Syntactic Analysis of Graphical Information Using Recursive Scoping

Alexei Razumowsky [1]

[1] *Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 65 Profsoyuznaya street, Moscow, 117997, Russia*

Abstract

The report presents a new method for structuring, segmentation, and algorithmic design of the character array parser using VRML data as an example. The key feature of the method is the possibility of forming a hierarchically complex object by means of recursive data structuring, which allows you to cover the entire contents of the object, including its arbitrary nesting of child objects. This leads to a highly manageable development of the parsing algorithm, allowing you to focus each time on a specific piece of data, while not losing sight of the entire aggregate coherence of the information. The results obtained can easily be used in plans for creating convenient data storage structures related to information security, solving the problem of containing the amount of data in files, data management problems in heterogeneous systems, and hierarchical data solutions in the Internet of Things.

Keywords

Syntax analysis, recursive scope, VRML format, polymorphic hierarchy.

1. Introduction

Any subject or system analysis is necessary and in demand because of the content complexity of the data of the subject or system. Information complexity is something that is difficult to imagine or not amenable to mental coverage. Therefore, the information must be properly prepared before direct use. Among the main methods of such training is the analysis of information and its practical testing. In the modern world, the abundance of information is the norm, so not only the resources that provide information are in demand, but also the methods that can be used to find, correlate, transform the necessary information, analyze and synthesize it.

In this study, we study the data of the VRML (Virtual Reality Modeling Language) graphic format, which was created for modeling virtual reality. The format, which gained huge popularity in the 90s and was subsequently replaced by its successor, the X3D virtual reality modeling language, is still quite popular. Its applications range from interactive vector graphics and web technologies, to medical applications, construction, cartography, and multimedia systems. The need to use this format is still high in computer-aided design systems, including data exchange between various graphics systems [1-4]. The language operates with objects that describe geometric shapes and their location in space [5].

Using character format files, the VRML language allows you to determine not only the geometric properties of objects in three-dimensional space, such as the location and shape of complex surfaces and polyhedra, but also physical data about their color, texture, gloss, transparency , and light sources. In addition, as a reaction to user actions or other external events, it is possible to supplement the description of the model by setting parameters for movement, sounds, lighting, and other aspects of the virtual world [6].

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia
EMAIL: razumowsky@yandex.ru (A. Razumowsky)
ORCID: 0000-0003-1769-0178 (A. Razumowsky)

Before looking for typical algorithmic ways to parse data, it is important to understand that the analysis is consistent with the operations of sequentially reading certain portions of data from a file, step-by-step identification of VRML objects by characteristic features, matching with data already available in the storage, and, finally, forming the necessary structure for interacting with these objects.

Like most languages, VRML has a clear design and a specific set of lexical, syntactic and semantic rules that determine the appearance of the program and its actions. From which one can make a false conclusion that writing a parsing algorithm will not be difficult. The entire structure of the format has already been thought out by the developers of the language, and in this case, it is only necessary to create a method that will allow it to be correctly read. However, when writing a program, difficulties arise associated with the hierarchical features of the language.

We present a new approach to the algorithmic organization of data parsing, which eliminates the problems associated with the hierarchical complexity of VRML information. The method is based on recursive reading of VRML objects, their sequential analysis, identification and comparison with already defined data. To store the meaningful structure of objects, a special container is used, which simultaneously provides convenient access to the entire set of objects, as well as the ability to interact with both individual objects and with the entire structure of objects as a whole.

2. Research methods

Before identifying and making an attempt to solve the problems of parsing complex structured data, it is necessary to classify the types of their structural organization.

There are three kinds of descriptions in VRML format:

- Simple objects (without specification), which are sequentially read from a file and immediately placed in the appropriate place in the polymorphic hierarchy.
- "Anchor" objects (AO), highlighted in the format by the keyword "DEF". They contain the definition and description of object properties for later use in the hierarchy. The order of processing such objects is as follows: first, data is read that has a complete description of the object, then this data is placed in a separate DEF container, which stores "anchor" objects identified by the key-name. Further, a pointer to such an object is placed in the appropriate place in the hierarchy.
- Custom Objects Defined by the "USE" Specification: Indicate the use of a specific "anchor" object in the hierarchy. At the time of reading the user object, a search is performed for the corresponding key in the DEF container, after which the found pointer is placed in the storage.

The first and most obvious problem that you can face when analyzing VRML data is multiple nesting of objects, which in some cases is redundant (Fig. 1).

By means of the C ++ language, it is possible to describe and implement nesting, however, for this, objects must be of different types. In VRML data, you can often see how the same object (for example, an object of type Group) inherits itself several times from itself, which makes it impossible to consistently form the structure of nested data. In the illustration, the red frame denotes the situation of multiple nesting of the Group object (Fig. 1).

The next problem comes from the syntax of the VRML format. In VRML, as in many other languages, anchor objects (AO) are allowed, which can be described in a file only once, and then used an indefinite number of times by means of a nominal reference. In other words, these are definitions of complex data types, with the ability to create complexly grouped objects. The complication here is the likely loss of control over the heap, which is allocated every time data is read. Since AO can be used in an algorithm more than once, it is necessary to definitely know at what point it should be removed from memory. In other words, additional surveillance should be established for AO.

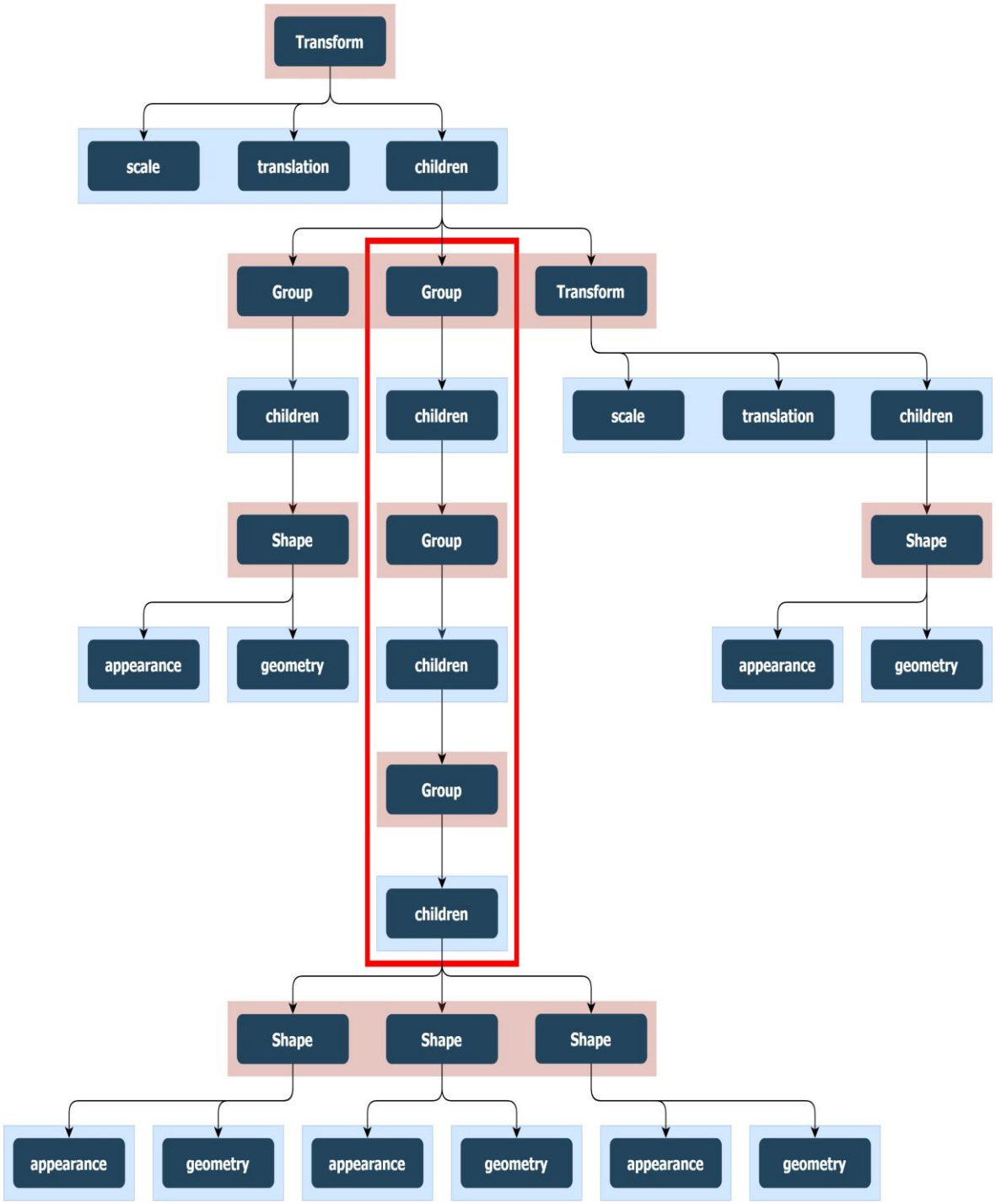


Figure 1: Example of VRML structure

The latter problem may not always be encountered, but only when reading a multi-file object: VRML assemblies. Parsing and reading of assemblies is organized similarly to single-file data. However, the incident is that the same file can be used multiple times in an assembly. This means that it is difficult to uniquely identify objects in such an assembly. In addition, a memory leak occurs due to the rewriting of pointers to "anchor" objects, which, having the same name, naturally behave differently in different files (Fig. 2).

Memory for such objects is allocated in the general order, but it may not be released in time, since the pointers to them are overwritten during the subsequent reading of data referring to the same object.

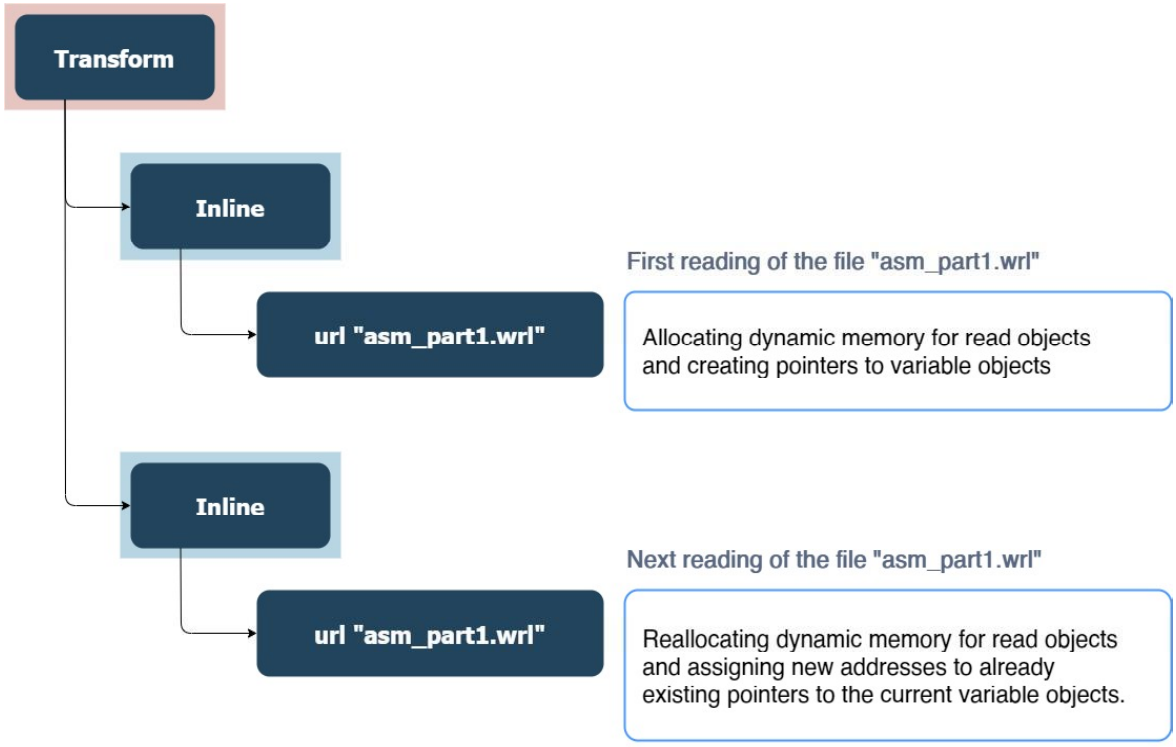


Figure 2: An example of reusing a file in an assembly

2.1. Creating a generic container for VRML objects

In order to solve the problem of nesting objects when creating a full-fledged VRML structure, it is also important to understand the semantics of the VRML format. So, there is a specific parent object that contains some properties and an array of children. Such an array, in turn, includes many similar objects that have individual properties and their own array of inherited objects. At the same time, such a representation still does not answer the main question: how to cope with the problem of multiple inheritance? After all, the array of child objects (children) can contain almost any arbitrary object of the VRML format: Transform, Group, Shape, and so on. Here we are seeing the complexity of data coverage only increase. A non-trivial view of multiply nested data is required. In other words, it is required to express nesting abstractly, while preserving the individual features of the data format.

Let's classify the types. The entire VRML structure consists of so-called nodes (Node) [5,7], which are subdivided into geometric nodes - Box, Cone, Cylinder, IndexFaceSet - and grouping nodes - Group, Transform, Collision. In addition, there are nodes that are responsible for the graphical representation (Appearance), as well as nodes that define the properties of geometric objects (geometry). Accordingly, focusing on such a classification, it is easy to form a polymorphic hierarchy in the C ++ language.

Let's define an abstract class Node as a generalization of all sorts of nodes in the VRML format. Then we will establish inheritance of other classes from the abstract class Node. Thus, we obtain the systemic connectivity of arbitrary data elements by including their types in a general polymorphic hierarchical composition.

Further, since any type formed on the basis of the list of VRML format keywords can be included in the hierarchy, it becomes possible to create an array of pointers to an object of the base data type (Node *) of this hierarchy. An object of any type included in the hierarchy can be placed into this array as a pointer to the memory dynamically allocated for the object. In this way, it will be easy to implement the VRML data store, while maintaining their diversity.

2.2. Indirect monitoring of anchor objects

For storing AO and their subsequent use when building a polymorphic hierarchy, an associative container of the standard library of std::map templates has been chosen. The key in it sets the name of the object to be read, and the value is a pointer to the dynamic AO.

In the process of reading data, a check is performed - is the object an "anchor"? In the positive case, its pointer is placed in the global container DEF by the key - the name of the nuclear system. Subsequently, when reading a custom object, a search is performed by key in the DEF container, after which memory is allocated for it, and the object itself is placed in the data hierarchy. If the object is not "anchor", then memory is immediately allocated for it, and it enters the target storage.

The main data store is the dynamic array nodes of the standard template library, of type std::vector<Node *>. It is designed to collect the entire polymorphic hierarchy of readable objects. You can identify whether the current object is an "anchor" by checking the name string def. If the object does not have it, then the object is simply placed in the main storage. Otherwise, the object is qualified as AO, then it is also placed in the std::map<string, Node *> DEF storage in order to indirectly monitor its subsequent use:

```
if (group->def != "") {
        DEF->insert(std::pair<string, Node*>(def, group));
        def.clear();
    }
```

If during reading a user object ("USE") is found, then by its name a search is performed in the DEF storage by the key (object name + file name) and the object found there is placed in the main nodes container:

```
if (str == "USE") {
        file >> str;
        shape->geometry = (Geometry*)DEF->at(str + "_" + cur_file);
    }
```

Thus, all AO are stored in the DEF container (Fig. 3).

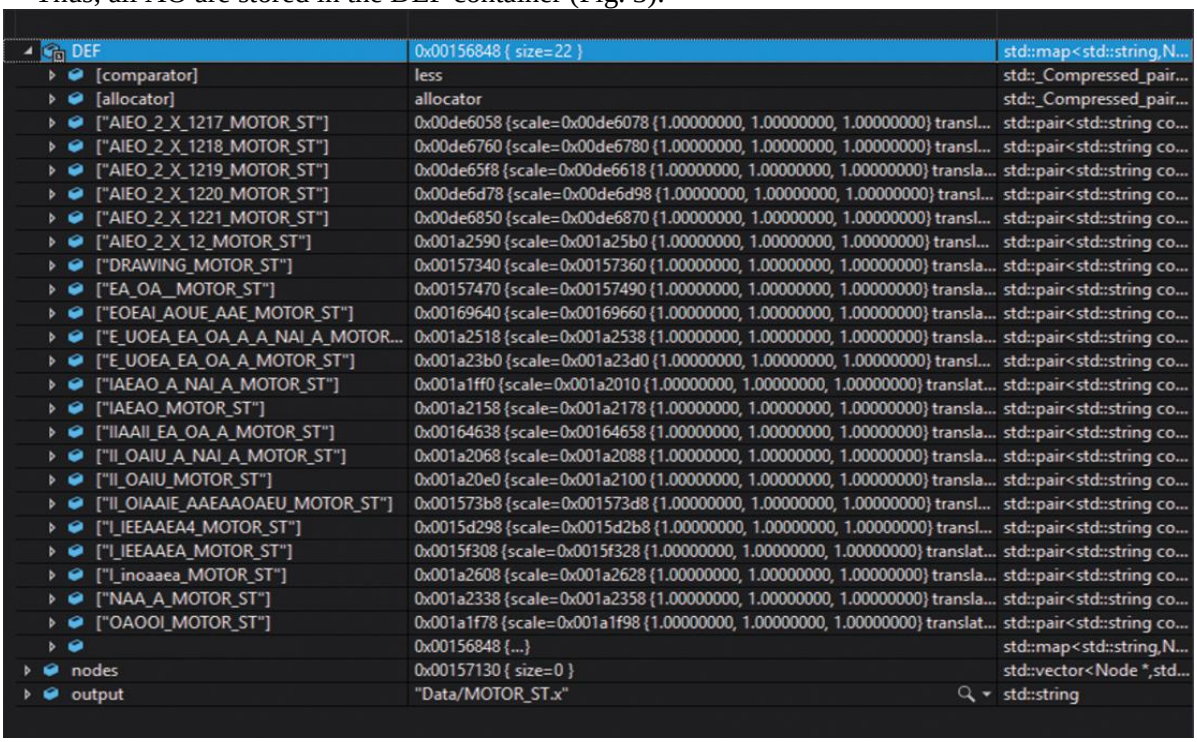


Figure 3: Contents of AO

The main repository, which provides the content of the hierarchical structural connectivity of objects, contains pointers to AO from DEF, as well as objects without specialization. In order to correctly delete and clean up the entire data structure, it is necessary to check before deleting an object

- is it an anchor? If not, then the deletion should be performed immediately. Otherwise, the deletion is performed when all parsing is complete.

2.3. Identification of AO

To carry out unambiguous identification by the name of an object, it is necessary to form a complex naming by adding a certain suffix to the name. Such an additional nominal addition will make it possible to distinguish between AO when reading VRML assemblies, in which the names of variable objects may be the same. It is convenient to use this suffix to define the name of the file in which the object being read is located.

However, the problem with ambiguous identification of variables when using the same file several times still remains, since the file names and object names can be the same. To solve this problem, it is necessary to create an additional set of elements related through the name. The standard template library provides an associative container std :: map, which is very convenient for achieving this goal:

std::map<string, vector<Node*>*>* FILES

In the FILES object, the key will be the name of the file read earlier, and the value will be a pointer to the correspondingly processed hierarchical object. When the file name is read, the FILES container is searched. And if this file has not been analyzed earlier, then the objects are sequentially read from this file with the creation of the corresponding hierarchy of objects, the pointer to which is used to add to the FILES storage.

This method of identifying variables has several advantages:

- Unambiguous identification of AO when reading an assembly with the same files.
- Unambiguous identification of AO when reading an assembly with different files, in which the names of the variables are the same.
- Optimization of the program execution time by skipping the repeated reading procedure of already existing data.

2.4. Recursive data scoping method

As problems are identified and their features are clarified, it is important to choose the right means and ways to achieve the target result. So, it is necessary to implement a set of methods that will allow the following:

- Read data from a VRML-format file.
- Parse the read data into tokens.
- Identify tokens in objects and match them to establish relationships between them and other properties.
- Build a polymorphic hierarchy based on the received objects.

The problem of the need for additional monitoring of AO, as well as the uncertainty of the level and degree of nesting of objects at the stage of reading data, confronts the need to identify objects as if their nesting depth and properties are already known. To do this, it is convenient to use a recursive approach, which will allow you to cover the data in its presumptive integrity. The main procedure that reads the VRML format is to set the parseNode function (Fig. 4).

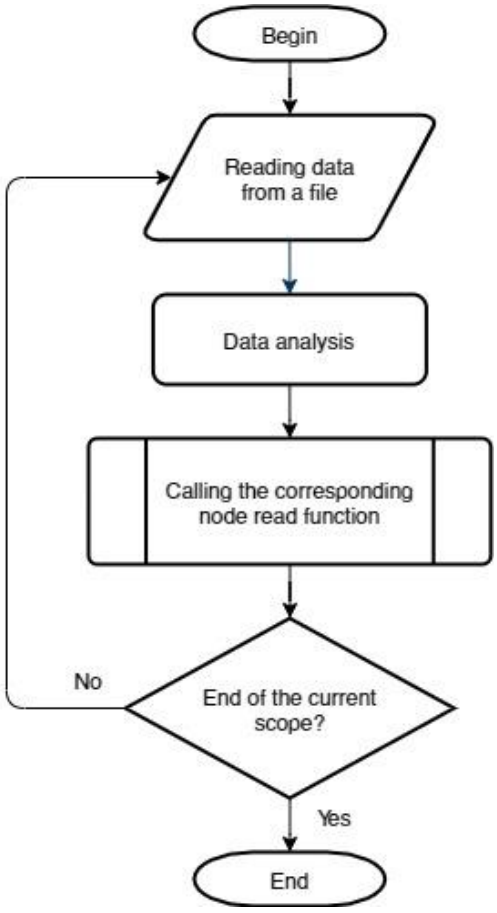


Figure 4: Flowchart of the VRML file reading algorithm parseNode

This function takes the following parameters:

- Object of the file stream, from where the data should be read;
- The container in which all read objects should be placed;
- A string defining the current position of the pointer to data in the file;
- The name of the current file to create unique object names.

Depending on the qualification of the object: Transform, Group, Shape, Inline, the corresponding function is called to set its properties. In addition, the name of the object is recorded for subsequent association and identification of AO. The name of the object is also appended to the name of the current file from which it was read. The implementation of all methods for reading grouping nodes - Group, Transform, Collision and others - is most clearly seen in the example of the algorithm for recursive reading of the Group object - parseGroup, since the grouping nodes differ only in their properties, and it is important to focus on the implementation of reading their child objects (Fig. 5).

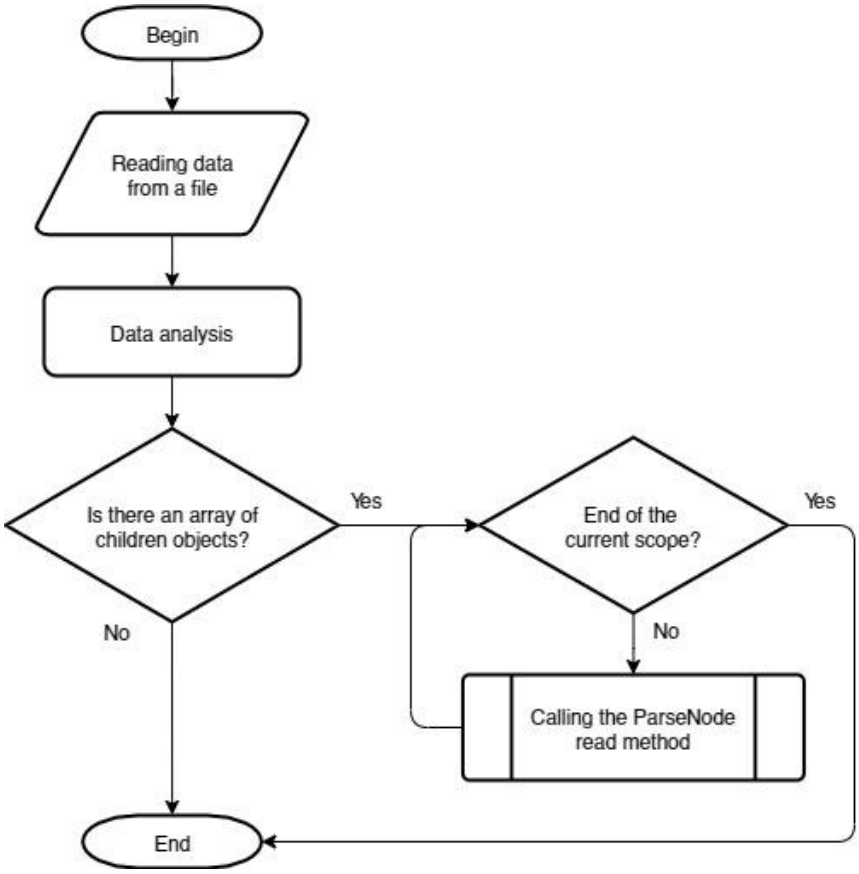


Figure 5: Flowchart of the parseGroup reading algorithm

If the object has an array of dependencies children, a loop is organized in which the parseNode method is iteratively called and the child object is read recursively and then placed in the children container.

The parseNode function has the following input parameters:

- Object of the current file stream;
- A string that defines the position of the pointer in the file;
- The name of this file to create unique object names;
- The container into which the read objects are to be placed. Here, the children container of the current object is passed as a container.

Similarly, the entire hierarchical chain of child objects is recursively covered: if the child object is also a grouping node, then a similar cycle is started when it is read.

The solution to the problem of unambiguous identification of objects when reading an assembly VRML file is assigned to the Originality function. This operation is responsible for the direct build-up of the polymorphic hierarchy. The main feature of the function is the unambiguous identification of objects using the FILES container described in 2.3 (Fig. 6).

The implementation of the Originality method is divided into two stages:

1. Reading the file name.
2. Check whether the file was read earlier or not.

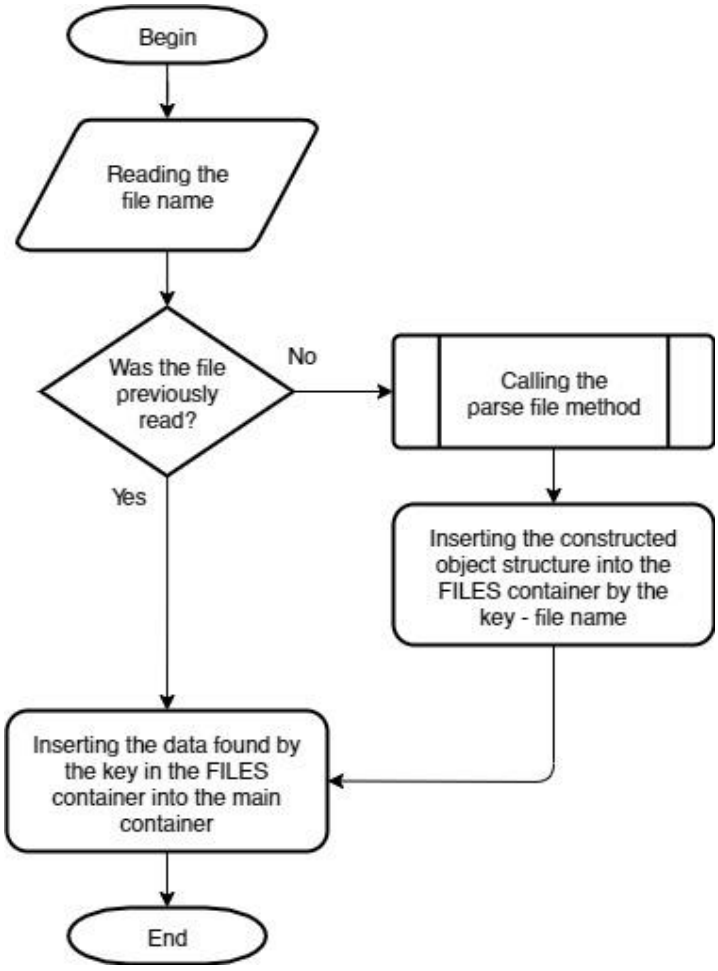


Figure 6: Flowchart of VRML assembly reading algorithm

3. Obtained results

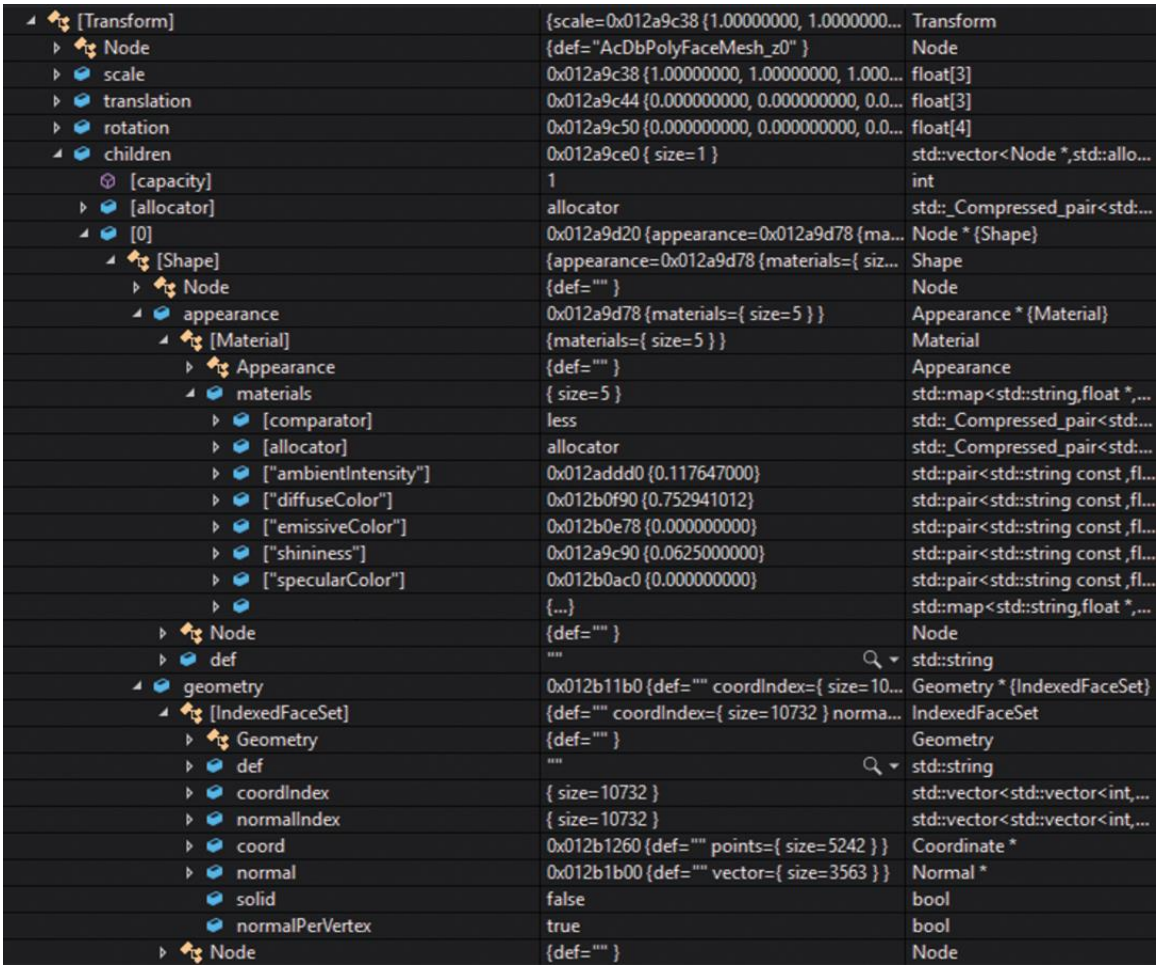
As it reads a VRML file (or builds it), the program recursively builds a polymorphic hierarchy of all objects, their properties and descendants (Fig. 7).

The resulting hierarchy is presented in a tree-like form and is stored in an object of type std :: vector. This storage method allows for the best convenient access to all objects, as well as ease of interaction with them. In addition, using the recursive coverage method, it is quite easy to carry out additional monitoring of AO, since each AO is observed simultaneously both as a separate object and containing a set, the volume of which is unknown in advance, of other objects.

The format for representing a VRML structure as a polymorphic hierarchy has the following advantages:

- Ability to operate with individual objects in the context of their entire hierarchical structure.
- Ability to process the entire array of accumulated data in a uniform way, since interaction with each object uses the functionality of polymorphism of the object-oriented approach.

The latter advantage contributes to a more efficient way of outputting all accumulated data to a file of a format of a different configuration.



[Transform]	{scale=0x012a9c38 {1.00000000, 1.0000000...	Transform
Node	{def="AcDbPolyFaceMesh_z0" }	Node
scale	0x012a9c38 {1.00000000, 1.00000000, 1.000...	float[3]
translation	0x012a9c44 {0.000000000, 0.000000000, 0.0...	float[3]
rotation	0x012a9c50 {0.000000000, 0.000000000, 0.0...	float[4]
children	0x012a9ce0 { size=1 }	std::vector<Node *,std::allo...
[capacity]	1	int
[allocator]	allocator	std::_Compressed_pair<std:...
[0]	0x012a9d20 {appearance=0x012a9d78 {ma...	Node * {Shape}
[Shape]	{appearance=0x012a9d78 {materials={ siz...	Shape
Node	{def="" }	Node
appearance	0x012a9d78 {materials={ size=5 } }	Appearance * {Material}
[Material]	{materials={ size=5 } }	Material
Appearance	{def="" }	Appearance
materials	{ size=5 }	std::map<std::string,float *,...
[comparator]	less	std::_Compressed_pair<std:...
[allocator]	allocator	std::_Compressed_pair<std:...
["ambientIntensity"]	0x012addd0 {0.117647000}	std::pair<std::string const ,fl...
["diffuseColor"]	0x012b0f90 {0.752941012}	std::pair<std::string const ,fl...
["emissiveColor"]	0x012b0e78 {0.000000000}	std::pair<std::string const ,fl...
["shininess"]	0x012a9c90 {0.0625000000}	std::pair<std::string const ,fl...
["specularColor"]	0x012b0ac0 {0.000000000}	std::pair<std::string const ,fl...
	{...}	std::map<std::string,float *,...
Node	{def="" }	Node
def	""	std::string
geometry	0x012b11b0 {def="" coordIndex={ size=10...	Geometry * {IndexedFaceSet}
[IndexedFaceSet]	{def="" coordIndex={ size=10732 } norma...	IndexedFaceSet
Geometry	{def="" }	Geometry
def	""	std::string
coordIndex	{ size=10732 }	std::vector<std::vector<int,...
normalIndex	{ size=10732 }	std::vector<std::vector<int,...
coord	0x012b1260 {def="" points={ size=5242 } }	Coordinate *
normal	0x012b1b00 {def="" vector={ size=3563 } }	Normal *
solid	false	bool
normalPerVertex	true	bool
Node	{def="" }	Node

Figure 7: Fragment of the contents of the VRML data store nodes

4. Conclusion

Any information has individual complexity, which begins to manifest itself in the analysis and parsing of data. The initial assumption of simplicity of parsing, due to the fact that VRML is seen as a well-structured language, turns out to be deceiving. A closer look at the task reveals the problems associated with finding the right way to overcome the barriers of complexity.

When choosing a parsing method, it is important to pay attention to the details of the connectivity of information elements, as well as to the sequence of their processing and transformations.

The recursive data-scoping method presented in the article offers a simple and elegant solution to the problems of information connectivity and data processing order. By means of a recursive way of reading and structuring objects, it becomes possible to create massive hierarchical structures, with elements of which can then be easily interacted with. This opens up the prospect of operating objects not only as separate units, but also in the context of taking into account the entire hierarchical structure.

The presented method of parsing data makes it possible, by sequentially reading the data, to form "on the fly" a hierarchy of objects directly based on the structure of their format. In other words, the direct benefit of this approach is the absence of additional transformations and structuring when reading data from a file. This shortens the read time and also reduces the complexity of the program code.

The most important conclusion is that, thanks to the method of recursive data scoping, it is possible to programmatically perceive and transform any hierarchical arrays of character data. The proposed approach can be expanded in terms of research on parallel data processing, as well as to create convenient data storage structures related to information security [8]. The problems of containing the amount of data in files and increasing the speed of reading remain urgent [9] - there is also a wide field of applications of the presented method. In terms of big data management [10, 11], in connection with the emergence of heterogeneous and hybrid storage systems such as Hadoop and Spark, a significant

prospect for this method is seen. Finally, a broad horizon opens up in the creation and processing of hierarchical data related to the Internet of Things (IoT) [12], which promises significant benefits in the perception of distributed information.

5. References

[1] F. Ziwar, R. Elias, VRML to WebGL Web-based converter application, 2014 International Conference on Engineering and Technology (ICET). IEEE. (2014) 1-6. doi: 10.1109/ICEngTechnol.2014.7016798.

[2] J. Whyte, N. Bouchlaghem, A. Thorpe, R. McCaffer, From CAD to virtual reality: modelling approaches, data exchange and interactive 3D building design tools, Automation in construction Vol. 10, N. 1. (2000) 43-55.

[3] T. Pazlar, Ž. Turk, Interoperability in practice: geometric data exchance using the IFC standard, Journal of Information Technology in Construction (ITcon) Vol. 13, N. 24. (2008) 362-380.

[4] D. Huber, The ASTM E57 file format for 3D imaging data exchange, Three-Dimensional Imaging, Interaction, and Measurement. International Society for Optics and Photonics, 2011, January. Vol. 7864. P. 78640A. doi: 10.1117/12.876555.

[5] G. Taubin, W. P. Horn, F. Lazarus, J. Rossignac, Geometry coding and VRML, Proceedings of the IEEE Vol. 86, N. 6. (1998) 1228-1243.

[6] D. R. Nadeau, Building virtual worlds with VRML, IEEE Computer Graphics and Applications Vol.19, N. 2. (1999) 18-29.

[7] M. Pesce, VRML browsing and building cyberspace (No. 006 P473), New Riders Publishing, 1995.

[8] R. Tamassia, N. Triandopoulos, Computational bounds on hierarchical data processing with applications to information security, International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 2005, July, pp. 153-165.

[9] S. F. Schulze, P. LaCour, P. D. Buck, GDS-based mask data preparation flow: data volume containment by hierarchical data processing, 22nd Annual BACUS Symposium on Photomask Technology, International Society for Optics and Photonics, 2002, December. Vol. 4889. pp. 104-114.

[10] K. R. Krish, B. Wadhwa, M. S. Iqbal, M. M. Rafique, A. R. Butt, On efficient hierarchical storage for big data processing, 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2016, May, pp. 403-408.

[11] X. Wang, L. T. Yang, L. Kuang, X. Liu, Q. Zhang, M. J. Deen, A tensor-based big-data-driven routing recommendation approach for heterogeneous networks, IEEE Network, 2019, Vol. 33, N. 1. pp. 64-69. doi: 10.1109/MNET.2018.1800192.

[12] S. Wan, Y. Zhao, T. Wang, Z. Gu, Q. H. Abbasi, K. K. R. Choo, Multi-dimensional data indexing and range query processing via Voronoi diagram for internet of things, Future Generation Computer Systems, 2019, Vol. 91, pp. 382-391. doi: 10.1016/j.future.2018.08.007.