COMPRESSION

```
PROCEDURE initCompression (c, meshType) {       #c is a starting corner, meshType: manifold or t-patch
…GLOBAL M[]={0…}, U[]={0…};                     # init tables for marking visited vertices and triangles
…GLOBAL T = 0;                                  # id of the last triangle compressed so far
… EncodeDelta (c.n);                            # estimate first vertex and encode as delta in vertices file
…IF meshType ==manifold THEN M[c.n.v] = 1;      # if we do not have a border mark first vertex as visited
… EncodeDelta (c);     M[c.v] = 1;              # estimate third vertex and mark it as visited
… EncodeDelta (c.p);  M[c.p.v] = 1;             # estimate second vertex and mark it as visited
…U[c.t] = 1;                                    # mark the first triangle as visited
…a = c.o;                                       # a is corner for triangles incident on first vertex
…count = 1;                                     # init number of triangles incident on the first corner
…WHILE a != c.p.o.p   DO{                       # traverse fan of 'C' triangles incident on
……U[a.t] = 1;  T++; count++;                    # paint the triangle, increment # of triangles
…… EncodeDelta (a);  M[a.v] = 1;                # estimate next vertex and mark it as visited
……a = a.n.o;}                                   # continue around with the right neighbor of a
…U[a.t] = 1;  T++; count++;                     # mark the 'R' triangle incident on the first vertex
…WRITE(clers, meshType);                        # encode meshType in clers file
…WRITE(clers, count);                           # encode number of triangles incident on first vertex
…Compress(a.p.o )}                              # start compression with triangle adjacent to 'R'


RECURSIVE PROCEDURE Compress(c) {               # compressed  the rest of t-meshes starting with corner c
…REPEAT {                                       # visits triangle-spanning tree until matching RETURN
……U[c.t] = 1;  T++;                             # mark current triangle as visited, increments triangle count
……IF c.n.o.t.u  > 1 THEN {                      # checks for handles from right
………WRITE(handles, c.n.o.t.u);                  # encodes pair of opposite corners to be glued for handle
………WRITE(handles , T*3+1)}
……IF c.p.o.t.u  > 1 THEN {                      # checks for handles from left
………WRITE(handles, c.p.o.t.u);                  # encodes pair of opposite corners to be glued for handle
………WRITE(handles , T*3+2)}
……IF c.v.m != 1                                 # test whether 'C' (tip vertex) was not visited
………THEN {WRITE(clers, 'C');                    # IF WAS NOT, appends encoding of 'C' to clers
…………EncodeDelta(c);  M[c.v] = 1;               # estimate next vertex and mark it as visited
…………c = c.r}                                   # continue with the right neighbor
………ELSE IF c.r.t.u > 0                         #IF WAS,  test whether right triangle was visited
…………THEN IF c.l.t.u > 0                        # test whether left triangle was visited
……………THEN {WRITE(clers, 'E'); RETURN }        # append code for 'E' and pop stack pushed by 'S'
……………ELSE  {WRITE(clers, 'R'); c = c.l }      # append code for 'R', move to left neighbor
…………ELSE IF c.l.t.u  > 0                        # test whether left triangle was visited
……………THEN {WRITE(clers,' L'); c = c.r }       # append code for 'L', move to right triangle
……………ELSE {U[c.t] = T*3+2;                    # store corner number in decompression (potential handle)
………………WRITE(clers, 'S');                     # append code for 'S'
………………Compress(c.r);                         # recursive call to first visit right branch of split
………………c = c.l;                               # upon return, move to left triangle
………………IF c.t.u > 0 THEN RETURN}}}            # if the triangle to the left was visited (handle), then return


PROCEDURE EncodeDelta(c) {                      # if neighbors a,b, and c were visited, uses parallelogram a+b-d
…IF c.o.v.m > 0  && c.p.v.m > 0 THEN {pred = (c.n.v.d+c.p.v.d-c.o.v.d); delta = c.v.g – pred} # a, b d known (case 1)
…ELSE  IF c.o.v.m > 0 THEN {pred = (2*c.n.v.d -c.o.v.d); delta = c.v.g – pred} # a and d are known (case 2)
…ELSE  IF c.n.v.m > 0  && c.p.v.m > 0 THEN {pred = (c.n.v.d +c.p.v.d)/2; delta = c.v.g – pred}  # a, b known (case 3)
…ELSE  IF c.n.v.m > 0 THEN {pred = c.n.v.d ; delta = c.v.g – pred}                      # a is known (case 4)
…ELSE  IF c.p.v.m > 0 THEN {pred = c.p.v.d;  delta = c.v.g – pred }                     # b is known (case 5)
…ELSE {pred =  {0,0,0}; delta =  c.v.g - pred}  # nothing is known (case 6)
…D[c.v] = delta + pred;                         # update vertex as it will be decoded for future predictions
…WRITE(vertices, delta)}                        # store corrective vectors in the vertices file
```

DECOMPRESSION

```
PROCEDURE initDecompression {
…GLOBAL V[] = { 0,2,1,0,0,0,0,0,…};               # table of vertex Ids for each corner
…GLOBAL O[] = {–1,–1,–3, –3, –3, –3…};            # table of opposite corner Ids for each corner
…GLOBAL T = 0;                                    # id of the last triangle decompressed so far
…GLOBAL N = 2;                                    # id of the last vertex encountered
…GLOBAL A = 0;                                    # id of the last handle encountered
…H = READ (handles)                               # read handle pairs from handles file into array H
…GLOBAL meshType = READ(clers);                   # read meshType from clers file
…GLOBAL I = READ(clers);                          # read number of incident triangles on first vertex
…WRITE (“C,C,…C,R”, clears);                      # append (I-2) Cs and 1R to the beginning of clers file
…DecompressConnectivity(2);                       # start connectivity decompression
…GLOBAL M[]={0…}, U[]={0…};                       # init tables for marking visited vertices and triangles
…G[0] = DecodeDelta (0);                          # estimate 1st  vertex
…IF meshType == manifold  THEN M[0] = 1;          # if we do not have a hole mark 1st  vertex as visited
…G[1] = DecodeDelta (2); M[1] = 1;                # estimate third vertex and mark it as visited
…G[2] = DecodeDelta (1); M[2] = 1;                # estimate second vertex and mark it as visited
…GLOBAL N = 2;                                    # id of the last vertex encountered
…U[0] = 1;                                        # paint the triangle and go to opposite corner
…DecompressVertices(O[2]);}                       # start vertices decompression


RECURSIVE PROCEDURE DecompressConnectivity(c) {
…REPEAT {                                         # Loop builds triangle tree and zips it up
……T++;                                            # new triangle
……O[c] = 3T; O[3T] = c;                           # attach new triangle, link opposite corners
……V[3T+1] = c.p.v; V[3T+2] = c.n.v;               # enter vertex Ids for shared vertices
……c = c.o.n;                                      # move corner to new triangle
……Switch READ(clers) {                            # select operation based on next symbol
………Case  C: {O[c.n] =  –1; V[3T] = ++N;}         # C: left edge is free, store ref to new vertex
………Case  L: {O[c.n] =  –2;                       # L: orient free edge
…………IF !CheckHandle(c.n) THEN zip(c.n);}          # check for handles, if non, try to zip
………Case  R: {O[c]= –2; CheckHandle(c); c = c.n; }         # R: orient free edge, check for handles, go left
………Case  S: {DecompressConnectivity (c);  c = c.n;        # S: recursion going right, then go left
…………IF c.o >=0 DO RETURN; }                       # if the triangle to the left was visited, then return
………Case  E: {O[c] =  –2; O[c.n] =  –2;           # E: left and right edges are  free
…………CheckHandle(c);                              # check for handles on the right
…………IF !CheckHandle(c.n) THEN zip(c.n);          # check for handles on the left, if non, try to zip
…………RETURN }}}}                                  # pop


PROCEDURE BOOLEAN CheckHandle(c) {
…IF  c != H[A+1] OR A >= sizeof(H) THEN RETURN FALSE ELSE { # check if this is a handle
……O[c] = H[A];  O[H[A]] = c;                      # link opposite corners
……a = c.p; WHILE a.o>=0 && a!= H[A] DO {a=a.o.p;}       # find corner of next free edge if any
……IF a.o == -2 DO Zip(a);                         # zip if found cw edge
……a = c.o.p; WHILE a.o>=0 && a!= c DO {a=a.o.p;}        # find corner of next free edge if any
……IF a.o == -2 THEN Zip(a);                       # zip if found cw edge
……A+=2;                                           # next handle
……RETURN TRUE}}


RECURSIVE PROCEDURE Zip(c) {                      # tries to zip free edges opposite c
…b = c.n; WHILE b.o>=0 && b.o!=c DO b=b.o.n;      # search clockwise for free edge
…IF b.o != –1 THEN RETURN;                        # pop if no zip possible
…O[c]=b; O[b]=c;                                  # link opposite corners
…a = c.n;  V[a.n] = b.n.v;                        # assign co-incident corners
…WHILE a.o>=0 && a!=b DO {a=a.o.n; V[a.n]=b.n.v};       # update all incident corners to zipped vertex
…c = c.p; WHILE c.o >= 0 && c!= b DO c = c.o.p;   # find corner of next free edge on right
…IF c.o == –2 THEN Zip(c) }                       # try to zip again
```

```
RECURSIVE PROCEDURE DecompressVertices(c) {
…REPEAT {                                      # start traversal for triangle tree
……U[c.t] = 1;                                  # mark the triangle as visited
……IF c.v.m != 1 THEN {                         # test whether tip vertex was visited
………G[++N] = DecodeDelta (c);                  # update new vertex
………M[c.v] = 1;                                # mark tip vertex as visited
………c = c.r;}                                  # continue with the right neighbor
……ELSE IF c.r.t.u == 1                         # test whether right triangle was visited
………THEN IF c.l.t.u == 1                       # test whether left triangle was visited
…………THEN  RETURN                              # pop
…………ELSE  { c = c.l }                         # move to left triangle
………ELSE IF c.l.t.u == 1                       # test whether left triangle was visited
…………THEN { c = c.r }                          # move to right triangle
…………ELSE { DecompressVertices (c.r);          # recursive call to visit right branch first
……………c = c.l ;                               # move to left triangle
……………IF c.t.u > 0 THEN RETURN}}}             # if the triangle to the left was visited, then return


PROCEDURE DecodeDelta(c) {                     # uses parallelogram if neighbors are known
…delta = READ(vertices);                       # read next vertex delta
…IF c.o.v.m > 0  && c.p.v.m > 0 THEN RETURN  (delta + (c.n.v.g+c.p.v.g-c.o.v.g));     # a, b, d known (case 1)
…IF c.o.v.m > 0 THEN RETURN   (delta + (2*c.n.v.g -c.o.v.g));                         # a, d known (case 2)
…IF c.n.v.m > 0  && c.p.v.m > 0 THEN RETURN  (delta + (c.n.v.g +c.p.v.g)/2);          # a, b known (case 3)
…IF c.n.v.m > 0 THEN RETURN  (delta + c.n.v.g);  # a is known (case 4)
…IF c.p.v.m > 0 THEN RETURN  (delta + c.p.v.g);  # b is known (case 5)
…RETURN (delta) }                              # no known neighbors
```