

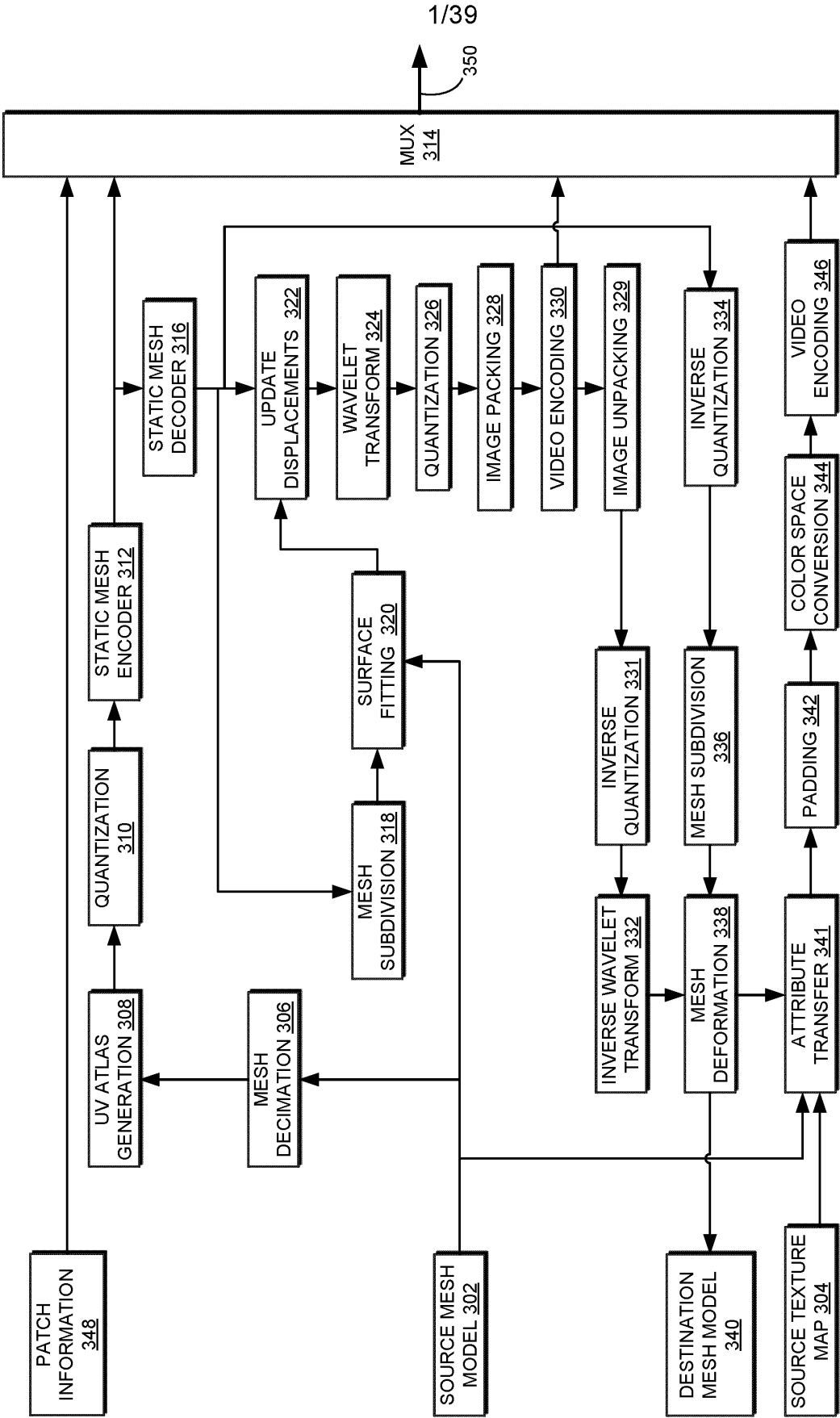


FIG. 1

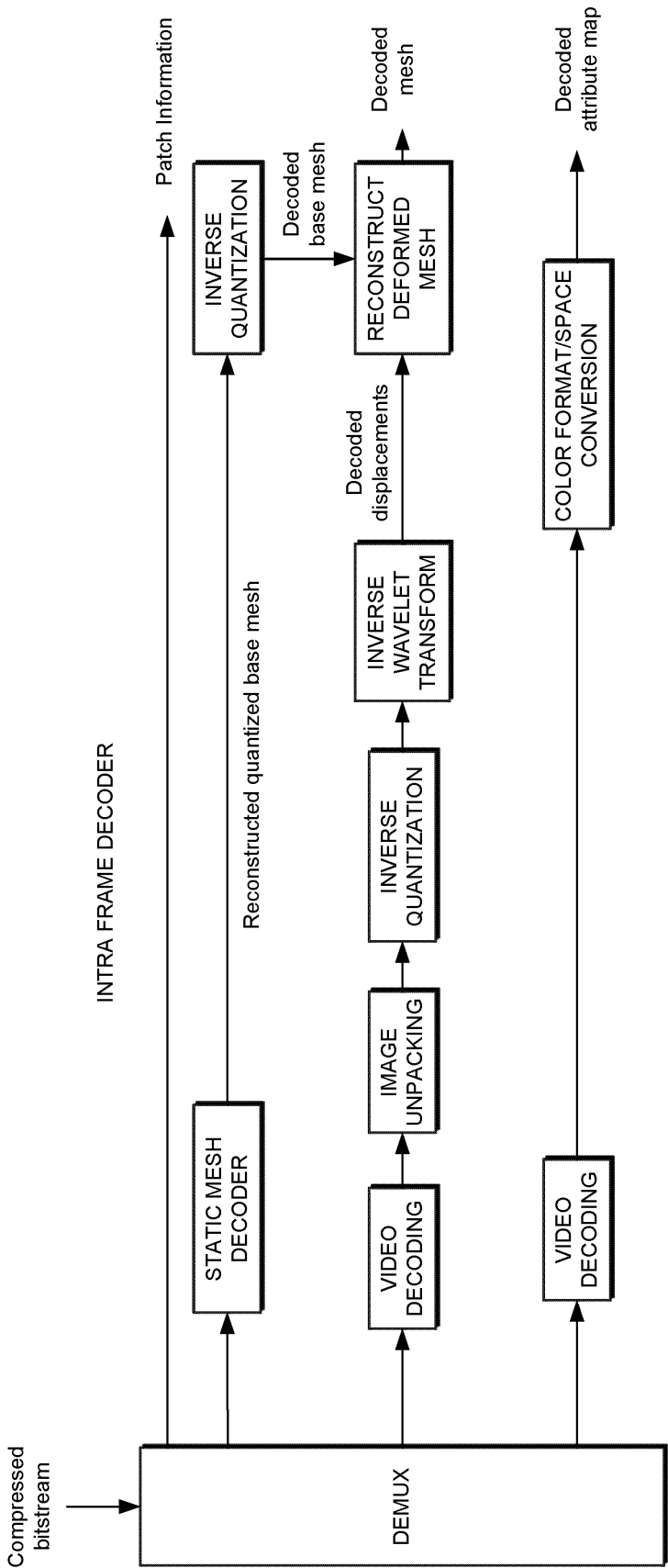


FIG. 2

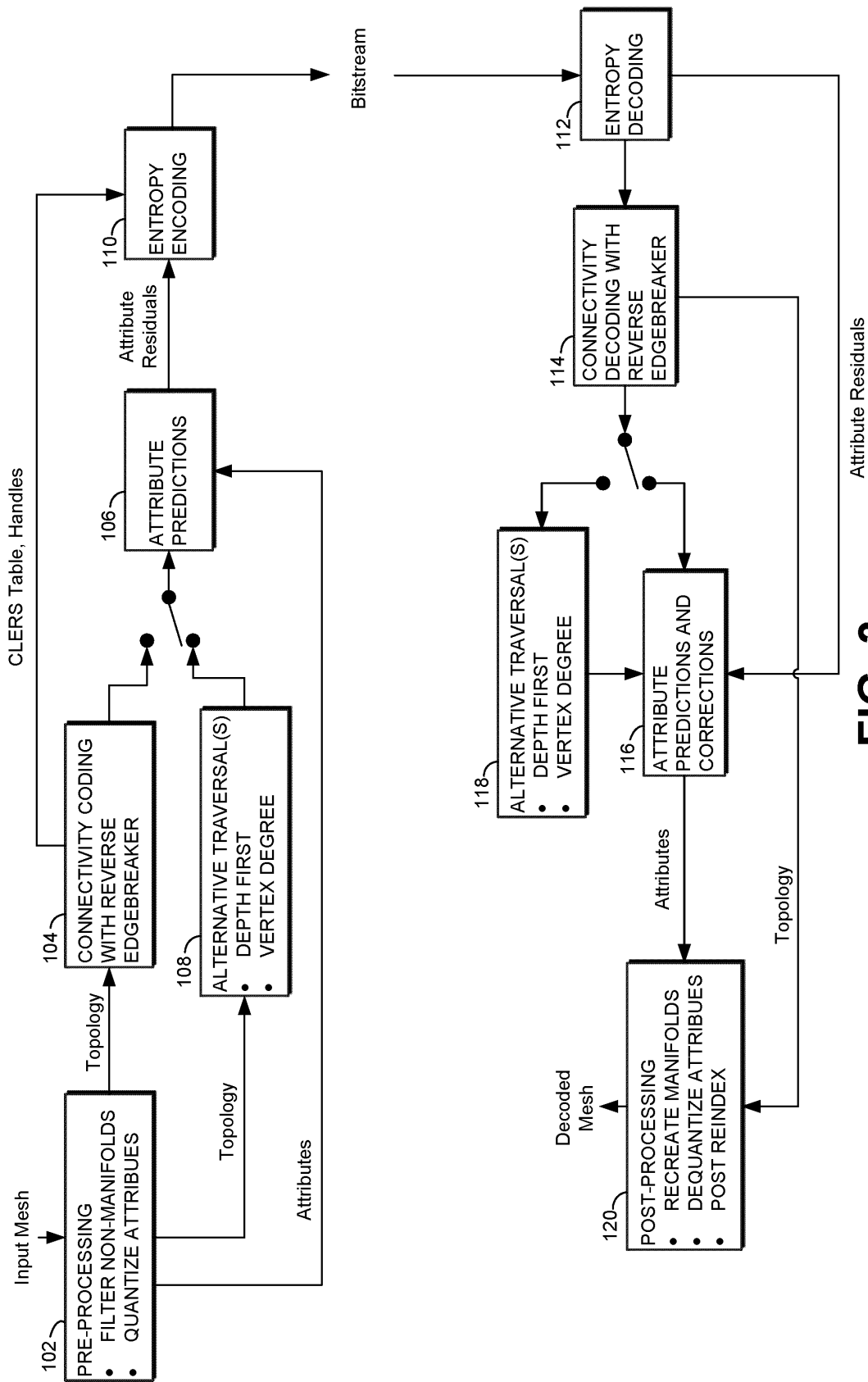


FIG. 3

```

void EBReversiEncoder::encode(const Model& input) {

    numOfInputTriangles = input.getTriangleCount(); // store number of triangles of the input mesh
    numOfInputVertices  = input.getPositionCount(); // store number of vertices of the input mesh

    // alias
    auto& ov = _ovTable; // the corner table
    auto& V  = _ovTable.V; // vertex indices
    auto& O  = _ovTable.O; // Opposite corners
    auto& TC = _ovTable.TC; // texture coordinates indices

    // first convert input triangle mesh into a corner table named OVTable (or ov)
    convertModelToCTMesh(input, _ovTable, cfg.deduplicate, cfg.useMainIndexIfEqual);

    // Quantize the attributes of the Corner table (CT) mesh if option is on
    if (!cfg.intAttr) {
        quantize();
    }

    // init some global variables
    nV = ov.getPositionCount(); // vertex count in OV table, may include some isolated vertices
    nT = ov.getTriangleCount(); // triangle count in OV table,
    nUV = _ovTable.getUVCount(); // UV coordinates count in OV table,

    // Allocate memory for tables
    MV.assign(std::max(nV, nUV), 0); // Table for marking visited vertices and Uvs (may be
    // oversized due to potential isolated vertices that
    // will be skipped)
    MT.assign(nT, 0); // Table for marking visited triangles
    MC.assign(nUV, 0); // Table for marking visited corners for uv with sep index
    processedCorners.reserve(ov.V.size());
    oVertices.reserve(ov.positions.size());
    oUVCoords.reserve(ov.positions.size());

    // encode the topology using edge breaker
    encodeTopology();

    // encoding of main index without predictions
    // is performed during encodeTopology
    if (cfg.posPred != EBConfig::PosPred::NONE){
        // encode attributes using main index table
        encodeAttributesWithMainIndexTable();
    }

    // encode attributes using auxiliary index table, if any
    if (TC.size() != 0) {
        encodeAttributesWithAuxiliaryIndexTable();
    }

    // encode the per face attributes
    encodePerFaceAttributes();
}

```

FIG. 4

```

bool EBReversiDecoder::decode(Model& output)
{
    auto& ov = _ovTable;
    auto& V = _ovTable.V;
    auto& O = _ovTable.O;
    auto& G = _ovTable.positions;
    auto& UV = _ovTable.uvcoords;
    auto& TC = _ovTable.TC;
    auto& OTC = _ovTable.OTC;

    // Allocate Memory for Tables

    V.resize(3 * nT, -1); // table of vertex Ids for each corner
    O.resize(3 * nT, -1); // table of opposite corner Ids for each corner or free edge orientation
    G.resize(nV, vec3(0.0)); // Allocate memory for geometry array
    UV.resize(nUV, vec2(0.0)); // Allocate memory for uv coordinates array
    if (hasSeparateUvIndex) {
        TC.resize(3 * nT, -1);
        OTC.resize(3 * nT, -3);
    }
    if (iNotPredictedFaceId.size() != 0)
    {
        UF.assign(nT, 0);
        _ovTable.ids.assign(nT, -1);
    }
    FC.resize(nB, 0); // FC[0] = 0 first corner, the rest will be filled by decompress
    MV.resize(std::max(nV, nUV), 0); // vertex marking table

    // parse and transform the handles input table and the input CLERS string
    // to make them suitable for the edgebreaker decoding
    topoPrepass();

    // start the spiral reversi
    spiraleReversi();

    // decode attributes using main index table
    if (cfg.posPred != EBConfig::PosPred::NONE)
        decodeAttributesWithMainIndexTable();

    // decode attributes using auxiliary index table if any
    if (hasSeparateUvIndex)
        decodeAttributesWithAuxiliaryIndexTable();

    // decode per face attributes
    if (hasFID)
        decodePerFaceAttributes();

    // finalize the deduplication if needed
    if (cfg.deduplicate)
        applyIndexDeduplicatePerFan();

    // Dequantize the attributes of the Corner table (CT) mesh if option is on
    if (!cfg.intAttr)
        dequantize();

    // Reindex the topology if option is on
    if (cfg.reindexOutput)
        postReindex();

    // convert the decoded corner table back to a triangle set
    convertCTMeshToModel(_ovTable, output, cfg.deduplicate);

    return true;
}

```

FIG. 5

```

void EBReversiEncoder::encodeTopology () {

    _T = 0; // index of last triangle
    nB = 0; // number of components with boundary

    bool hasBoundary = false; // this is to be lazy if no boundary
    for (auto c = 0; c < 3 * nT; ++c) { // tests all corners for a boundary
        if (O[c] < 0) { // looks for edges on the boundary
            hasBoundary = true;
            MV[V[ov.p(c)]] = 1; // mark vertices
            MV[V[ov.n(c)]] = 1;
            O[c] = -1; // reset the boundary ID
        }
    }

    if (hasBoundary) // if it has a boundary, process first all the components with boundary
    {
        for (auto c = 0; c < 3 * nT; ++c) { // compress components with boundary
            if (O[c] < 0 && !MT[ov.t(c)]) { // starts from an unvisited border triangle
                writeBoundary(ov.n(c)); // encodes the geometry of this boundary
                _T--;
                ccStartCorners.push_back(c); // record the start corner of the component
                compressComponent(c); // starts the compression of this component
                nB++; _T++;
            }
        }
    }

    // test all triangles
    if (_T != ov.ntri())
    {
        for (auto c = 0; c < 3 * nT; ++c) { // compress component without boundary
            if (!MT[ov.t(c)]) { // starts with an unvisited triangle
                MT[ov.t(c)] = 1; // marks triangle as visited
                MV[V[c]] = 1; // marks vertices as visited
                MV[V[ov.n(c)]] = 1;
                MV[V[ov.p(c)]] = 1;
                ccStartCorners.push_back(c); // record the start corner of the component
                processedCorners.push_back(c); // record corner processing order
                writeVertex(c); // encodes the geometry of this triangle (c, c.n, c.p)
                writeVertex(ov.n(c));
                writeVertex(ov.p(c));
                compressComponent(ov.r(c)); // starts the compression of this component
                _T++;
            }
        }
    }
}

```

FIG. 6

```

void EBReversiEncoder::compressComponent(int _c) {

    auto& ov = _ovTable; // alias
    auto& V = _ovTable.V; // alias
    auto& O = _ovTable.O; // alias
    std::stack<int> stack; // stack of corners

    int c = _c; // init start corner c
    while (true) { // visits the triangle spanning tree
        // notations for correspondence with paper
        const auto c_r = ov.r(c); // right corner of current corner c
        const auto c_l = ov.l(c); // left corner of current corner c
        const auto c_t = ov.t(c); // triangle index of current corner c
        const auto c_r_t = ov.t(c_r); // triangle index of right corner of current corner c
        const auto c_l_t = ov.t(c_l); // triangle index of left corner of current corner c
        const auto c_v = ov.v(c); // vertex index of current corner c
        processedCorners.push_back(c); // record the corner in the processed corners table
        MT[c_t] = 1; _T++; // marks current triangle as visited
        checkHandle(c); // checks for handles (paper)
        if (MV[c_v] != 1) { // tests whether the tip vertex was visited
            oClers.push_back('C'); // case C: unvisited tip vertex
            writeVertex(c); // encodes the vertex of the corner and marks it
            MV[c_v] = 1;
            c = c_r; // continues with the right neighbor
        }
        else if (c_r < 0 || MT[c_r_t] != 0) { // tests whether right triangle was visited
            if (c_l < 0 || MT[c_l_t] != 0) { // tests whether left triangle was visited
                oClers.push_back('E'); // case E: both visited
                do {
                    if (stack.empty()) // if the stack is empty
                        return;
                    else {
                        c = stack.top(); // otherwise, pops stack pushed by S
                        stack.pop();
                    }
                } while (MT[ov.t(c)] != 0); // pops until finding a non handle corner
            }
            else { // case R: right visited, left not; moves left
                oClers.push_back('R');
                c = c_l;
            }
        }
        else if (c_l < 0 || MT[c_l_t] != 0) { // tests whether left triangle was visited
            oClers.push_back('L'); // case L: left visited, right not;
            c = c_r; // moves right
        }
        else {
            oClers.push_back('S'); // case S: both unvisited
            if (ov.isOnBoundary(c) == -1) { // if the corner is on an unvisited boundary
                writeBoundary(c); // encodes the internal boundary
                MT[c_t] = -3 * _T - 2; // stores corner number (potential boundary)
            }
            else { // else stores corner number (potential handle)
                MT[c_t] = 3 * _T + 2;
            }
            stack.push(c_l); // pushes the left corner on the stack
            c = c_r; // moves right
        }
    }
}

```

FIG. 7

```
void EBReversiEncoder::writeBoundary(int _b) {  
  
    auto& ov = _ovTable;  
    auto& V = _ovTable.V;  
    auto& O = _ovTable.O;  
  
    nBounds = nBounds + 1;           // increase the boundary Id  
  
    auto b = ov.n(_b);                // goes to the next triangle,  
    while (ov.r(b) >= 0) {            // goes to right towards the boundary  
        b = ov.r(b);  
    }  
    do {                              // starts the boundary encoding  
        b = ov.n(b);                 // starts from the next triangle  
        O[b] = -(nBounds + 2);        // and marks the boundary  
        while (ov.r(b) >= 0) {        // goes to the rightmost triangle  
            b = ov.r(b);  
        }  
        writeVertex(ov.p(b));         // encodes the vertex of the corner  
    } while (ov.r(b) != -(nBounds + 2)); // stops when all the boundary is marked  
}
```

FIG. 8

```

void EBReversiEncoder::checkHandle(const int h) {
    // aliases
    auto& ov = _ovTable;
    auto& V = _ovTable.V;
    auto& O = _ovTable.O;
    const auto h_r = ov.r(h);
    const auto h_l = ov.l(h);
    const auto h_r_t = ov.t(h_r);
    const auto h_l_t = ov.t(h_l);

    const int br = (h_r >= 0) ? MT[h_r_t] : -1;
    if (br != 0 && br != 1 && br != -1) { // checks for handle from the right
        oHandles.push_back(br); // encodes pair of corners to be glued as a handle
        oHandles.push_back(_T * 3 + 1);
    }

    const int bl = (h_l >= 0) ? MT[h_l_t] : -1;
    if (bl != 0 && bl != 1 && bl != -1) { // checks for handle from the left
        oHandles.push_back(bl); // encodes pair of corners to be glued as a handle
        oHandles.push_back(_T * 3 + 2);
    }
}

```

FIG. 9

```

void EBReversiEncoder::writeVertex(const int c) {

    const auto& V = _ovTable.V;
    const int& v = V[c];
    const auto& UV = _ovTable.uvcoords;
    const auto& OTC = _ovTable.OTC;
    const bool predictUVs = (UV.size() && !OTC.size()); // predict UVs in first pass if no separate index

    bool write = false;

    // if predictions are off, positions are pushed at the end of the output vertices table
    if (cfg.posPred == EBConfig::PosPred::NONE) {
        oVertices.push_back(_ovTable.positions[v]);
    }
    // if predictions are on, those will be performed and stored in a second pass.
    else write = true;

    // if predictions are off and model has Uvs, UVs are pushed at the end of the output vertices table
    if (predictUVs && cfg.uvPred == EBConfig::UvPred::NONE) {
        oUVCoords.push_back(UV[V[c]]);
    }
    // if predictions are on, those will be performed a second pass.
    else write = true;

    // used in second pass, to preserve corners order to be used for predictions
    if (write && cfg.traversal != EBConfig::Traversal::EB)
        writtenCorners.push_back(c);
}

```

FIG. 10

```

void EBReversiDecoder:: topoPrepass(void)
{
    auto& ov = _ovTable;
    auto& V = _ovTable.V;
    auto& O = _ovTable.O;
    // Reconstruct handles, filter between real handles and
    // boundaries that are also coded using handles
    // init O table with each pair of handle
    for (auto i = 0; i < iHandles.size() / 2; ++i) {
        const int b = iHandles[i * 2 + 0];
        const int c = iHandles[i * 2 + 1];
        if (b > 0) // handle
        {
            O[b] = c;
            O[c] = b;
        }
        else // boundary
        {
            O[-b] = -c;
            O[c] = b;
        }
    }
    // parse the CLERS string in forward order and construct the enriched table of symbols
    clersIndex = 0; // index to consume clers table by invoking readClers()
    _T = nC = 0; // set current triangle and number of components to 0
    ccStartCorners.clear();
    symb.clear();
    symb.reserve(iClers.size()); // reserve memory for the extended symbol table (incl P)

    int i = 0; // number of parentheses
    if (nB == 0) { // if the next comp has no boundary, add P symbol
        ccStartCorners.push_back(ov.p(0));
        symb.push_back('P');
        _T++;
    }
    else
        ccStartCorners.push_back(0);
    while (_T < nT) { // loop on all the triangles
        const char s = readClers(); // but the first one if no boundary on first CC
        const int c = 3 * _T; // reads the next symbol
        symb.push_back(s); // store the next symbol
        _T++;
        if (s == 'S' && ov.l(c) == -1) { // on an S symbol not related to a handle
            i++; // increase the number of parentheses
        }
        if (s == 'E') { // on an E symbol
            if (i > 0) { // if the S are not all matched
                i--;
            }
        }
        else {
            nC++; // else we increase the number of components
            if (_T < nT) { // record the id of the start corner
                if (nC >= nB) // on a P symbol
                    ccStartCorners.push_back(ov.p(_T * 3));
                else // not on a P symbol
                    ccStartCorners.push_back(_T * 3);
            }
            if (nC < nB && nC >= 0) { // strict comp so we skip the last one
                // (first one is 0, already in table at init)
                FC[nC] = _T; // stores the end of previous component
            }
            if (nC >= nB && _T < nT) { // if the next comp has no boundary, add P symbol if not last CC
                symb.push_back('P');
                _T++;
            }
        }
    }
}

```

FIG. 11

```

void EBReversiDecoder::spiraleReversi(void)
{
    auto& ov = _ovTable; auto& V = _ovTable.V; auto& O = _ovTable.O;
    processedCorners.clear(); processedCorners.reserve(V.size()); // order of corners processing
    readCorners.clear(); // order of corners writing (used by attribute pass)
    readCorners.reserve(V.size()); // all corners might not have reads but we reserve max possible memory
    std::stack<int> stack; // stack of corners
    _T = symb.size() - 1; // current triangle index
    _N = nV - 1; // current vertex index
    while (nC > 0) { // reads the enrich CLERS string in reverse order
        int c = -1; // initiates first corner
        while (_T >= 0 && (nC > nB || (nC >= 0 && _T >= FC[nC - 1]))) // filter components with boundary
        {
            bool isP = false; // used to note if current state is a P
            switch (symb[_T]) { // test the next symbol
                case 'C':
                    match(c, 3 * _T + 1); // matches the right gate
                    closeStar(3 * _T + 2, _N);
                    readCorners.push_back(ov.n(3 * _T + 2));
                    _N--;
                    break;
                case 'L': match(c, 3 * _T + 1); break;
                case 'R': match(c, 3 * _T + 2); break;
                case 'S':
                    match(c, 3 * _T + 1); // matches the right gate
                    c = 3 * _T + 2;
                    if (O[c] == -1) { // if it is not a handle
                        match(c, stack.top()); // match the left gate
                        stack.pop();
                    }
                    else if (O[c] < 0) { // if it is an interior boundary
                        match(c, -O[c]);
                        c = ov.p(c); // addition to the paper
                        while (ov.r(c) >= 0) { // goes backward on the boundary
                            c = ov.r(c);
                        }
                        readBoundary(ov.p(c), _N); // decodes the interior boundary
                    }
                    break;
                case 'E':
                    if (c > 0) {
                        stack.push(c);
                    }
                    break;
                case 'P':
                    isP = true; // used later to store ids in processedCorners
                    match(c, 3 * _T); // matches the gate
                    closeStar(3 * _T + 1, _N - 2); // close star for next corner
                    closeStar(3 * _T + 2, _N - 1); // close star for previous corner
                    closeStar(3 * _T + 0, _N); // close the star of the corner
                    readCorners.push_back(3 * _T + 1); // order +1, +0, +2 is critical
                    readCorners.push_back(3 * _T + 0);
                    readCorners.push_back(3 * _T + 2);
                    _N -= 3; nC--; c = -1; // update for new component,
                    break;
            }
            c = 3 * _T;
            if (isP) // record the id of the processed corner
                processedCorners.push_back(ov.p(c)); // on a P symbol, we store c.p
            else
                processedCorners.push_back(c); // not on a P symbol, we store c
            _T--; // move to previous symbol
        }
        if (nC != 0) {
            nC--;
            readBoundary(3 * FC[nC] + 1, _N); // decodes exterior boundary
        }
    }
}

```

FIG. 12

```

void EBReversiDecoder::readBoundary(int b, int& N)
{
    auto& ov = _ovTable;
    auto& V = _ovTable.V;
    auto& O = _ovTable.O;

    nBounds = nBounds + 1;          // increase the boundary Id (for ease of debugging)

    while (ov.l(b) >= 0) {
        b = ov.l(b);                // go to the left toward the boundary
    }
    do {
        readCorners.push_back(b);
        readVertex(b, N);           // decode vertex geometry
        ov.setV(b, N);
        b = ov.p(b);
        ov.setO(b, -(nBounds + 2)); // marks the corner with the boundId
        while (ov.l(b) >= 0) {       // updates all incident corners to the vertex
            b = ov.l(b);
            ov.setV(ov.n(b), N);
        }
        N--;                        // added decrement N here (not in paper)
    } while (ov.v(b) < 0);          // until closing the boundary
}

```

FIG. 13

```

void EBReversiDecoder::closeStar(int c, int N) {

    auto& ov = _ovTable;
    auto& V = _ovTable.V;
    readVertex(N);                // corresponds to corner ov.n(c)
    int b = c;
    while (ov.l(b) >= 0 && ov.l(b) != c) { // assign incident corners to the new vertex
        ov.setV(ov.n(b), N);
        b = ov.l(b);
    }
    ov.setV(ov.n(b), N);
    match(ov.p(b), c);            // closes the star
}

```

FIG. 14

```

inline bool readVertex(int N) {

    // positions
    if (cfg.posPred == EBConfig::PosPred::NONE)
    {
        _ovTable.positions[N] = iVertices[N];
    }
    if (cfg.uvPred == EBConfig::UvPred::NONE)
    {
        if (iUVCoords.size() != 0 && _ovTable.OTC.size() == 0)
            _ovTable.uvcoords[N] = iUVCoords[N];
    }

    MV[N] = 1; // mark the vertex

    return true;
}

```

FIG. 15

```

void EBReversiDecoder::postReindex() {
    //
    PredictionVertexTraverser traverser;
    traverser.init(&_amp;ovTable);
    for (auto corner : ccStartCorners)
        traverser.traverse(corner);
    //
    std::vector<int> newIdx(_amp;ovTable.positions.size());
    std::vector<glm::vec3> newPos(_amp;ovTable.positions.size());
    // we need to swap uvcoords also if same index
    // and potentially all other attributes sharing the position index
    std::vector<glm::vec2> newUV(hasSeparateUvIndex ? 0 : _amp;ovTable.positions.size());
    for (auto i = 0; i < traverser.visitedCorners.size(); ++i) {
        const auto vIdx = _amp;ovTable.v(traverser.visitedCorners[i]);
        newIdx[vIdx] = i;
        newPos[i] = _amp;ovTable.positions[_amp;ovTable.v(traverser.visitedCorners[i])];
        if (!hasSeparateUvIndex && _amp;ovTable.uvcoords.size() != 0)
            newUV[i] = _amp;ovTable.uvcoords[vIdx];
    }

    _amp;ovTable.positions.swap(newPos);
    if (!hasSeparateUvIndex)
        _amp;ovTable.uvcoords.swap(newUV);

    for (auto c = 0; c < _amp;ovTable.V.size(); ++c) {
        _amp;ovTable.V[c] = newIdx[_amp;ovTable.V[c]];
    }
}

```

FIG. 16

```

void EBReversiEncoder::encodeAttributesWithMainIndexTable(void)
{
    if (cfg.traversal == EBConfig::Traversal::DEGREE) {
        // computes a corner order using prediction degree traversal
        PredictionVertexTraverser traverser;
        traverser.init(&_amp;ovTable);
        for (auto corner : ccStartCorners)
            traverser.traverse(corner);

        // perform the rpedicitons using the generated corner order
        encodeAttributesWithPredition( traverser.visitedCorners );
    }
    else { // otherwise uses the traversal order from topology encoding
        encodeAttributesWithPredition( writtenCorners );
    }
}

```

FIG. 17

```

void EBReversiEncoder::encodeAttributesWithPrediction(vector<int>& traversalOrder)
{
    const auto& ov = _ovTable;
    const auto& V = ov.V;
    const auto& UV = ov.uvcoords;
    const auto& OTC = ov.OTC;
    const bool predictUVs = (UV.size() && !OTC.size()); // predict UVs in first pass if no separate index

    // prepare the skip table if deduplication is active
    std::vector<bool> predictCorner;
    if (cfg.deduplicate) {
        predictCorner.assign(ov.V.size(), true);
    }

    // reset the vertex marking table
    for (auto& v : MV) v = -1;

    // goes through the corners, for the positions
    for (const auto& c : traversalOrder) {

        bool predictPosition = true;
        if (cfg.deduplicate)
        {
            // check for duplicated positions
            const auto dupIt = _ovTable.duplicatesMap.find(_ovTable.V[c]);
            if (dupIt != _ovTable.duplicatesMap.end())
            {
                isVertexDup.push_back(true);
                oDuplicateSplitVertexIdx.push_back(dupIt->second);
            }
            else
                isVertexDup.push_back(false);

            // early return if duplicate already coded
            if (dupIt != _ovTable.duplicatesMap.end())
            {
                if (processedDupIdx.find(dupIt->second) != processedDupIdx.end())
                {
                    // no need to encode as already processed
                    predictPosition = false;
                    // used by attributes using main index table
                    predictCorner[c] = false;
                }
                else {
                    processedDupIdx.insert(dupIt->second);
                }
            }
        }
        if (predictPosition){
            posEncodeWithPrediction(c);
        }
    }

    if (predictUVs) {
        // reset the vertex marking table
        for (auto& v : MV) v = -1;

        // goes through the corners, for the UVs using main index table V
        for (const auto& c : traversalOrder) {
            const auto doPrediction = predictCorner.size() != 0 ? predictCorner[c] : true;
            if (doPrediction) {
                uvEncodeWithPrediction(c, V);
            }
        }
    }
}

```

FIG. 18

```

void EBReversiEncoder::encodeAttributesWithAuxiliaryIndexTable(void)
{
    const auto& ov = _ovTable;
    const auto& TC = _ovTable.TC;
    const auto& UV = _ovTable.uvcoords;
    const auto altTraversal = (cfg.traversal != EBConfig::Traversal::EB);

    if (ov.OTC.size()) // if hasUV and separate table
    {
        // for alternative traversals
        // create a temp OV table where O/V is set with OTC/TC
        std::vector<int> uvCcStartCorners; // extended set of start corners
        if (altTraversal) {
            _uvCornerTable.O = _ovTable.O; // we will add UV seams cuts on top of this
            _uvCornerTable.V = _ovTable.TC;
            uvCcStartCorners = ccStartCorners;
        }
        // define seams
        for (auto corner : processedCorners)
        {
            const int corners[3] = { corner, ov.n(corner), ov.p(corner) };

            for (int i = 0; i < 3; ++i) {
                const int cur_corner = corners[i];
                const int opp_corner = ov.O[cur_corner];
                // no explicit seams on boundary edges opp < 0,
                // also skip if already set to -3 (i.e. < -2)
                if (opp_corner < 0 || ov.OTC[cur_corner] < -2) {
                    continue;
                }
                const bool isSeam = (ov.OTC[cur_corner] == -2);
                oUVSeams.push_back(isSeam);
                // used to identify seams when predicting uvs
                const int updval = isSeam ? -3 : -4;
                _ovTable.OTC[cur_corner] = updval;
                _ovTable.OTC[opp_corner] = updval;
                // special work for alternative traversals
                if (altTraversal && isSeam) {
                    _uvCornerTable.O[cur_corner] = -3;
                    _uvCornerTable.O[opp_corner] = -3;
                    // we store seam corners as a potential start corners
                    uvCcStartCorners.push_back(cur_corner);
                    uvCcStartCorners.push_back(opp_corner);
                }
            }
        }
        // reset the vertex marking table
        for (auto& v : MV) v = -1;
        // do the traversal and predict
        if (cfg.traversal == EBConfig::Traversal::DEGREE) {
            PredictionVertexTraverser traverser;
            traverser.init(&_uvCornerTable, _ovTable.uvcoords.size(), _ovTable.getTriangleCount());
            for (auto corner : uvCcStartCorners) {
                if (!traverser.isCornerFaceVisited(corner))
                    traverser.traverse(corner);
            }
            for (auto c : traverser.visitedCorners) // compute the predictions
            {
                Write AUX_UV_ENCODE statement here (see Figure 20)
            }
        }
        else { // otherwise use traversal order from topology
            for (auto corner : processedCorners)
            {
                const int corners[3] = { corner, ov.n(corner), ov.p(corner) };
                for (int i = 0; i < 3; ++i) {
                    const int c = corners[i];

                    Write AUX_UV_ENCODE statement here (see Figure 20)
                }
            }
        }
    }
}

```

FIG. 19

```
if (TC[c] < 0)
    continue;
switch (cfg.uvPred) {
case EBConfig::UvPred::NONE:
    if (MV[TC[c]] > 0)
        continue;
    MV[TC[c]] = 1;
    oUVCoords.push_back(UV[TC[c]]);
    break;
case EBConfig::UvPred::STRETCH:
    uvEncodeWithPrediction(c, TC);
    break;
}
```

FIG. 20

```

void EBReversiEncoder::posEncodeWithPrediction(int c) {
    const auto MAX_PARALLELOGRAMS = 4; // max number of parallelograms, can be changed at will
    const auto& ov = _ovTable;
    const auto& V = ov.V;
    const auto& O = ov.O;
    const auto& G = ov.positions;
    const auto& v = ov.v(c);

    // is vertex already predicted ?
    if (MV[v] > 0)
        return;
    // we mark the vertex
    MV[v] = 1;

    // search for some parallelogram estimations around the vertex of the corner
    // the triangle fan might not be complete since we do not use dummy points,
    // but we know that a vertex is not non-manifold, so we have only one fan per vertex
    // also some opposite might not be defined due to boundaries
    // we use OV accessors and test to filter negative values

    vec3 predPos(0, 0, 0); // the predicted position
    int count = 0; // number of valid parallelograms found
    int altC = c;

    // loop through corners attached to the current vertex
    // swing right around the fan
    int nextC = ov.n(0[ov.n(altC)]);
    while (nextC >= 0 && nextC != c)
    {
        altC = nextC;
        nextC = ov.n(0[ov.n(altC)]);
    };
    bool isBoundary = (nextC != c);

    // now we are position on the right most corner sharing v
    // we turn left and evaluate the possible predictions
    const int startC = altC;
    do
    {
        if (count >= MAX_PARALLELOGRAMS) break;

        const auto& oppoV = ov.v(0[altC]);
        const auto& prevV = ov.v(ov.p(altC));
        const auto& nextV = ov.v(ov.n(altC));

        if ((oppoV > -1 && prevV > -1 && nextV > -1) &&
            ((MV[oppoV] > 0) && (MV[prevV] > 0) && (MV[nextV] > 0)))
        {
            // parallelogram prediction estG = prevG + nextG - oppoGd
            vec3 estG = G[prevV] + G[nextV] - G[oppoV];
            predPos += estG; // accumulate parallelogram predictions
            ++count;
        }
        altC = ov.p(0[ov.p(altC)]); // swing around the triangle fan
    } while (altC >= 0 && altC != startC); // incomplete fan or full rotation

    // 1. use parallelogram prediction when possible
    if (count > 0) {
        oVertices.push_back(G[v] - round(predPos / vec3(count))); // round each component
        return;
    }

    // Continued in Figure 22

```

FIG. 21

```
// Continues code from Figure 21

// 2. or fallback to delta with available values
const auto& c_p_v = ov.v(ov.p(c));
const auto& c_n_v = ov.v(ov.n(c));

if (c_p_v > -1 && MV[c_p_v] > -1) {
    oVertices.push_back(G[v] - G[c_p_v]);
    return;
}

if (c_n_v > -1 && MV[c_n_v] > -1) {
    oVertices.push_back(G[v] - G[c_n_v]);
    return;
}

// 3. if we are on a boundary
// then we may use deltas from previous vertex on the boundary
if (isBoundary) {
    const auto b = ov.p(startC); // b is on boundary
    const auto b_v = ov.v(b);
    auto marked = MV[b_v];
    if (marked > -1) {
        oVertices.push_back(G[v] - G[b_v]);
        return;
    }
}

// 4. no other choice
osVertices.push_back(G[v]); // global value (it is a start, pushed in separate table)

return;
}
```

FIG. 22

```

void EBReversiEncoder::uvEncodeWithPrediction(int c, const std::vector<int>& tcIndices)
{
    const auto MAX_STRETCH_PREDS = 8; // Maximum number of stretch predictions
    const auto& ov = _ovTable;
    const auto& O = ov.O;
    const auto& OTC = _ovTable.OTC;
    const auto& G = ov.positions;
    const auto& UV = ov.uvcoords;
    const auto& TC = tcIndices; // texture coordinate index table
    const auto& tc = TC[c]; // texture coordinate index for corner c

    // we mark the vertex
    MV[tc] = 1;

    // search for some stretch estimations around the vertex of the corner
    // the triangle fan might not be complete since we do not use dummy points,
    // but we know that a vertex is not non-manifold, so we have only one fan per vertex
    // also some opposite might not be defined due to boundaries
    // we use OV accessors and test to filter negative values

    Vec2 predUV(0, 0); // the predicted uv coordinate
    int count = 0; // number of valid stretch found
    int altC = c;

    // loop through corners attached to the current vertex
    // swing right around the fan untill full loop, border or UV seam
    bool onSeam = (OTC.size() != 0 ? (OTC[ov.n(altC)] == -3) : false);
    int nextC = ov.n(O[ov.n(altC)]);
    while (nextC >= 0 && nextC != c && !onSeam)
    {
        altC = nextC;
        onSeam = (OTC.size() != 0 ? (OTC[ov.n(altC)] == -3) : false);
        nextC = ov.n(O[ov.n(altC)]);
    };
    bool isBoundary = (!onSeam && nextC != c);

    // now we are position on the right most corner sharing v
    // we turn left an evaluate the possible predictions and stop on first found
    const int startC = altC;
    do
    {
        if (count >= MAX_STRETCH_PREDS) break; // (MAX_STRETCH_PREDS limit reached)

        const auto altV = TC[altC];
        const auto prevV = TC[ov.p(altC)];
        const auto nextV = TC[ov.n(altC)];

        if ((altV > -1 && prevV > -1 && nextV > -1) &&
            ((MV[altV] > 0) && (MV[prevV] > 0) && (MV[nextV] > 0)))
        {
            vec2 estUV(0, 0);
            predictUV(altC, TC, estUV);
            predUV += estUV;
            ++count;
        }
        onSeam = (OTC.size() != 0 ? (OTC[ov.p(altC)] == -3) : false);
        altC = ov.p(O[ov.p(altC)]); // swing around the triangle fan
    } while (altC >= 0 && altC != startC && !onSeam); // incomplete fan or full rotation

    // 1. use stretch prediction when possible
    if (count > 0) {
        predUV = glm::round(predUV / glm::vec2(count));
        oUVCoords.push_back(UV[tc] - predUV);
        return;
    }

    // continued in Figure 24

```

FIG. 23

```
// continues code from Figure 23

// 2. or fallback to delta with available values
const auto& c_p_tc = TC[ov.p(c)];
const auto& c_n_tc = TC[ov.n(c)];

if (c_p_tc > -1 && MV[c_p_tc] > -1) {
    oUVCoords.push_back(UV[tc] - UV[c_p_tc]);
    return;
}

if (c_n_tc > -1 && MV[c_n_tc] > -1) {
    oUVCoords.push_back(UV[tc] - UV[c_n_tc]);
    return;
}

// 3. if we are on a boundary
// then we may use deltas from previous vertex on the boundary
if (isBoundary) {
    const auto b = ov.p(startC); // b is on boundary
    const auto b_tc = TC[b];
    if (MV[b_tc] > -1) {
        oUVCoords.push_back(UV[tc] - UV[b_tc]);
        return;
    }
}

// 4. no other choice, it is a global start, pushed in separate table
osUVCoords.push_back(UV[tc]);

return;
}
```

FIG. 24

```

void EBReversiEncoder::predictUV(
    const int c, const std::vector<int>& tcIndices,
    glm::vec2& resUV)
{
    const auto& OV = _ovTable;
    const auto& O = _ovTable.O;
    const auto& OTC = _ovTable.OTC;
    const auto& V = _ovTable.V;
    const auto& G = _ovTable.positions;
    const auto& UV = _ovTable.uvcoords;
    const auto& TC = tcIndices;    // texture coordinates indices

    // we do not accumulate uv predictions, stop on first available
    glm::dvec2 uvPrev = UV[TC[OV.p(c)]];
    glm::dvec2 uvNext = UV[TC[OV.n(c)]];
    glm::dvec2 uvCurr = UV[TC[c]];
    glm::dvec3 gPrev = G[V[OV.p(c)]];
    glm::dvec3 gNext = G[V[OV.n(c)]];
    glm::dvec3 gCurr = G[V[c]];
    glm::dvec3 gNgP = gPrev - gNext;
    glm::dvec3 gNgC = gCurr - gNext;
    glm::dvec2 uvNuvP = uvPrev - uvNext;
    double gNgP_dot_gNgC = glm::dot(gNgP, gNgC);
    double d2_gNgP = glm::dot(gNgP, gNgP);
    //
    if (d2_gNgP > 0)
    {
        glm::dvec2 uvProj = uvNext + uvNuvP * (gNgP_dot_gNgC / d2_gNgP);
        glm::dvec3 gProj = gNext + gNgP * (gNgP_dot_gNgC / d2_gNgP);
        double d2_gProj_gCurr = glm::dot(gCurr - gProj, gCurr - gProj);
        const glm::dvec2 uvProjuvCurr = glm::dvec2(uvNuvP.y, -uvNuvP.x)
            * std::sqrt(d2_gProj_gCurr / d2_gNgP);
        glm::dvec2 predUV0(uvProj + uvProjuvCurr);
        glm::dvec2 predUV1(uvProj - uvProjuvCurr);

        const bool orientation = length(uvCurr - predUV0) < length(uvCurr - predUV1);
        resUV = round(orientation ? predUV0 : predUV1);
        oOrientations.push_back(orientation); // stores the orientation
    }
    // else average the two predictions
    else
    {
        resUV = round((UV[TC[OV.n(c)]] + UV[TC[OV.p(c)]] / 2.0f);
    }
}

```

FIG. 25

```

void EBReversiDecoder::decodeAttributesWithMainIndexTable(void)
{
    if (cfg.traversal == EBConfig::Traversal::DEGREE) {
        // computes a corner order using prediction degree traversal
        PredictionVertexTraverser traverser;
        traverser.init(&_ovTable, _ovTable.getPositionCount(), _ovTable.getTriangleCount());
        for (auto corner : ccStartCorners)
            traverser.traverse(corner);

        decodeAttributesWithPredition(traverser.visitedCorners, false);
    }
    else { // otherwise uses the traversal order from topology encoding
        decodeAttributesWithPredition(readCorners, true);
    }
}

```

FIG. 26

```

void EBReversiDecoder::decodeAttributesWithPrediction( std::vector<int>& traversalOrder, bool reverse )
{
    const auto& ov = _ovTable;
    const auto& V = ov.V;
    const auto& OTC = ov.OTC;
    auto& G = _ovTable.positions;
    auto& UV = _ovTable.uvcoords;
    // predict UVs in first pass if no separate index
    const bool predictUVs = (UV.size() && !OTC.size());
    // go through traversal in forward or reverse order
    const int start = reverse ? traversalOrder.size() - 1 : 0;
    int inc = reverse ? -1 : 1;

    // prepare the skip table if deduplication is active
    std::vector<int> skippedCorners;
    if (cfg.deduplicate) {
        skippedCorners.assign(ov.V.size(), -1);
    }

    // reset the vertex marking table
    for (auto& v : MV) v = -1;

    // goes through the corners, for the positions
    for (int i = start; i >= 0 && i < traversalOrder.size(); i += inc) {
        const auto& c = traversalOrder[i];
        bool predictPosition = true;
        if (cfg.deduplicate && isVertexDup.size())
        {
            if (isVertexDup[Vcur++])
            {
                auto splitIdx = iDuplicateSplitVertexIdx[Dcur++];
                auto dupIt = processedDupIdx.find(splitIdx);
                if (dupIt != processedDupIdx.end())
                {
                    duplicates.push_back(std::make_pair(c, dupIt->second));
                    G[V[c]] = _ovTable.positions[dupIt->second];
                    predictPosition = false;
                    // used by attributes using main index table
                    skippedCorners[c] = dupIt->second;
                }
                else
                {
                    processedDupIdx[splitIdx] = V[c];
                }
            }
        }
        if (predictPosition) {
            posDecodeWithPrediction(c);
        }
    }
    if (predictUVs) {
        // reset the vertex marking table
        for (auto& v : MV) v = -1;

        // goes through the corners, for the UVs using main index table V
        for (int i = start; i >= 0 && i < traversalOrder.size(); i += inc) {
            const auto& c = traversalOrder[i];
            const auto skipIdx = skippedCorners.size() != 0 ? skippedCorners[c] : -1;
            if (skipIdx < 0) {
                uvDecodeWithPrediction(c, V);
            } else {
                UV[V[c]] = _ovTable.uvcoords[skipIdx];
            }
        }
    }
}

```

FIG. 27

```

void EBReversiDecoder::decodeAttributesWithAuxiliaryIndexTable(void) {

    const auto& ov = _ovTable;
    auto& G = _ovTable.positions;
    auto& O = _ovTable.O;
    auto& V = _ovTable.V;
    auto& OTC = _ovTable.OTC;
    auto& TC = _ovTable.TC;
    auto& UV = _ovTable.uvcoords;
    const auto altTraversal = (cfg.traversal != EBConfig::Traversal::EB);

    // create a temp OV table where O/V is set with OTC/TC
    CTMesh tmpOV;
    std::vector<int> uvCcStartCorners;
    if (altTraversal) {
        _uvCornerTable.O = _ovTable.O;           // we will add UV seams cuts on top of this
        uvCcStartCorners = ccStartCorners;        // extended set of start corners
    }
    // use OTC to describe seams
    for (int i = processedCorners.size() - 1; i >= 0; --i)
    {
        const auto& corner = processedCorners[i];
        const int corners[3] = { corner, ov.n(corner), ov.p(corner) };
        const int src_face_id = ov.t(corner);
        for (int c = 0; c < 3; ++c) {
            const int cur_corner = corners[c];
            const int opp_corner = O[cur_corner];
            // no explicit seams on boundary edges (keep OTC to -3)
            // or skip since already done (e.g. -2 or -1)
            if (opp_corner < 0)
                continue;
            const int opp_face_id = ov.t(opp_corner);
            // decode only one side
            if (opp_face_id < src_face_id)
                continue;
            // use pre evaluated seam code
            const auto isSeam = iUVSeams[uvSeamsIndex++];
            const auto fval = (isSeam) ? -2 : -1;
            OTC[cur_corner] = fval;
            OTC[opp_corner] = fval;
            //
            if (altTraversal && isSeam) {
                tmpOV.O[cur_corner] = -2;
                tmpOV.O[opp_corner] = -2;
                // we store seam corners as a potential start corners
                uvCcStartCorners.push_back(cur_corner);
                uvCcStartCorners.push_back(opp_corner);
            }
        }
    }

    // Continued in Figure 29

```

FIG. 28

```
// continues code from Figure 28

// regenerate UV indices (TC) using seams information from OTC
int NTC = 0;
for (int i = processedCorners.size() - 1; i >= 0; --i) {
    const auto& corner = processedCorners[i];
    const int corners[3] = { corner, ov.n(corner), ov.p(corner) };
    for (int c = 0; c < 3; ++c) {
        const int cur_corner = corners[c];
        if (TC[cur_corner] >= 0) // index already allocated
            continue;
        const auto newTC = NTC++; // inc after assign
        TC[cur_corner] = newTC;
        // swing to right most extent (geo border -3 or uv seam -2)
        auto movC = cur_corner;
        while (O[ov.n(movC)] >= 0 && OTC[ov.n(movC)] == -1)
        {
            movC = ov.n(O[ov.n(movC)]);
            TC[movC] = newTC;
            if (movC == cur_corner)
                break;
        }
        // then the other side (geo border or uv seam)
        movC = cur_corner;
        while (O[ov.p(movC)] >= 0 && OTC[ov.p(movC)] == -1)
        {
            movC = ov.p(O[ov.p(movC)]);
            TC[movC] = newTC;
            if (movC == cur_corner)
                break;
        }
    }
}

// continued in Figure 30
```

FIG. 29

```

// continues the code from Figure 29

// reset the vertex marking table
for (auto& v : MV) v = -1;

if (cfg.traversal == EBConfig::Traversal::DEGREE) {
    _uvCornerTable.V = _ovTable.TC; // use reconstructed TC
    PredictionVertexTraverser traverser;
    traverser.init(&_amp;_uvCornerTable, ov.uvcoords.size(), ov.getTriangleCount());
    for (auto corner : uvCcStartCorners) {
        if (!traverser.isCornerFaceVisited(corner))
            traverser.traverse(corner);
    }
    //
    for (auto cur_corner : traverser.visitedCorners)
    {
        AUX_UV_DECODE
    }
}
else { // otherwise use traversal order from topology
    for (int i = processedCorners.size() - 1; i >= 0; --i)
    {
        const auto& corner = processedCorners[i];
        const int corners[3] = { corner, ov.n(corner), ov.p(corner) };
        for (int c = 0; c < 3; ++c) {
            const int cur_corner = corners[c];
            AUX_UV_DECODE
        }
    }
}
}

```

FIG. 30

```

if (TC[cur_corner] < 0)
    continue;
switch (cfg.uvPred) {
case EBConfig::UvPred::NONE:
    if (MV[TC[cur_corner]] > 0)
        continue;
    MV[TC[cur_corner]] = 1;
    UV[TC[cur_corner]] = readUvDelta();
    break;
case EBConfig::UvPred::STRETCH:
    uvDecodeWithPrediction(cur_corner, TC);
    break;
}

```

FIG. 31

```

void EBReversiDecoder::posDecodeWithPrediction(int c) {
    const auto MAX_PARALLELOGRAMS = 4;
    const auto& ov = _ovTable;
    const auto& V = ov.V;
    const auto& O = ov.O;
    const auto& iG = iVertices;
    auto& isG = isVertices;
    auto& G = _ovTable.positions;
    const auto& v = ov.v(c);

    // is vertex already predicted?
    if (MV[v] > 0)
        return;
    // we mark the vertex
    MV[v] = 1;

    // search for some parallelogram estimations around the vertex of the corner.
    // the triangle fan might not be complete since we do not use dummy points,
    // but we know that a vertex is not non-manifold, so we have only one fan per vertex
    // also some opposite might not be defined due to boundaries
    // we use OV accessors and test to filter negative values

    vec3 predPos(0, 0, 0);          // the predicted position
    int count = 0;                  // number of valid parallelograms found
    int altC = c;

    // loop through corners attached to the current vertex
    // swing around the fan until we find a border
    int nextC = ov.n(O[ov.n(altC)]);
    while (nextC >= 0 && nextC != c)
    {
        altC = nextC;
        nextC = ov.n(O[ov.n(altC)]);
    };
    bool isBoundary = (nextC != c);

    // now we are position on the right most corner sharing v
    // we turn left and evaluate the possible predictions
    int startC = altC;
    do
    {
        if (count >= MAX_PARALLELOGRAMS) break;

        const auto& oppoV = ov.v(O[altC]);
        const auto& prevV = ov.v(ov.p(altC));
        const auto& nextV = ov.v(ov.n(altC));

        if ((oppoV > -1 && prevV > -1 && nextV > -1) &&
            ((MV[oppoV] > 0) && (MV[prevV] > 0) && (MV[nextV] > 0)))
        {
            // parallelogram prediction estG = prevG + nextG - oppoGd
            vec3 estG = G[prevV] + G[nextV] - G[oppoV];
            predPos += estG;          // accumulate parallelogram predictions
            ++count;
        }
        altC = ov.p(O[ov.p(altC)]); // swing around the triangle fan
    } while (altC >= 0 && altC != startC); // incomplete fan or full rotation

    // 1. use parallelogram prediction when possible
    if (count > 0) {
        G[v] = readDelta() + round(predPos / vec3(count)); // round each component
        return;
    }

    // continued in Figure 33

```

FIG. 32

```
// Continues code from Figure 32

// 2. or fallback to delta with available values
const auto& c_p_v = ov.v(ov.p(c));
const auto& c_n_v = ov.v(ov.n(c));

if (c_p_v > -1 && MV[c_p_v] > -1) {
    G[v] = readDelta() + G[c_p_v];
    return;
}

if (c_n_v > -1 && MV[c_n_v] > -1) {
    G[v] = readDelta() + G[c_n_v];
    return;
}

// 3. or maybe we are on a boundary
// then we may use deltas from previous vertex on the boundary
if (isBoundary) {
    const auto b = ov.p(startC); // b is on boundary
    const auto b_v = ov.v(b);
    auto marked = MV[b_v];
    if (marked > -1) {
        G[v] = readDelta() + G[b_v];
        return;
    }
}

// 4. no more choices
G[v] = readStart();

return;
}
```

FIG. 33

```

void EBReversiDecoder::uvDecodeWithPrediction(int c, const std::vector<int>& tcIndices)
{
    const auto MAX_STRETCH_PREDS = 8; // max predictions can be modified
    auto& ov = _ovTable;
    const auto& O = ov.0;
    const auto& OTC = _ovTable.OTC;
    auto& UV = ov.uvcoords;
    const auto& TC = tcIndices;
    const auto& tc = TC[c];
    // is vertex already predicted ?
    if (MV[tc] > 0)
        return;
    // we mark the vertex
    MV[tc] = 1;
    // search for valid min stretch estimations around the vertex of the corner
    // the triangle fan might not be complete since we do not use dummy points,
    // but we know that a vertex is not non-manifold, so we have only one fan per vertex
    // also some opposite might not be defined due to boundaries
    // we use OV accessors and test to filter negative values
    glm::vec2 predUV (0, 0); // the predicted uv
    int altC = c;
    // loop through corners attached to the current vertex
    // swing around the fan until we find a border
    bool onSeam = (OTC.size() != 0 ? (OTC[ov.n(altC)] == -2) : false);
    int nextC = ov.n(O[ov.n(altC)]);
    while (nextC >= 0 && nextC != c && !onSeam)
    {
        altC = nextC;
        onSeam = (OTC.size() != 0 ? (OTC[ov.n(altC)] == -2) : false);
        nextC = ov.n(O[ov.n(altC)]);
    };
    bool isBoundary = (!onSeam && nextC != c);
    // now we are position on the right most corner sharing v
    // we turn left and evaluate the possible predictions
    int startC = altC;
    int count = 0; // number of valid stretch found
    do
    {
        if (count >= MAX_STRETCH_PREDS) break; // we reached max predictions
        const auto altV = TC[altC];
        const auto prevV = TC[ov.p(altC)];
        const auto nextV = TC[ov.n(altC)];
        if ((altV > -1 && prevV > -1 && nextV > -1) &&
            ((MV[altV] > 0) && (MV[prevV] > 0) && (MV[nextV] > 0)))
        {
            glm::vec2 estUV(0, 0);
            predictUV(altC, TC, estUV);
            predUV += estUV;
            ++count;
        }
        onSeam = (OTC.size() != 0 ? (OTC[ov.p(altC)] == -2) : false);
        altC = ov.p(O[ov.p(altC)]); // swing around the triangle fan
    } while (altC >= 0 && altC != startC && !onSeam); // incomplete fan or full rotation

    // 1. use min stretch prediction when possible
    if (count > 0) {
        predUV = glm::round(predUV / glm::vec2(count));
        UV[tc] = predUV + readUvDelta();

        return;
    }
    // Code continues in Figure 35
}

```

FIG. 34

```
// Continues code from Figure 34

// 2. or fallback to delta with available values
const auto& c_p_tc = TC[ov.p(c)];
const auto& c_n_tc = TC[ov.n(c)];

if (c_p_tc > -1 && MV[c_p_tc] > -1) {
    UV[tc] = readUvDelta() + UV[c_p_tc];
    return;
}

if (c_n_tc > -1 && MV[c_n_tc] > -1) {
    UV[tc] = readUvDelta() + UV[c_n_tc];
    return;
}

// 3. or maybe we are on a boundary
// then we may use deltas from previous vertex on the boundary
if (isBoundary) {
    const auto b = ov.p(startC); // b is on boundary
    const auto b_tc = TC[b];
    if (MV[b_tc] > -1) {
        UV[tc] = readUvDelta() + UV[b_tc];
        return;
    }
}

// 4. no more choices, it is a start
UV[tc] = readUvStart();

return;
}
```

FIG. 35

```

void EBReversiDecoder::predictUV( // directly stores the result value in the UV table
    const int c,
    const std::vector<int>& tcIndices,
    vec2& resUV
){
    // aliases
    auto& OV = _ovTable;
    auto& G = _ovTable.positions;
    auto& O = _ovTable.O;
    const auto& OTC = _ovTable.OTC;
    auto& V = _ovTable.V;
    auto& UV = _ovTable.uvcoords;
    const auto& TC = tcIndices; // texture coordinate indices
    // accumulate uv predictions
    const glm::dvec2 uvPrev = UV[TC[OV.p(c)]];
    const glm::dvec2 uvNext = UV[TC[OV.n(c)]];
    const glm::dvec3 gPrev = G[V[OV.p(c)]];
    const glm::dvec3 gNext = G[V[OV.n(c)]];
    const glm::dvec3 gCurr = G[V[c]];
    const glm::dvec3 gNgP = gPrev - gNext;
    const glm::dvec3 gNgC = gCurr - gNext;
    const glm::dvec2 uvNuvP = uvPrev - uvNext;
    const double gNgP_dot_gNgC = glm::dot(gNgP, gNgC);
    const double d2_gNgP = glm::dot(gNgP, gNgP);
    if (d2_gNgP > 0)
    {
        const glm::dvec2 uvProj = uvNext + uvNuvP * (gNgP_dot_gNgC / d2_gNgP);
        const glm::dvec3 gProj = gNext + gNgP * (gNgP_dot_gNgC / d2_gNgP);
        const double d2_gProj_gCurr = glm::dot(gCurr - gProj, gCurr - gProj);
        const glm::dvec2 uvProjuvCurr =
            glm::dvec2(uvNuvP.y, -uvNuvP.x) * std::sqrt(d2_gProj_gCurr / d2_gNgP);
        const glm::dvec2 predUV0(uvProj + uvProjuvCurr);
        const glm::dvec2 predUV1(uvProj - uvProjuvCurr);
        const glm::dvec2 predUVd = readOrientation() ? predUV0 : predUV1;
        resUV = glm::vec2(std::round(predUVd.x), std::round(predUVd.y));
    }
    // else average the two predictions
    else
    {
        resUV = round((UV[TC[OV.n(c)]] + UV[TC[OV.p(c)]] / 2.0f);
    }
}

```

FIG. 36

```

void EBReversiEncoder::encodePerFaceAttributes(void)
{
    const auto& ov = _ovTable;
    if (ov.ids.size()) // has face ids
    {
        for (auto corner : processedCorners)
        {
            const int corners[3] = { corner, ov.n(corner), ov.p(corner) };

            const int tri = ov.t(corner);
            auto curFaceID = ov.ids[tri];
            // store a boolean values checking for equality between successive values
            oFidsIdIsDifferent.push_back(curFaceID == prevFaceId);
            // if the id associated to the current face is equal
            // to the previously encoded value, nothing to do
            if (curFaceID != prevFaceId)
            {
                // check if the 3 adjacent faces (through opposites) have already been encoded
                // may be on a boundary, in this case the 0 is negative and t(0[]) too, hence the test
                const auto tFIdx = ov.t(ov.O[corners[0]]);
                const auto tRIdx = ov.t(ov.O[corners[1]]);
                const auto tLIdx = ov.t(ov.O[corners[2]]);
                bool decodedFacing = tFIdx < 0 ? false : UF[tFIdx];
                bool decodedRight = tRIdx < 0 ? false : UF[tRIdx];
                bool decodedLeft = tLIdx < 0 ? false : UF[tLIdx];
                // if already encoded, retrieve corresponding ids
                int faceIdFacing = decodedFacing ? ov.ids[ov.t(ov.O[corners[0]])] : -1;
                int faceIdRight = decodedRight ? ov.ids[ov.t(ov.O[corners[1]])] : -1;
                int faceIdLeft = decodedLeft ? ov.ids[ov.t(ov.O[corners[2]])] : -1;
                // starting from each of the adjacent triangles
                // (sharing one edge with the triangle we are encoding)
                for (auto k = 0; k < 3; k++)
                {
                    // we will potentially update the "decoded" (decodable) status and related id
                    // for each of the 3 reference triangles
                    auto dec = (k == 0) ? decodedRight : ((k == 1) ? decodedLeft : decodedFacing);
                    if (!dec)
                    {
                        // initialise a starting "rotating corner" as a corner of the non encoded
                        // adjacent triangle (R for k==0, L if k==1, F if k==2)
                        // rc is negative if we reach a boundary, we use accessor ov.o() that tests
                        // range and returns 0 if so
                        auto rc = ov.o(corners[(k + 1) % 3]);
                        // then start looping through adjacent triangles, swinging around a first
                        // corner shared with the triangle we are encoding
                        rc = ov.o(ov.n(rc));
                        // stopping if an encoded triangle is found, or if the analyzed face is one of
                        // the 3 original R,L,F triangles
                        // rc is negative if we reach a boundary
                        while (rc >= 0 && rc != ov.p(ov.O[corners[(k) % 3]]) && !UF[ov.t(rc)])
                        {
                            rc = ov.O[ov.n(rc)];
                        }
                        if (rc < 0 || !UF[ov.t(rc)])
                        {
                            // if no encoded face found, then swing around the other shared corners
                            // rc is negative if we reach a boundary, we use accessor ov.o() that tests
                            // range and returns 0 if so
                            rc = ov.o(corners[(k + 1) % 3]);
                            rc = ov.o(ov.p(rc));
                            while (rc >= 0 && rc != ov.n(ov.O[corners[(k + 2) % 3]]) && !UF[ov.t(rc)])
                            {
                                rc = ov.O[ov.p(rc)];
                            }
                        }
                    }
                }
            }
        }
    }
    // continues in Figure 38

```

FIG. 37

```

// Continues from Figure 37

    auto newDecoded = -1; // negative if no new decoded was found
    if (rc >= 0 && UF[ov.t(rc)]) {
        newDecoded = ov.ids[ov.t(rc)];
    }
    // if an already encoded face is found during the procedure, then update
    // the corresponding "decoded" (can be decoded) status, and Id value
    if (newDecoded >= 0) {
        if (k == 0) {
            decodedRight = true;
            faceIdRight = newDecoded;
        }
        else if (k == 1) {
            decodedLeft = true;
            faceIdLeft = newDecoded;
        }
        else
        {
            decodedFacing = true;
            faceIdFacing = newDecoded;
        }
    }
}

}

if (decodedRight){
    oFaceIdIsRight.push_back(faceIdRight == curFaceID);
}
if (!decodedRight || (faceIdRight != curFaceID))
{
    if ((decodedLeft) && (faceIdLeft != faceIdRight)){
        oFaceIdIsLeft.push_back(faceIdLeft == curFaceID);
    }
    if (!decodedLeft || (faceIdLeft != curFaceID))
    {
        if ((decodedFacing) && (faceIdFacing != faceIdRight)
            && (faceIdFacing != faceIdLeft))
        {
            oFaceIdIsFacing.push_back(faceIdFacing == curFaceID);
        }
        if (!decodedFacing || (faceIdFacing != curFaceID)){
            oNotPredictedFaceId.push_back(curFaceID);
        }
    }
}

}
UF[tri] = 1; // mark the face as encoded
prevFaceId = curFaceID;
}
}
}

```

FIG. 38

```

void EBReversiDecoder::decodePerFaceAttributes(void)
{
    const auto& ov = _ovTable;
    auto& O = _ovTable.O;
    auto& ID = _ovTable.ids;

    int prevFid = -1;

    for (int i = processedCorners.size() - 1; i >= 0; --i)
    {
        const auto& corner = processedCorners[i];
        const int corners[3] = { corner, ov.n(corner), ov.p(corner) };
        const int tri = ov.t(corner);

        const bool isSame = iFidsIdIsDifferent[curFidIsDifferent++];
        if (isSame)
        {
            _ovTable.ids[tri] = prevFid;
            UF[tri] = 1;
            continue;
        }

        // use _UF and shift
        // may be on a boundary, in this case the 0 is negative and t[0[]] to, hence the test
        const auto tFIdx = ov.t(ov.O[corners[0]]);
        const auto tRIdx = ov.t(ov.O[corners[1]]);
        const auto tLIdx = ov.t(ov.O[corners[2]]);
        bool decodedFacing = tFIdx < 0 ? false : UF[tFIdx];
        bool decodedRight = tRIdx < 0 ? false : UF[tRIdx];
        bool decodedLeft = tLIdx < 0 ? false : UF[tLIdx];
        int faceIdFacing = decodedFacing ? ID[ov.t(O[corners[0]])] : -1;
        int faceIdRight = decodedRight ? ID[ov.t(O[corners[1]])] : -1;
        int faceIdLeft = decodedLeft ? ID[ov.t(O[corners[2]])] : -1;

        for (auto k = 0; k < 3; k++)
        {
            const auto dec = (k == 0) ? decodedRight : ((k == 1) ? decodedLeft : decodedFacing);
            if (!dec)
            {
                // initialise a starting "rotating corner" as a corner of the non encoded
                // adjacent triangle (R for k==0, L if k==1, F if k==2)
                auto rc = ov.o(corners[(k + 1) % 3]);
                // then start looping through adjacent triangles, swinging around a first
                // corner shared with the triangle we are encoding
                rc = ov.o(ov.n(rc));
                // stopping if an encoded triangle is found, or if the analyzed face is one of
                // the 3 original R,L,F triangles
                while (rc >= 0 && rc != ov.p(O[corners[(k) % 3]]) && !UF[ov.t(rc)])
                {
                    rc = O[ov.n(rc)];
                }
                if (rc < 0 || !UF[ov.t(rc)])
                {
                    // if no encoded face found, then swing around the other shared corners
                    rc = ov.o(corners[(k + 1) % 3]);
                    rc = ov.o(ov.p(rc));
                    while (rc >= 0 && rc != ov.n(O[corners[(k + 2) % 3]]) && !UF[ov.t(rc)])
                    {
                        rc = O[ov.p(rc)];
                    }
                }
            }
        }
    }

    // continues in Figure 40

```

FIG. 39

```

// continues from Figure 39

    auto newDecoded = -1; // negative if no new decoded was found
    if (rc >= 0 && UF[ov.t(rc)]) {
        newDecoded = ov.ids[ov.t(rc)];
    }
    // if an already encoded face is found during the procedure, then update
    // the corresponding "decoded" (can be decoded) status, and Id value
    if (newDecoded >= 0) {
        if (k == 0) {
            decodedRight = true;
            faceIdRight = newDecoded;
        }
        else if (k == 1) {
            decodedLeft = true;
            faceIdLeft = newDecoded;
        }
        else
        {
            decodedFacing = true;
            faceIdFacing = newDecoded;
        }
    }
}

UF[tri] = 1;
int fid = -1;
if ((fid < 0) && decodedRight)
{
    if (iFaceIdIsRight[curFidIsRight++){
        fid = faceIdRight;
    }
}
if ((fid < 0) && decodedLeft && (faceIdLeft != faceIdRight))
{
    if (iFaceIdIsLeft[curFidIsLeft++){
        fid = faceIdLeft;
    }
}
if ((fid < 0) && decodedFacing && (faceIdFacing != faceIdRight) && (faceIdFacing != faceIdLeft))
{
    if (iFaceIdIsFacing[curFidIsFacing++){
        fid = faceIdFacing;
    }
}
if (fid < 0){
    fid = iNotPredictedFaceId[curFidNotPredicted++];
}
_ovTable.ids[tri] = fid;
UF[tri] = 1;
prevFid = fid;
}
}

```

FIG. 40

```

void EBReversiDecoder::applyIndexDeduplicatePerFan(void) {

    const auto& ov = _ovTable;
    auto& V = _ovTable.V;
    auto& O = _ovTable.O;

    for (auto& dup : duplicates)
    {
        auto curCorner = dup.first;
        auto splitVtxToVertex = dup.second;

        V[curCorner] = splitVtxToVertex;
        auto next_c = curCorner;
        while (next_c = _ovTable.n(O[_ovTable.n(next_c)]),
            next_c != curCorner && next_c >= 0) {
            V[next_c] = splitVtxToVertex;
        }
        if (next_c < 0)
        {
            next_c = curCorner;
            while (next_c = _ovTable.p(O[_ovTable.p(next_c)]),
                next_c != curCorner && next_c >= 0) {
                V[next_c] = splitVtxToVertex;
            }
        }
    }
}

```

FIG. 41

```

const auto NumHandles = oHandles.size() / 2;
mpcp.getMeshCcWithHandlesCount() = NumHandles;
mpcp.getMeshHandleIndexFirstDelta().resize(NumHandles);
mpcp.getMeshHandleIndexSecondDelta().resize(NumHandles);
for (auto i = 0; i < NumHandles; i++) {
    mpcp.getMeshHandleIndexFirstDelta()[i] = (int)oHandles[2 * i + 0] - (i ? oHandles[2 * i - 2] : 0);
    mpcp.getMeshHandleIndexSecondDelta()[i] = (int)oHandles[2 * i + 1] - (i ? oHandles[2 * i - 1] : 0);
}

```

FIG. 42

```

iHandles.resize(NumHandles*2);
uint32_t curHandle1 = 0;
uint32_t curHandle2 = 0;
mpcp.getMeshHandleIndexFirstDelta().resize(NumHandles);
mpcp.getMeshHandleIndexSecondDelta().resize(NumHandles);
if (NumHandles)
{
    for (auto i = 0; i < NumHandles ; i++) {
        iHandles[2 * i + 0] =
            mpcp.getMeshHandleIndexFirstDelta()[i] + (i ? iHandles[2 * i - 2] : 0);
        iHandles[2 * i + 1] =
            mpcp.getMeshHandleIndexSecondDelta()[i] + (i ? iHandles[2 * i - 1] : 0);
    }
}

```

FIG. 43

```

// ...
auto NumPositionStart = 0;
auto NumPositionStart = osVertices.size();
auto& mpdi = mpcp.getMeshPositionDeduplicateInformation();

if (mpep.getMeshPositionDeduplicateMethod() ==
    MeshPositionDeduplicateMethod::MESH_POSITION_DEDUP_DEFAULT) {
    mpdi.getMeshPositionDeduplicateCount() = oDuplicateSplitVertexIdx.size();
    unsigned int NumSplitVertex = 0;
    if (mpdi.getMeshPositionDeduplicateCount() > 0)
    {
        mpdi.getMeshPositionDeduplicateIdx().resize(mpdi.getMeshPositionDeduplicateCount());
        for (auto i = 0; i < mpdi.getMeshPositionDeduplicateCount(); ++i) {
            mpdi.getMeshPositionDeduplicateIdx()[i] = oDuplicateSplitVertexIdx[i];
            NumSplitVertex = std::max(NumSplitVertex, mpdi.getMeshPositionDeduplicateIdx()[i] + 1);
        }
        mpdi.getMeshPositionDeduplicateStartPositions() = osVertices.size();
NumPositionStart = mpdi.getMeshPositionDeduplicateStartPositions();
auto NumAddedDuplicatedVertex = mpdi.getMeshPositionDeduplicateCount() - NumSplitVertex;
auto NumPositionIsDuplicateFlag = isVertexDup.size();
auto NumVerticesCheck = mpcp.getMeshVertexCount();
+ mpdi.getMeshPositionDeduplicateStartPositions();
+ mpcp.getMeshVirtualVertexCount();
+ NumAddedDuplicatedVertex;
        mpdi.getMeshPositionIsDuplicateSize() = meshPositionIsDuplicateFlagBuffer.size();
        mpdi.getMeshPositionIsDuplicateFlag() = meshPositionIsDuplicateFlagBuffer;
    }
else
NumPositionStart = osVertices.size();
}
else
NumPositionStart = osVertices.size();

NumPositionStart = osVertices.size();
mpcp.getMeshPositionStartCount() = NumPositionStart;
mpcp.getMeshPositionStart().resize(NumPositionStart);
for (auto i = 0; i < NumPositionStart; i++) {
    mpcp.getMeshPositionStart()[i].resize(3);
    for (auto j = 0; j < 3; j++) {
        mpcp.getMeshPositionStart()[i][j] = osVertices[i][j];
    }
}
auto NumPredictedPositions = mpcp.getMeshVertexCount() - mpcp.getMeshCcCount();
auto NumPredictedPositions = mpcp.getMeshVertexCount() - NumPositionStart;

// continues ...

```

FIG. 44

```

// ...
numOfTriangles = mpcp.getMeshClerCount() + mpcp.getMeshCcCount();
nT = mpcp.getMeshTriangleCount();

numSplitVert = 0;
numAddedDupVert = 0;
auto& mpdi = mpcp.getMeshPositionDeduplicateInformation();

// TODO - rewrite using globals from parsing to avoid code replication
if (mpep.getMeshPositionDeduplicateMethod() ==
    MeshPositionDeduplicateMethod::MESH_POSITION_DEDUP_DEFAULT) {
    iDuplicateSplitVertexIdx.resize(mpdi.getMeshPositionDeduplicateCount());
    unsigned int NumSplitVertex = 0;
    if (mpdi.getMeshPositionDeduplicateCount() > 0)
    {
        for (auto i = 0; i < mpdi.getMeshPositionDeduplicateCount(); ++i) {
            iDuplicateSplitVertexIdx[i] = mpdi.getMeshPositionDeduplicateIdx()[i];
            NumSplitVertex = std::max(NumSplitVertex, mpdi.getMeshPositionDeduplicateIdx()[i] + 1);
        }
NumPositionStart = mpdi.getMeshPositionDeduplicateStartPositions();
        auto NumAddedDuplicatedVertex = mpdi.getMeshPositionDeduplicateCount() - NumSplitVertex;
NumVertices = mpcp.getMeshVertexCount();
+ mpdi.getMeshPositionDeduplicateStartPositions();
+ NumAddedDuplicatedVertex;
        NumPositionIsDuplicateFlag = mpcp.getMeshVertexCount()
            + mpdi.getMeshPositionDeduplicateStartPositions()
            + NumAddedDuplicatedVertex;
        numSplitVert = NumSplitVertex;
numAddedDupVert = NumAddedDuplicatedVertex;
        numAddedDupVert = mpdi.getMeshPositionDeduplicateCount() - NumSplitVertex;
    }
else
NumPositionStart = mpcp.getMeshCcCount();
}
else
NumPositionStart = mpcp.getMeshCcCount();

NumPositionStart = mpcp.getMeshPositionStartCount();
isVertices.resize(NumPositionStart);
for (auto i = 0; i < NumPositionStart; i++) {
    for (auto j = 0; j < 3; j++) {
        isVertices[i][j] = mpcp.getMeshPositionStart()[i][j];
    }
}

NumPredictedPositions = mpcp.getMeshVertexCount() - mpcp.getMeshCcCount();
NumPredictedPositions = mpcp.getMeshVertexCount() - NumPositionStart;

// continues ...

```

FIG. 45

```
// ...  
if (sVertices.size()) // mpara case with separate table  
numOfVertices = iVertices.size() + sVertices.size() + iDummyVertices.size() + numAddedDupVert;  
else  
numOfVertices = iVertices.size() + numAddedDupVert;  
  
nV = iVertices.size() + isVertices.size() + numAddedDupVert; // use known added vert count  
const auto recUv = iUVCoords.size() + isUVCoords.size();  
nUV = hasSeparateUvIndex ? recUv : (recUv ? nV : 0);  
  
// Arithmetic decoding of deduplication info  
if (NumVertices)  
if (NumPositionIsDuplicateFlag)  
{  
    isVertexDup.resize(NumVertices);  
  
    AdaptiveBitModel ctx;  
    EntropyDecoder    ad;  
    const auto* const  bufferPtr =  
        reinterpret_cast<const char*>(&mpdi.getMeshPositionIsDuplicateFlag()[0]);  
    ad.setBuffer(mpdi.getMeshPositionIsDuplicateSize(), bufferPtr);  
    ad.start();  
for (uint32_t i = 0; i < NumVertices; i++)  
    for (uint32_t i = 0; i < NumPositionIsDuplicateFlag; i++)  
        isVertexDup[i] = ad.decode(ctx);  
}  
  
// continues ...
```

FIG. 46

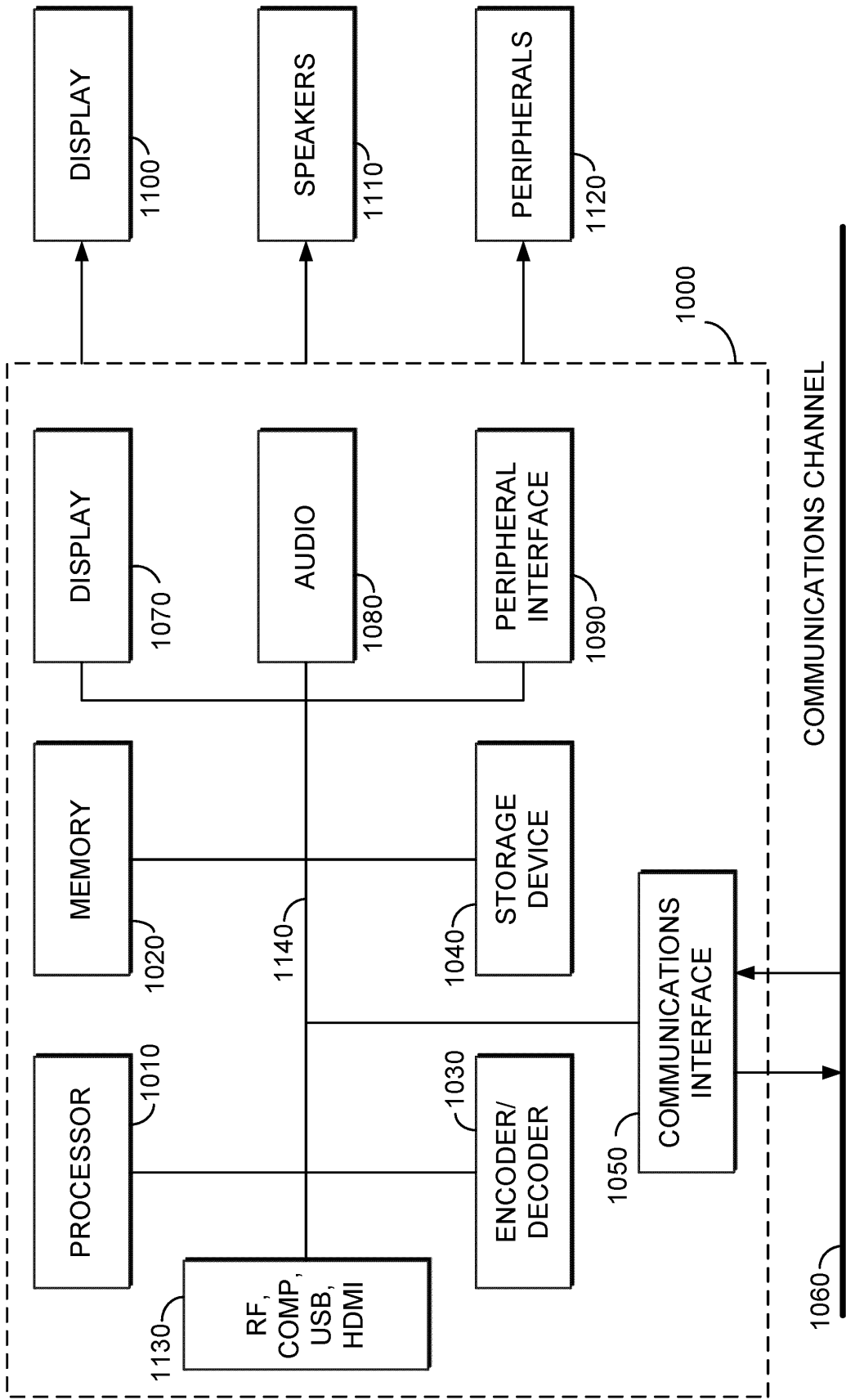


FIG. 47