Supplementary Material #2: X3D Parsing Algorithm

Fig. 1 summarizes the pipeline of the X3D parser. In the following, we will elaborate the components in the pipeline. Relevant listings from the paper are replicated here for clarity.

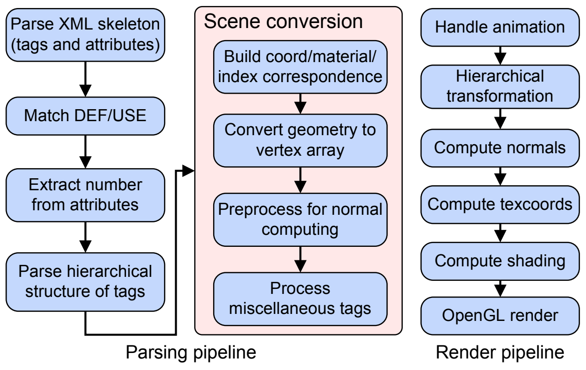
Parse XML skeleton (tags and attributes)
Match DEF/USE
Extract number from attributes
Parse hierarchical structure of tags
Scene conversion
Build coord/material/ index correspondence
Convert geometry to vertex array
Preprocess for normal computing
Process miscellaneous tags
Handle animation
Hierarchical transformation
Compute normals
Compute texcoords
Compute shading
OpenGL render
Parsing pipeline
Render pipeline

Figure 1: *X3D parser pipeline.*

XML Skeleton We define XML skeleton to be a list of XML tags and attributes, appearing in the same order as the input file.

We use an automaton-based method to generate the skeleton. The input file is parsed by a finite automaton, where a state transition is made at each character. Automaton states are computed using a function composition `scan` as in [Hillis and Guy L. Steele 1986]. Tags and attributes are then recognized at specific state/character combinations.

Match DEF/USE X3D allows reuse of data using DEF/USE attributes. A DEF attribute defines the name of a tag. A later USE attribute with the same name indicates a reference to the matching DEF. The DEF tag's data is reused in the USE's context.

To find a matching DEF for each USE, we construct a hash table for all DEF names. A superstep fills in the table for each DEF, and a subsequent superstep looks up names for each USE. We currently do not handle hash collisions, as none has occurred in our test scenes.

Extracting numbers The subroutine in Listing 1 is used to extract numbers from attribute values.

Listing 1 Number extraction

```
/*
extract numbers from plain text
input:
    begin/end: offset to begin/end of regions
    n: number of regions to parse
returns:
    parsed numbers
*/
float* getNumbers(int* begin, int* end, int n){
    float* ret = NULL;
    spawn(n){
        id = thread.rank;
        s = begin[id]; e = end[id];
        pt = s+thread.fork(e-s+1);
        c = charAt(pt-1); c2 = charAt(pt);
        thread.kill(isDigit(c)||!isDigit(c2));
        require
            ret = dnew[thread.size]float;
        ret[thread.rank] = parseNumber(pt);
    }
    return ret;
}
```

Parse hierarchy We store the hierarchical structure of a X3D file as a pointer to its parent in each tag.

A tag's parent is computed by finding the nearest preceding node with exactly one fewer nest level. First, a scan is used to compute nest levels, by substituting 1 for beginning tags and -1 for end tags. The maximal nest level is then computed using a reduction. Finally, parents for nodes on each level are computed using a scan with operator `replace(a,b)= b==-1?a:b`. In this replace scan, node ID is substituted for nodes on the parent level, and -1 is substituted for other nodes.

Scene conversion For each geometry tag, corresponding connectivity, coordinates and material tags are all found using the replace scan. A GPU-ready vertex array is then constructed using these information.

For subsequent normal computing, a one-ring neighborhood is computed using Listing 2.

Listing 2 Find neighboring triangles

```
findFaces(int* pf, int* hd, int* ib, int n){
    spawn(n*3){
        rk = thread.rank;
        f = rk/3;           //face id
        v = ib[rk];         //vertex id
        thread.sortby(v);
        rk = thread.rank;
        pf[rk] = f;
        barrier;
        if(rk==0||thread.get(rk-1,v)!=v)
            hd[v] = rk;
    }
}
```

Data in miscellaneous tags like lighting and camera are collected and provided on the CPU during rendering. We compact them into a list and send the list to the CPU. Such tags typically contain insignificant data, and its overhead does not have a noticeable impact on performance.

Rendering During rendering, we first handle animation by updating the vertex buffer using current frame's data. Vertices are then transformed, and normals and texture coordinates are computed. The prepared vertex buffer is finally shaded and rendered using OpenGL.

References

HILLIS, W. D., AND GUY L. STEELE, J. 1986. Data parallel algorithms. *Commun. ACM 29*, 12, 1170–1183.