas)) 0;lineto2 (5*(!pas))
(12*(!pas));
moveto2 0 (4*(!pas));lineto2 (6*(!pas)) (4*(!pas));moveto2 0 (5*(!
```

```
pas));lineto2 (6*(!pas)) (5*(!pas));
moveto2 ppas (7*(!pas));lineto2 (3*ppas) (7*(!pas));
moveto2 ppas (8*(!pas));lineto2 (3*ppas) (8*(!pas));
moveto2 ppas (!pas);lineto2 (2*ppas) (!pas);moveto2 ppas (2*(!pas));lineto2
(2*ppas) (2*(!pas));
moveto2 ppas ((!pas)+3*ppas);lineto2 (2*ppas) ((!pas)+3*ppas);moveto2 ppas
(2*(!pas)+3*ppas);
lineto2 (2*ppas) (2*(!pas)+3*ppas);
moveto2 (!pas) ppas;lineto2 (!pas) (2*ppas);moveto2 (2*(!pas)) ppas;lineto2
(2*(!pas)) (2*ppas);
moveto2 ((!pas)+2*ppas) (2*ppas);lineto2 ((!pas)+2*ppas) (3*ppas);moveto2
(2*(!pas)+2*ppas) (2*ppas);
lineto2 (2*(!pas)+2*ppas) (3*ppas);;

let draw_pos pos =
draw_coins pos.sc pos.irc;
draw_milieux pos.sm pos.irm;dessine_structure();;

let to_pos liste_couleurs = (*convertit une liste de mouvements
ÈlÈmentaires en une position, chaque mouvement ÈlÈmentaire horaire dÈsigne
en fait une couleur, et la liste reprÈsente les couleurs des facettes sur
la face verte, puis rouge, blanche, bleue, jaune et orange*)
let couleurs = [|white;yellow;blue;green;orange;red|] and tab =
list_of_vect tab
in
let rec prog = function
[] -> []
|(y::l) -> couleurs.(index y tab - 6)::(prog l)
in
let vect_couleurs = vect_of_list (prog liste_couleurs) in
let pos = {sc = [|1; 2; 3; 4; 5; 6; 7; 8|];
  sm = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12|];
  irc = [|0; 0; 0; 0; 0; 0; 0; 0|];
  irm = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]} in  for i = 0 to 7 do
let a,b,c = plan_coins.(i) in
let n1,x1,y1 = a and n2,x2,y2 = b and n3,x3,y3 = c and j = ref true in
let k = vect_couleurs.(9*(n1-1)+2*x1+3*(2-2*y1)),vect_couleurs.
(9*(n2-1)+2*x2+3*(2-2*y2)),
vect_couleurs.(9*(n3-1)+2*x3+3*(2-2*y3)) and liste = list_of_vect
couleur_coins in
try
let ind = index k liste in
pos.sc.(ind) <- 1+i;pos.irc.(ind) <- 0;j:=false
with _ -> ();
try
let ind = index (rotation k 1) liste in
pos.sc.(ind) <- 1+i;pos.irc.(ind) <- (-1);j:=false
with _ -> ();
try
let ind = index (rotation k (-1)) liste in
pos.sc.(ind) <- 1+i;pos.irc.(ind) <- 1;j:=false
with _ -> ();
if !j then invalid_arg "cube non-valide"
done;
for i = 0 to 11 do
let a,b = plan_milieux.(i) in
let n1,x1,y1 = a and n2,x2,y2 = b and j = ref false in
let k = vect_couleurs.(9*(n1-1)+x1+3*(2-y1)),vect_couleurs.
(9*(n2-1)+x2+3*(2-y2)) and
liste = list_of_vect couleur_milieux in
try
let ind = index k liste in
pos.sm.(ind) <- 1+i;pos.irm.(ind) <- 0;j:=false
with _ -> ();try
let ind = index (rotation2 k 1) liste in
pos.sm.(ind) <- 1+i;pos.irm.(ind) <- 1;j:=false
with _ -> ();
if (!j) then invalid_arg "cube non-valide"
done;
pos;;



        (*** 9) Dessin en 3D *)
```

```
(*les variables 'pas' indiquent l'emplacement et la dimension du cube*)
let pas2 = 150;;
let pas22 = 150.;;
let pas3 = 400;;
let coord_coins = ref [|(pas2,pas2,pas2);(-pas2,pas2,pas2);(-pas2,-
pas2,pas2);(pas2,-pas2,pas2);(pas2,pas2,-pas2);(-pas2,pas2,-pas2);(-pas2,-
pas2,-pas2);(pas2,-pas2,-pas2)|];;

let coins_maxx () = let mc = map_vect (function y -> let k,a,v=y in k) !
coord_coins and l = ref [] and mx = ref 0 in
for i = 0 to 7 do
if mc.(i)> !mx then begin mx:=mc.(i);l:=[i] end else
if mc.(i) = !mx then l:=i::(!l)
done; !l ;;

let rec inter2 (a::l) = it_list intersect a l;;

let faces_visibles () = inter2 (map (function y -> faces_par_coin.(y))
(coins_maxx ()));;


let true_coord = ref [|(pas22,pas22,pas22);(-.pas22,pas22,pas22);
(-.pas22,-.pas22,pas22);(pas22,-.pas22,pas22);(pas22,pas22,-.pas22);
(-.pas22,pas22,-.pas22);(-.pas22,-.pas22,-.pas22);
(pas22,-.pas22,-.pas22)|];;

let pi = acos(-1.);;

let abs_f f = if f>=0. then f else (-. f);;
let signe f = if f>0. then 1 else if f<0. then -1 else 0;;
let arrondir f = let k = int_of_float f in if abs_f (float_of_int k -.
f)>0.5 then k + (signe f) else k;;
let constr_coord () = coord_coins := map_vect (fun (a,b,c) -> arrondir
a,arrondir b,arrondir c) !true_coord;;

(*n dÈsigne le nombre de degrÈs*)
let rotation_verticale n = let t = (float_of_int n)*.pi/.180. in
true_coord:=map_vect (fun (a,b,c) -> (a,b*.cos(t)-.c*.sin(t),c*.cos(t)
+.b*.sin(t))) !true_coord;constr_coord ();;
let rotation_horizontale n = let t = (float_of_int n)*.pi/.180. in
true_coord:=map_vect (fun (a,b,c) -> (a*.cos(t)-.b*.sin(t),b*.cos(t)
+.a*.sin(t),c)) !true_coord;constr_coord ();;
let rotation_transversale n = let t = (float_of_int n)*.pi/.180. in
true_coord:=map_vect (fun (a,b,c) -> (a*.cos(t)-.c*.sin(t),b,c*.cos(t)
+.a*.sin(t))) !true_coord;constr_coord ();;


(*opÈrations sur les points*)
let add (a,b) (c,d) = a+c,b+d;;
#infix "add";;
let sub (a,b) (c,d) = a-c,b-d;;
#infix "sub";;
let mul a (c,d) = a*c,a*d;;
#infix "mul";;
let div a (c,d) = c/a,d/a;;
#infix "div";;

let ligne (a,b) = lineto (a+pas3) (b+pas3);;
let point (a,b) = moveto (a+pas3) (b+pas3);;
let fill_poly22 v = fill_poly (map_vect (fun y -> y add (pas3,pas3)) v);;


let drawc couleur (face,xx,y) faces =
if mem face faces then begin
let k = coins_par_face.(face - 1) and coord_coins = map_vect (fun (a,b,c) -
> b,c ) !coord_coins in let c1 =
coord_coins.(k.(0)-1) and  c2 = coord_coins.(k.(1)-1) and  c3 =
coord_coins.(k.(2)-1)
and c4 = coord_coins.(k.(3)-1) and w = 2*xx and h = 2*y in let u = (c2 sub
c1) and v = (c4 sub c1)  in
set_color couleur;
fill_poly22 [|c1 add (3 div ((w mul u) add (h mul v)));c1 add (3 div (((w
+1) mul u) add (h mul v)));c1 add
```

```
(3 div (((w+1) mul u) add ((h+1) mul v)));c1 add (3 div ((w mul u) add ((h
+1) mul v)))|];
set_color couleur_face.(face-1);
fill_poly22 [|c1 add (3 div ((1 mul u) add (1 mul v)));c1 add (3 div (((2)
mul u) add (1 mul v)));
c1 add (3 div (((2) mul u) add ((2) mul v)));c1 add (3 div ((1 mul u) add
((2) mul v)))|];
set_color black;
point (c1 add (3 div ((w mul u) add (h mul v))));ligne (c1 add (3 div (((w
+1) mul u) add (h mul v))));
ligne (c1 add (3 div (((w+1) mul u) add ((h+1) mul v)))); ligne (c1 add (3
div ((w mul u) add ((h+1) mul v))));
ligne (c1 add (3 div ((w mul u) add (h mul v))));
end;;

let draw_coins3D pc irc faces=
for i = 0 to 7 do
let couleur1,c2,c3 = rotation couleur_coins.(i) irc.(i) and
destination1,d2,d3 = plan_coins.(pc.(i)-1) in
drawc couleur1 destination1 faces;
drawc c2 d2 faces;
drawc c3 d3 faces
done ;;

let drawm couleur (face,xx,y) faces =
if mem face faces then begin
let k = coins_par_face.(face - 1) and coord_coins = map_vect (fun (a,b,c) -
> b,c ) !coord_coins in
let c1 = coord_coins.(k.(0)-1) and  c2 = coord_coins.(k.(1)-1) and  c3 =
coord_coins.(k.(2)-1)
and c4 = coord_coins.(k.(3)-1) and w = xx and h = y in let u = (c2 sub c1)
and v = (c4 sub c1)  in
set_color couleur;
fill_poly22 [|c1 add (3 div ((w mul u) add (h mul v)));c1 add (3 div (((w
+1) mul u) add (h mul v)));
c1 add (3 div (((w+1) mul u) add ((h+1) mul v)));c1 add (3 div ((w mul u)
add ((h+1) mul v)))|];
set_color black;
point (c1 add (3 div ((w mul u) add (h mul v))));ligne (c1 add (3 div (((w
+1) mul u) add (h mul v))));
ligne (c1 add (3 div (((w+1) mul u) add ((h+1) mul v)))); ligne (c1 add (3
div ((w mul u) add ((h+1) mul v))));
ligne (c1 add (3 div ((w mul u) add (h mul v))));
end;;

let draw_milieux3D pm irm faces=
for i = 0 to 11 do
let couleur1,c2 = rotation2 couleur_milieux.(i) irm.(i) and destination1,d2
= plan_milieux.(pm.(i)-1) in
drawm couleur1 destination1 faces;
drawm c2 d2 faces;
done ;;

let draw3D pos = let f = faces_visibles() in
draw_coins3D pos.sc pos.irc f;
draw_milieux3D pos.sm pos.irm f;;

rotation_transversale 30;
rotation_horizontale 30;
rotation_verticale 30;;

(*utile pour l'enveloppe convexe des coins projetÈs en 2D parrallelement ‡
x*)
let min_x () = let mc = map_vect (fun (y,z,t) -> z) !coord_coins and l =
ref [] and mx = ref 0 in
for i = 0 to 7 do
if mc.(i)< !mx then begin mx:=mc.(i);l:=[i] end else
if mc.(i) = !mx then l:=i::(!l)
done; !l ;;

(*opÈration 'privÈ de' sur les ensembles*)
let rec privÈ2 ensemble (s::l) = if mem s ensemble then privÈ2 ensemble l
else s ;;
```

```
(*fait tourner une liste pour placer un de ses ÈlÈments en tÍte ; erreur si
la liste ne contient pas l'ÈlÈment*)
let rec placer k hist= function
(y::l) -> if y = k then (k::l)@hist else placer k (hist@[y]) l;;

let enveloppe () = (*dÈtermine l'enveloppe convexe des coins projetÈs en 2D
sur le plan de l'Ècran*)
let res = ref [] and m = faces_visibles() in if list_length m = 1 then
res:=list_of_vect (coins_par_face.(hd m - 1)) else
if list_length m = 2 then let f1 = list_of_vect (coins_par_face.(hd m -1))
and
f2 =  list_of_vect (coins_par_face.(hd (tl m) -1)) in
begin
let k = hd (intersect f1 f2) in
let s = placer k [] f1 in if mem (hd (tl s)) (intersect f1 f2) then
res:=(placer k [] f2)@[hd (tl (tl s));hd (tl (tl (tl s)))] else
let ss = placer k [] f2 in res:=s@[hd (tl (tl ss));hd (tl (tl (tl ss)))]
end
else (*il y a 3 faces visibles*)
begin
let f1 = list_of_vect (coins_par_face.(hd m -1)) and f2 =  list_of_vect
(coins_par_face.(hd (tl m) -1)) and f3 =  list_of_vect (coins_par_face.(hd
(tl (tl m) ) -1)) in
res:=[privÈ2 f3 (intersect f1 f2);privÈ2 (union f2 f3) f1;privÈ2 f2
(intersect f1 f3);privÈ2 (union f2 f1) f3;
privÈ2 f1 (intersect f3 f2);privÈ2 (union f1 f3) f2]
end;
(*placer l'enveloppe dans le bon sens maintenant*)
res:=(map (function y -> y-1) !res);
let k = hd (intersect !res (min_x())) in
placer k [] !res;;

(*pour Èviter les clignotements, on efface lors de la rotation 3D que la
partie qu'il est nÈcessaire d'effacer,
c-‡-d celle sur laquelle on ne va pas redessiner*)
let clear () = let t = (map (function a -> let (vv,c,d) = !coord_coins.(a)
in c+pas3,d+pas3 ) (enveloppe())) in
set_color black;fill_poly (vect_of_list
([115,100;700,100;700,700;115,700;115,100]@t@[hd t]));;

(*pos3D fait Èvoluer la position en 3 dimensions avec la souris*)
let pos3D p = open_graph "800x800+0+0" ; clear() ; draw3D p ; let k = ref
(wait_next_event [Poll]) in
let m = ref (!k.mouse_x,!k.mouse_y) in
while  button_down() = false & !k.key=`\000` do
let t = !k.mouse_x,!k.mouse_y in
if t <> !m then begin let a,b = t sub !m in m:=t; rotation_horizontale
a;rotation_transversale b;clear();draw3D p end;
k:=wait_next_event [Mouse_motion;Button_down;Key_pressed] done;;

        (*** 10) Mini logiciel pour le Rubik's Cube *)

let position_sauvÈe = ref id;;

let maxixi (a,b) = max a b;;

let draw3D2 pos = draw3D pos; draw_pos pos;;

let norme (a,b) = a*a + b*b;;

let (projette : point3D -> point) = fun (e,f,g) -> (f,g);;

let next_coin (xx,y) face = let point = (xx-pas3),(y - pas3) and k =
coins_par_face.(face-1) and mx = ref (-1) and
vx = ref 100000000 in
for i = 0 to 3 do
let p = norme (point sub (projette !coord_coins.(k.(i)-1))) in if p< !vx
then begin vx:=p;mx:=k.(i)
end done;
!mx;;

let help_string = "==Rubik's Cube 3D en CAML==
** Souris:
---------------
```

```
-> dÈplacement du curseur :
    - en mode {rotation 3D du cube}, fait pivoter le cube en 3D ;
    - en mode {rotation des faces}, ne fait rien si le bouton de la souris
n'est pas enfoncÈ
-> appui bref sur le bouton :
    - dans la partie en haut ‡ droite de l'Ècran, fait tourner les faces
dans le sens trigo/horaire (selon
      le mode trigo ou horaire, touche 'c') en appuyant sur les centres des
faces en 2D
    - en mode {rotation des faces},dans la zone noire, ne fait rien,
      dans la zone blanche ‡ gauche, passe en mode {rotation 3D du cube}.
    - en mode {rotation 3D du cube}, passe en mode {rotation des faces}.
-> appui prolongÈ sur le bouton :
    - en mode {rotation 3D du cube}, passe en mode {rotation des faces}.
    - en mode {rotation des faces}, fait tourner les faces en le combinant
avec le mouvement du curseur

** Touches:
-----------------
- ESPACE : bascule entre le mode de rotation 3D du cube et le mode de
rotation des faces en 3D
- ENTER : permet d'Èditer une position du Rubik's Cube. Pour l'Èditer, on
choisit la couleur en cliquant sur le
  centre des faces en 2D puis l'on remplit certaines cases avec cette
couleur. ENTER pour confirmer la position.
  Les cases dont la couleur n'a pas ÈtÈ dÈfinie sont grisÈes.
- W,R,B,J,O,V (ou w,r,b,j,o,v) pour faire tourner dans le sens trigo (ou
horaire) les faces blanche, rouge, bleue, jaune, orange, ou verte ...
- z : annule le dernier mouvement effectuÈ
- i : donne l'inverse de la position
- m : mÈlange le Rubik's Cube (position alÈatoire)
- S : sauve la position courante dans la variable !position_sauvÈe
- s : rÈsout la position actuelle ; ENTER pour effectuer un mouvement de
rÈsolution, q pour stopper la rÈsolution
- l : donne le nombre de positions isomorphes ‡ la position ('L' minuscule)
- p : donne la paritÈ de la position
- c : bascule entre mode trigonomÈtrique / horaire pour la rotation en 2D
des faces en haut ‡ droite
- a : indique si la position est accessible
- ESC : initialise le cube ‡ l'identitÈ
- q : quitte le programme Cube ; Q : quitte CAML";;
let help() = print_string help_string;;

let dessiner liste = clear_graph();moveto 0 700;
do_list (function s -> draw_string s;moveto 0 (snd (current_point()) -
17  )) liste;;

exception numÈro of int;;
let rec convertir chaine = let n = string_length chaine and res = ref "" in
try
for i=0 to (n-1) do
if chaine.[i]=`\n` then raise (numÈro i)
else if chaine.[i] = `\\` then res:=!res^"\\"
else res:=!res^(string_of_char chaine.[i])
done;
[!res]
with
numÈro i -> [!res]@(convertir (sub_string chaine (i+1) (n-i-1)));;

let en3D p = let ppas = !pas*3/2 in let m = ref (mouse_pos ()) and position
= ref p and bouge = ref false and
color_vect = make_vect 54 v and entrÈ = ref false and color = ref id and
color2 = ref black and var = ref true and
var2 = ref true and rÈsolv = ref [] and
tab_centres = [|(b1,b2) add (3*ppas,7*ppas);(b1,b2) add (3*ppas,5*ppas);
(b1,b2) add (5*ppas,5*ppas);
(b1,b2) add (ppas,3*ppas);(b1,b2) add (3*ppas,3*ppas);(b1,b2) add
(3*ppas,ppas)|]
and lastmov = ref id and last = ref false and tab_mvts = [|v;r;w;b;j;o|]
and tab_mvts2 = [|V;R;W;B;J;O|] and
tassoc = [(green,1);(red,2);(white,3);(blue,4);(yellow,5);(orange,6)] and
sens = ref 0 and listmov = ref [] in
draw3D !position;clear();
while !var do
```

```
let k = wait_next_event [ Key_pressed ; Mouse_motion;Button_down] in
let t = k.mouse_x,k.mouse_y and m1 = ref 0 and m2 = ref 0 and n1 = ref 0
and n2 = ref 0 in
if fst t <=115 & button_down() then begin
moveto 750 240;draw_string "Mode: ROTATION 3D               ";
let k = wait_next_event [Mouse_motion;Button_down;Key_pressed] in
m:=k.mouse_x,k.mouse_y;let k = ref (wait_next_event
[Mouse_motion;Button_down;Key_pressed]) in
while  button_down() = false & !k.key=`\000` do
let t = !k.mouse_x,!k.mouse_y in
if maxixi t <= 700 then
begin if t <> !m then begin let a,b = t sub !m in m:=t;
rotation_horizontale a;rotation_transversale b;clear();
draw3D2 !position end;
end ;
k:=wait_next_event [Mouse_motion;Button_down;Key_pressed]
done;moveto 750 240;draw_string "Mode: rotation des faces"
end else
if maxixi t <= 700 & button_down() then
try
let f = faces_visibles() in
begin
let (a,b) = t in m1:=assoc (point_color a b) tassoc; (*cela ne sert qu'‡
arrÍter si le curseur est sur le noir*)
while button_down() do () done;
(*coin et face d'arrivÈe*)
draw3D id;
let tx,ty = mouse_pos() in
m1:=assoc (point_color tx ty) tassoc;
n1:=next_coin t !m1;
n2:=next_coin (tx,ty) !m1;
if (!n1) = (!n2) then
draw3D !position
else begin
for i = 0 to 5 do
let mm = (list_of_vect coins_par_face.(i)) in
if i <> (!m1 - 1) & (intersect [!n1;!n2] mm) = [!n1;!n2]  then begin
if  hd (tl (placer (!n1) [] mm)) = (!n2) then
begin
position:=!position x tab_mvts2.(i);listmov:=tab_mvts2.(i)::(!listmov) end
else begin position:=!position x tab_mvts.(i) ;listmov:=tab_mvts.(i) :: (!
listmov) end;
end
done;
draw3D2 !position;
end
end
with _ -> draw3D !position;
else
begin
if button_down() then
for i = 0 to 5 do
if norme (t sub tab_centres.(i)) < (ppas*ppas/9) then
if !entrÈ then
begin color:=tab_mvts.(i);color2:=point_color k.mouse_x k.mouse_y;
set_color !color2;fill_rect 900 200 50 50;set_color black;moveto 900
200;lineto 950 200;lineto 950 250;
lineto 900 250;lineto 900 200
end
else
begin
if !sens = 0 then
begin position:=!position x tab_mvts.(i);listmov:=tab_mvts.(i) :: (!
listmov) end else
begin position:=!position x tab_mvts2.(i);listmov:=tab_mvts2.(i) :: (!
listmov) end;
clear();
draw3D2 !position;end
done;
if button_down() then
if !entrÈ then begin
for i = 0 to 5 do
if norme (t sub tab_centres.(i)) < (ppas*ppas) then
begin
```

```
let (a,b) = ((!pas+1)/2) div (t sub tab_centres.(i)) and fonc e = if e = 2
or  e = 3 then 1 else
if e = (-2) or e = (-3) then (-1) else e in
let a,b = (fonc a), (fonc b) and c,d = tab_centres.(i) in
if (a,b) <> (0,0) then begin
color_vect.(9*i+a+1+3*(1-b)) <- !color; set_color !color2;
fill_rect (c+ 1+(a+1)*(!pas)-ppas) (d +1+(b+1)*(!pas) - ppas) (!pas - 1) (!
pas - 1)
end
end
done;
end
end;
if k.keypressed & !rÈsolv = [] then
begin
begin
match k.key with
`w` -> position:=!position x w;listmov:=w :: (!listmov)
|`j` -> position:=!position x j;listmov:=j :: (!listmov)
|`o` -> position:=!position x o;listmov:=o :: (!listmov)
|`r` -> position:=!position x r;listmov:=r :: (!listmov)
|`v` -> position:=!position x v;listmov:=v :: (!listmov)
|`b` -> position:=!position x b;listmov:=b :: (!listmov)
|`W` -> position:=!position x W;listmov:=W :: (!listmov)
|`J` -> position:=!position x J;listmov:=J :: (!listmov)
|`O` -> position:=!position x O;listmov:=O :: (!listmov)
|`R` -> position:=!position x R;listmov:=R :: (!listmov)
|`V` -> position:=!position x V;listmov:=V :: (!listmov)
|`B` -> position:=!position x B;listmov:=B :: (!listmov)
|`H` -> dessiner (convertir help_string);let k = wait_next_event
[Key_pressed] in ();clear_graph();
|`h` -> dessiner (convertir help_string);let k = wait_next_event
[Key_pressed] in ();clear_graph();
|`m` -> let k = !position in position:=rand_pos();listmov:=( (inverse k) x
(!position))  :: (!listmov)
|`S` -> position_sauvÈe:=!position
|`l` -> set_color black; moveto 300 400 ;draw_string ("Nombre de positions
isomorphes: "^
(string_of_int (morph !position))); let k = wait_next_event [Key_pressed]
in ();
|`p` -> set_color black; moveto 300 400; if paritÈ !position = 1 then
draw_string "Position paire" else
draw_string "Position impaire" ; let k = wait_next_event [Key_pressed] in
();
|`i`  ->position:= inverse !position; listmov:= ((!position) x (!
position)):: (!listmov)
|` ` -> moveto 750 240;draw_string "Mode: ROTATION 3D               ";
let k = wait_next_event [Mouse_motion] in
m:=k.mouse_x,k.mouse_y;let k = ref (wait_next_event
[Mouse_motion;Button_down;Key_pressed]) in
while  button_down() = false & !k.key=`\000` do
let t = !k.mouse_x,!k.mouse_y in
if maxixi t <= 700 then
begin if t <> !m then begin let a,b = t sub !m in m:=t;
rotation_horizontale a;rotation_transversale b;clear();
draw3D2 !position end; end ;
k:=wait_next_event [Mouse_motion;Button_down;Key_pressed]
done;moveto 750 240;draw_string "Mode: rotation des faces"
|`c` -> sens:=1- (!sens);moveto 770 680;if !sens=1 then draw_string "Sens
trigonomÈtrique  " else
draw_string "Sens horaire                    "
|`\013` -> if !entrÈ then begin try
set_color white;fill_rect 900 200 51 51;
let k = !position in entrÈ:=false;position:=to_pos (list_of_vect
color_vect);listmov:= ((inverse k) x (!position))::(!listmov);
if trace !position <> [0;0;0] then begin
moveto 250 400;
set_color black;draw_string "Attention: cette position est inaccessible";
let k = wait_next_event [Key_pressed] in ();
end
with _ ->  moveto 300 400;
set_color black;draw_string "Erreur dans la position"; let k =
wait_next_event [Key_pressed] in ();
end else begin entrÈ:=true;var2:=false;clear_graph();
```

```
set_color gris;let p = !pas*3 in
fill_poly [|
b1+p,b2;b1+p,b2+p;b1,b2+p;b1,b2+2*p;b1+p,b2+2*p;b1+p,b2+4*p;b1+2*p,b2+4*p;
b1+2*p,b2+3*p;b1+3*p,b2+3*p;b1+3*p,b2+2*p;b1+2*p,b2+2*p;b1+2*p,b2|];
dessine_structure() end
|`s` -> moveto 300 400;if trace !position = [0;0;0] then begin
draw_string "Recherche d'une solution en cours...";
rÈsolv:= snd (rÈsoudre !position) end else begin
draw_string "Cette position n'est pas accessible !";let k = wait_next_event
[Key_pressed] in (); end
|`q` -> var:=false
|`Z` -> if !last then begin last:=false;position:=!position x (!
lastmov);listmov:= (!lastmov) :: (!listmov) end
|`z` -> if !listmov <> [] then begin
last:=true;position:=!position x (inverse (hd !listmov));lastmov:=(hd !
listmov);listmov:= tl !listmov; end
|`a` ->moveto 300 400;if (trace !position) <> [0;0;0] then draw_string
"Cette position n'est pas accessible !"
else draw_string "Cette position est accessible !";let k = wait_next_event
[Key_pressed] in ()
|`Q` -> quit()
|`\027` -> listmov:= (inverse !position) :: (!listmov);position:=id
| _ -> var2:=false; end;
if !var2 then
begin clear();
draw3D2 !position; end;var2:=true end;
if k.keypressed & !rÈsolv <> []  then match k.key with
|`\013` -> if !rÈsolv <> [] then begin position:= !position x tabu.(hd !
rÈsolv);rÈsolv:=tl !rÈsolv;clear();draw3D2 !position  end
|`q` -> begin rÈsolv:=[];clear();draw3D2 !position end
|_ -> ();
done;;

let cube() = open_graph "800x800+0+0";
moveto 750 240 ; draw_string "Mode: rotation des faces";
moveto 770 680 ; draw_string "Sens horaire                    ";
draw_pos id; en3D id;;
```