Language Support for Distributed Virtual Reality

Bedřich Beneš (benes@sgi.felk.cvut.cz)
Aleš Holeček (zara@sgi.felk.cvut.cz)
Jiří Žára (zara@sgi.felk.cvut.cz)

CTU, Fac. of Electrical Eng.
Dept. of Computer Science
Karlovo nám. 13,
121 35 Praha 2

Abstract

One of the main issues in the research on Multi-user Distributed Virtual Reality Systems is the virtual environment description language. A number of different languages with various level of description formalism already exist. In this paper we provide a quick overview of the existing systems. We look more into details of Virtual Reality Modeling Language based on SGI's Open Inventor format. The problems of multi-user reality systems are farther examined from the point of view of keeping the world database consistent. We present methods reducing the communication demands based on using more complex specification of the environment behavior and distribution of the world database. Finally, the paper provides an explanation of our research. We attempt to design an algorithm for direct global visibility preprocessing, which is an important issue related to the world database subdivision.

Keywords

Scene description, distributed virtual reality, global visibility, Open Inventor, VRML.

1 Introduction

Over the past several years, the words Virtual Reality (VR) have become very popular among the majority of the human population. Although this phrase is one of the most cited phrases in all media types, there is no exact definition of VR given. Everyone has their own ideas of what exactly virtual reality means. For some people it is a science, which requires head-mounted displays and electronic data-gloves, for others every video game is a virtual reality.

Our definition lies somewhere in between those two extremes. We define *Virtual Reality* as a computer-generated simulation of a three-dimensional environment, which the user is able to interact with and manipulate the contents of it ([3]).

Previous research shows that a complete simulation of all phenomena of the *real world* is not necessary in VR application. The human perception of the real world is mediated through the five major senses; sight, hearing, touch, taste and smell. Not all of them are equally important. It is sufficient to create a simulated world, which is *real enough* that a human observer can accept it and participate in it.

Virtual Reality is already being used in many different application areas. Good examples could be entertainment industry, computer games, architecture design and walkthroughs, engineering design, medicine, education, chemistry, biology and many others.

The field of Virtual Reality is moving gradually into a new phase. Concept of *Distributed Virtual Reality* (DVR) has been introduced and it has become one of the most popular research topics. The DVR can be described as a simulation of a complex environment running on several computers connected over the network. People visiting or working in the virtual world should be able to share it, interact with it, and communicate in it in a real time.

There are still many problems to be solved, before Virtual Reality Systems will provide immersive environment, where people could collaborate on projects, having all different types of communication available. Every new technology goes through a period, when methods and designs are being invented and changed. VR is not an exception. It is even more complicated by the fact that the foundations of the field, *virtual environment description language*, is still going through an evolution. Such a language should contain of structures for objects geometry annotation, as well as specification of objects which can change over time and in response to external stimuli (e.g. user interaction). It should support description of a virtual environment distributed over the network as well as an environment populated by more then one user.

In this paper, we would like to present a brief overview of the existing languages designed for the description of the virtual environments. We focus mainly on the standard – *Virtual Reality Modeling Language* (VRML). We look closely at the object geometry description facilities of VRML as well as the facilities provided for the object behavior description and user × scene interaction. We attempt to define the obstacles which are tightly related with the multi-user distributed reality systems. At the end we present what methods are investigated as a part of our research and how this research can influence the evolution of the virtual environment description language.

2 Frequently Used Jargon

In this paper we use certain jargon specific for the field of VR and DVR. Although some of the words are considered to be terms in the area of computer science, we will provide a quick explanation.

Each of the computers participating in the simulation is called *host*. On each host there are number of *entities*. Entity is any object in the virtual environment. If needed entity may communicate its state by sending *update messages*. The entity devoted to embodiment of a human participant is called *avatar*. By embodiment we understand that the user is provide with appropriate body images which represent the user to others in user × user interaction. The avatar should address not only issue of presence, but also location, orientation, identity, activity, availability, gestures, expressions, and others.

Next we define *focus*. It represents a subspace within which a person focuses their attention. The idea is that person is more aware of objects inside their focus and less aware of objects outside of their focus. Accordingly *nimbus* represents a subspace within which a person projects their presence. The idea is that an object within person's nimbus is more aware of the person, then an object outside of the nimbus. Union of focus and nimbus is called *awareness*.

World Database is a description of the virtual environment containing information about the geometry, constrains, behavior, location and state of the entities.

3 Existing systems

In this section we would like to give a quick overview of existing languages designed for describing the virtual environments. As it was already mentioned in Introduction, such a language should support time-varying models, user interaction, and multi-user sessions. It should be flexible and open, allowing extensions in all aspects of functionality. There is a number of research and commercial systems which support some or all of these requirements.

- **TBAG**, developed at Sun Microsystems, is a functional 3D system. Function objects in TBAG can be related to each other by multi-way constrains and then evaluated with different parameters to create time-varying or user controlled geometries.

- The **UGA** system, developed at Brown University, uses an interpreted language, *Flash*. This is a prototype-delegation object oriented language which was specifically developed

for fast prototyping of complex interactive 3D scenes. It supports a variety of animation techniques, including keyframing, inverse kinematics, physically based modeling, and evaluation of arbitrary functions.

- **ANIM3D** was developed by Digital corp. to visualize algorithms and uses a costume prototype delegation language *Obliq*, which provide for concurrent behaviors.
- **Alice**, from University of Virginia, is specifically designed to support rapid prototyping of 3D immersive environments, and uses interpreted language *Python* as an extension.
- The **Behavior Engine** system, developed by the BE Software Company, provides an system of Inventor-based node types for describing behavior. It defines a Pascal-based syntax for embedding new behavior internal nodes, and demonstrate some nodes for physically based modeling.
- **VRML+**, commercial product developed by the Worlds, Inc, allows multiple participants to take part in a dynamic environment, with animations, specialized audio, and texture mapping incorporated. VRML+ also includes behavior extension protocol which is language independent. It defines API for modifying the scene graphs. There is also proposed a networking protocol for connecting VRML servers and clients in VRML+.
- **VRML**, from SGI, is based on 3D graphics system *Open Inventor* developed also by SGI and written in C++. VRML is the one most commonly used description language. It is de facto standard in the area of languages for Distributed Virtual Reality and that is why we will devote the next section to it.

4 Virtual Reality Modeling Language

The geometry and behavior description facilities of Virtual Reality Modeling Language (VRML) are based on Silicon Graphics (SGI) file format known as Open Inventor. Although features of Open Inventor and VRML are not identical, we briefly introduce Open Inventor file format, because of its well defined and flexible syntax and functionality.

4.1 Open Inventor

Open Inventor is a complex definition of both data format for scene description and tools and library for 3D modelling and rendering. Although the domain of Open Inventor is Silicon Graphics hardware, more and more platforms with different operating systems accept it.

From the historical point of view, two versions of Inventor were introduced. Version 1.0 is called **IRIS Inventor** and is not supported anymore. Current, the second version (2.0) is called **Open Inventor** and allows describing not only static scenes, but also highly structured ones with a dynamics and a predefined behavior, included.

In the following text, we will speak about Open Inventor from the point of view of its **file format**. Programmer's tools as well as Open Inventor library will not be described. The Open Inventor files can be stored both in ASCII and binary format, but only text format is available for public purposes.

The scene data are arranged into five basic classes (see also fig. 1):

1. **Group** Array, Group, Separator, Switch, ...
2. **Shape** Cone, Cube, Cylinder, Sphere, NurbsCurve, NurbsSurface, IndexedFaceSet, Text2, Text3, ...
3. **Property** Coordinate, Material, NormalCoordinate, Texture, Transform, ...
4. **Light** Directional, Point, Spot
5. **Camera** Orthographic, Perspective

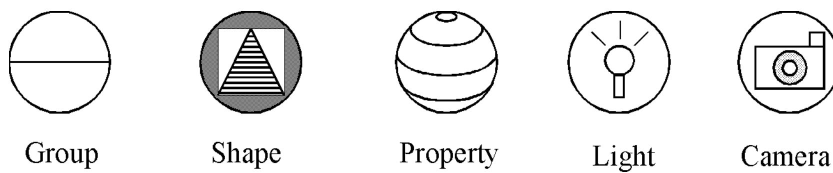

Figure 1: Symbols for basic Open Inventor entities

The basic idea of the hierarchical scene description is to arrange data into a tree structure, similar to PHIGS (see fig. 2). The tree consists of internal nodes (**Groups**) and leaves with **Shapes, Properties, Lights and Camera(s)**.

Two different types of inheritance are applied during traversing the scene tree. One type of inheritance is used for **Transformation** nodes, which are concatenated together. The path from the root to each Shape node determines, which accumulated transformations will be used for final positioning of an object. Transformations defined in certain subtree are not used in other subtree on the same tree level. This is ensured by a stack, which stores as many transformations as the current depth of the traversed tree is.

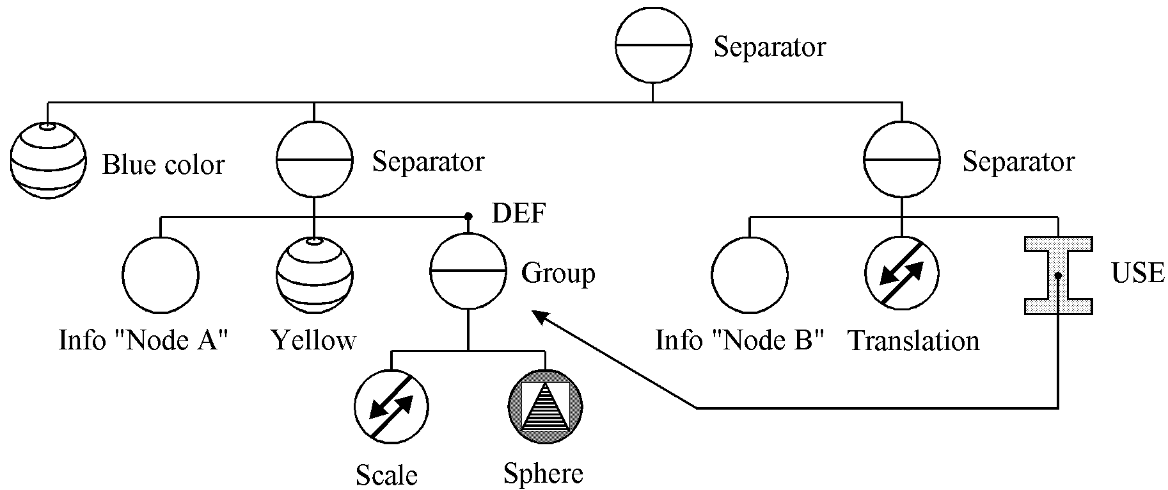

Figure 2: Hierarchy of a scene described by Open Inventor tree

Other kind of Property nodes like colors and textures are not accumulated, but they replaces the previous ones on the same level of tree. Processing of subtrees is similar to the transformation approach. An example is shown on fig. 2. The blue color node is placed on the first level of the tree. Next subtree defines a new color (yellow) and thus the sphere in this subtree is yellow. When the subtree is left, blue color is popped up from the color stack and blue is valid again for the rest of the tree.

Referencing already defined parts is possible using **DEF** and **USE** commands. The subtree defined after DEF statement can be later reused from an arbitrary place in a tree. The example is shown again on fig. 2, where the second object is scaled up sphere defined in previous subtree as a special block named *My_block*. The second sphere is scaled up first and then translated. Accumulated transformations are performed in reverse order, from the leaf to the root.

The following text presents Open Inventor file, which defines the tree on fig. 2.

```
#Inventor V2.0 ascii
Separator {
    BaseColor { rgb 0 0 1.0 }          #  blue as "global" color
    Separator {
        Info { string "Node A" }
        BaseColor { rgb  1 1 0 }       #  yellow color
        DEF My_block            #  begin of definition of "My_block"
            Group {
                Transform { scaleFactor 2 2 2 } #  scale up 2x
                Sphere { }      #  unit sphere scaled up
            }                   #  end of definition of "My_block"
    }
    Separator {
        Info { string "Node B" }
        Translation { translation 3 2.5 2 }
        USE My_block            #  reusing "My_block"
    }
}
#  Example of Open Inventor file format
```

Animation

Most of the 3D scene description languages do not implicitly contain formalism for object dynamics description. Open Inventor has implemented limited animation features based on simple transformation generators. Such nodes can be placed instead of (or together with) transformation nodes into any place in the scene tree:

Rotor	rotation generator
Shuttle	shuttle movement with constant speed
Pendulum	movement with respect of physical law

The following example shows node Rotor in Open Inventor file:

```
 #Inventor V2.0 ascii
Separator {
      Rotor {                    #  rotation generator
           rotation 0 0 1 0.1    #  Z is rotational axis, step is 0.1 rad
           speed 4.0             #  4 rotations per second
      }
      Cube { }                   #  unit cube
}
#  Rotation of cube
```

This approach is useful mostly for presentation purposes, because the animation is permanent regardless of time scheduling or object collisions. Anyway, this kind of animation can be used also in virtual reality scenes, where some objects are static (walls, furniture), some are simply animated (clocks, spiders, ventilators) and only limited number of objects is driven be special methods. Definition of such objects is possible in Open Inventor using user defined nodes.

User nodes

The common issue of making standards is extensibility with respect of consistency. Initially defined set of objects (nodes) can be found insufficient during a few years. Well defined format should contain possibilities for later extensions and definition of special user objects.

Open Inventor solves the problem of user defined data quite flexibly. Any unknown node identifier is taken as a special user node. The parameters of such a node are defined using statement **fields**, which declares user names and types. User defined node can contain either already known nodes or more complex subtree, but no data are displayed. The only exception is allowed by node **alternateRep**. Alternative Open Inventor tree can replace user defined node in case that application program is not able to process the user defined node by its special way.

Definition of user node called "Pencil" shows description of both user names and alternate representation constructed from cylinder and cone:

```
#Inventor V2.0 ascii
Separator {
   Pencil {
      fields [ SFFloat my_height, SFFloat my_radius, SFNode alternateRep ]
      my_height 10.0
      my_radius 0.8
      alternateRep Separator {          #  alternative representation
          Complexity { value 0.1 }      #  solids of revolution as polygons
          Cylinder { height 10.0 radius 0.8 }  #  body of Pencil
          Transform { translation 0 6.0 0.0 }
          Cone { bottomRadius 0.8 }             #  top of Pencil
      }
   }
}
#  User defined node "Pencil"
```

4.2 VRML

The **Virtual Reality Modeling Language** (**VRML**) is designed for distributed interactive simulation. The environment used for this simulation is a set of virtual worlds connected via **Internet** and hyperlinked with the World Wide Web URL format. VRML introduces all aspects of virtual reality – interaction, three dimensional realistic graphics with texture mapping, and lighting. It is the intention of the designers of VRML that it would become the standard language for description of interactive simulation.

The VRML was conceived at the annual World Wide Web conference in 1994 in Geneva. After one year duration of intensive discussion in 26th of May 1995 the **VRML Version 1.0** specification was released. Rather then some new specification the subset of existing one was used. The VRML is deeply based on the Open Inventor, which supports complete description of 3D scenes, lighting and materials. A subset of the Open Inventor was used and it was extended toward network exploitation.

The VRML version 1.0 does **not** support any interaction, except the hyperlink feature. The reason for this is, that no standard language for interaction has been designed yet and the authors of this project do not want to risk getting into "language war". It was promised, that **VRML version 2.0** will support arbitrary interactive behaviors.

The VRML version 1.0 meets the following three requirements:

- **Platform independence** which is guaranteed because the description file is in ASCII format. The seven bit text mode is fully hardware independent and is readable by any platform and under any operating system.
- **Extensibility** is solved by supporting self describing nodes already discussed in the previous section (see User nodes).
- **Ability to work over low-bandwidth connection** is ensured by high compression factor of the rendered scene description. The VRML is just a scene description format;

VRML browser can be thought as a parametric state machine and VRML defines its commands. Only the commands have to be sent over the network, what is reasonably expensive.

4.2.1 Open Inventor extension and differences

VRML is a subset of Open Inventor. The VRML supports 36 nodes consisting of the same groups as Open Inventor does: **Group** nodes (Group, Separator, etc.), **Shapes** (AsciiText, Cone, Cube, etc.), **Properties** (Coordinate3, FontStyle, etc.) **Lights** and **Cameras**; for complete information see [1].

The VRML does **not** support shape nodes as NurbsCurve and NurbsSurface, but introduces three new nodes: **WWWAnchor**, **WWWInline** and **LOD**.

The WWWAnchor group can be syntactically written as:

```
WWWAnchor {
            name "arbitrary URL"
}
```

When one of the children of the node is chosen, this node causes running the VRML browser and the current scene is replaced by a new one.

The WWWInline node has following syntax:

```
WWWInline {
            name "arbitrary URL"
            bboxSize SFFloat SFFloat SFFloat
            bboxCenter SFFloat SFFloat SFFloat
}
```

WWWInline node adds an object to the current scene. The description of the object can be located anywhere in the world of World Wide Web. The address is resolved from the `name`. When this object is being read it is not defined yet. It may take a while before it actually appears. If parameters `bboxSize` and `bboxCenter` are set, a proper bounding box is displayed before the object is loaded.

Suppose that we have the example from section 4.1. Instead of defining `My_block`, we will read its description from WWW site where is located. The section starting with the keyword DEF would be then replaced by the text:

```
WWWInline {
            name "http://sgi.felk.cvut.cz/IncludedObjects/My_block"
            bboxSize 2 2 2
            bboxCenter 0 0 0
}
```

Another **Group** node which is different from the Open Inventor specification is the LevelOfDetail node. It is used to allow an application to switch between variously complex representations of an object. The rule defining which level of detail is going to be used is based on the computation of a distance from the world-space eye point to the LOD center (parameter of the LOD node), transformed into the world space. If the distance is less then the first value specified in the ranges array (other parameter of the LOD node), then the first most detailed representation of the child is going to be used. For N values in the range array, there should be N+1 children of the LOD node defined. If too few children is specified, last child with the lowest level of detail is going to be used repeatedly. Extra children are simply ignored.

```
FILE FORMAT/DEFAULTS
LOD {
      range [ ]      #MFFloat
      center 0 0 0  #SFVect3f
    }
```

Other VRML extension is a filename in `Texture2` node, which can be an arbitrary URL. The node

```
Texture2 {
          filename "http://sgi.felk.cvut.cz/Textures/My_texture"
          wrapS 20
          wrapT 10
}
```

represents an example, when texture definition is located on the host `sgi.felk.cvut.cz` in the file `/Textures/My_texture`. The mapping of the texture is repeated 20 times in `S` and 10 times in `T` direction of the mapping coordinates.

Every VRML file starts with line `#VRML V1.0 ASCII` and the recommended file extension is `.wrl`.

5 Multi-user distributed virtual reality systems

As we have stated in the previous section, VRML 1.0 has only limited support for describing an interaction and a behavior of the entities in the virtual world. This fact could be seen as the main limitation of the first VRML version, for complex, interactive environments design. In this section, we discuss the obstacles which have to be dealt with, while developing a multi-user distributed virtual reality system (MUDVRS). We also indicate how some of the problems can be solved already on the level of the environment description language.

MUDVRS must allow *concurrent multi-user access* to the world database. It should provide for keeping the overall consistency of the distributed data, and any changes made in the database should be managed in a *real-time*. The avatars representing the users (humans), should be treated as elements of the world database. In other words, all the time there should be information about location and orientation of all the mobile entities in the virtual environment. These parameters represents the minimum information needed to reconstruct the awareness in the virtual world.

5.1 Naive Implementation

The simplest and naive approach to implement above described requirements is to have each host to *broadcast* the location of each mobile entity that it maintains. These broadcasts are received by every host in the simulation, and are used to update their local copy of the world database. Although this approach works acceptably on small, dedicated networks, there are number of problems known.

The biggest problem with broadcasting is that every machine on the subnet has to receive and process large amount of data. This includes also those machines which are not participating in the simulation. To demonstrate the amount of the broadcast data, think of tracking the current location and orientation of every avatar in the virtual environment. Most likely these parameters change during every run of the simulation loop.

It is obvious that the amount of transferred data, network bandwidth and latency play very important role in the speed and scalability of such VR system. We don't have to accent that networks are used also for other purposes than distributed simulations. The first networked version of the computer game "Doom" worked in a broadcast mode; each participant constantly broadcast the current state of their avatar. This is one of the main reasons why many universities

and companies adopted the "NO DOOM" policy. In the next section we will look at the possible improvements of the implementation strategy.

5.2 Dead Reckoning

It appears that the most obvious problem in the keeping the consistency of the database using the broadcasting model is the amount of transferred data. This issue was already addressed by developers of the *Distributed Interactive System* (DIS), which has been developed for very specific applications – military simulations. The authors of DIS suggested solution – using technique called *dead reckoning* (DR). The idea behind DR is simple: instead of sending entity's location, a host sends a message that consists of the entity's location, time stamp and velocity vector. This can be done even for more complex **behaviors** then just a change of location. For example updating the description of a human body can be done by sending rotation and angles of selected joints.

The user controlled entity runs its own full simulation, and also runs the DR model for itself. It keeps track of the actual behavior and compares it with the one predicted by the last sent update message. When these two values differ more then predefined ε, it sends another update to synchronize the behavior of all its copies on all hosts participating in the simulation. This way the distributed database is kept synchronized with lower communication cost.

In addition the entity sends out messages carrying the information that the entity is still "alive". This is happening in predefined time intervals. It also serves the purpose of delivering the information about the state of the entity to a host, which joined the simulation after it has already started. If update message gets lost too many times or the entity (the host of the entity) leaves the simulation, the entity is considered dead. It is not a part of the distributed database anymore.

Updating the state and behavior of the entities based on the dead reckoning significantly reduces the traffic on the network. Unfortunately, this method does not change anything on the fact, that the data has to be broadcast. It is important to keep in mind that the broadcasting not only reduces the performance of the simulation, but also negatively influences the scalability of the system. Even with the dead reckoning employed, the limited bandwidth of the network remains an issue. There is only so many entities, which can broadcast their update messages through the network. Complex virtual environments can consist of hundreds or thousands of such entities.

Another improvement to the communication scheme for multi-user VR system is based on location sensitive filtering of the update messages. The principles of it are discussed in the next paragraphs.

5.3 World spatial subdivision and Multicasting

The attempt to reduce the cost of keeping the distributed database consistent, leads into farther more examination of the communication scheme (broadcasting). It is obvious that each entity in the distributed environment needs to track only those changes in the world database which take a place in a "near" surrounding area of the entity. Dividing the world into areas – *zones* is the idea behind the message filtering.

Each entity participating in the simulation has its *Area of Interest*(AOI), consisting of zones "visible" from its location. For each of the zones a multicasting group is created. There is a *multicast server* chosen for each group. When any host sends a message to the multicast server, it is distributed to all hosts that belong to the particular multicast group. This way the entity's host is receiving only those messages which carry information about changes in the area of interest of the particular entity. In order to make the multicasting work properly, zone to zone visibility has to be defined.

If an entity is mobile and capable of changing location, it is important to ensure that its area of interest is also updated. If a new zone is added into the AOI of entity, the host must

start to receive the messages corresponding to this zone. There are still open questions mainly related to the definition of the zones:

- How to divide the world into the zones?
- How to find the zones directly visible from the current zone?
- How to find the zone we are currently in?

Many different ways to solve these problems exist. Some might be better than the others. The way, how the scene is partitioned much depends on the character of the virtual environment and the type of the simulation. For example in DIS (section 5.2), the cells are hexagonal which is well suited for military simulation. For architectural walkthroughs though, hexagonal cells don't provide satisfactory results. It is due to not respecting the topology of the scene.

Previous experience with the scene subdivision led us to an idea to partition the scene accordingly to direct – *global visibility*. Since the investigation is not finished, we can not present any concrete results yet. At least we want to give a brief description of the main ideas employed in our method.

6 Global Visibility Algorithm

In this section we will give a quick overview of the algorithm for global visibility computation. The algorithm attempts to divide the scene into areas, where the direct visibility can be precomputed.

We define *scene* to be the bounding box of all objects (polygons) referenced in the world database. First a spatial subdivision of the scene is performed. During this phase the scene is divided into convex 3D volumes called *cells*. The spatial subdivision algorithm is based on Binary Space Partitioning (BSP). We subdivide the scene recursively by a plane which is defined by the largest opaque polygon in the current cell. Small details are not considered occluding and are ignored during the subdivision phase.

The result of the subdivision is a graph structure, where nodes represent the cells. Two nodes are interconnected by an edge, if the corresponding cells share a boundary. The *boundaries* (in case of 3D space) are 2D convex polygons explicitly constructed in each subdivision step. All cells whose representation in the graph structure are connected by edges are said to be neighbors.

We define *portal* as a non-opaque convex part of the boundary. It is clear that one can see from one cell to its neighbors only through the portals. To see from cell to another one which is not a direct neighbor is possible only through a sequences of portals. We construct portals as a convex decomposition of a difference of a boundary and union of polygons laying on the boundary. Result is an *Adjacency multi-graph*, where two nodes (cells) are connected by an edge if they share a boundary with at least one portal.

Using spatial subdivision information, we can examine whether two cells are mutually visible. The algorithm is based on proofing existence of a stubbing line through a sequence of portals between these cells. This is examined by an depth first search traversal through the adjacency graph. In each step of the traversal, the set of portals is tested for an existence of a stubbing line. This is done by transformation of the portal edges into Plücker coordinates system and solving a d-dimensional linear programming problem ($d \geq 6$). This process is called *static visibility determination*.

Result of this algorithm is a non-regular grid, where the cell to cell visibility is correctly determined. Each cell of the grid represents one of the multicast zones. All the cells which are said to be mutually visible create the AOI of an entity located in one of these cells.

So far we have implemented and tested spatial subdivision algorithm on Silicon Graphics machines. We are currently working on polygon set operations for portal enumeration and static visibility algorithm.

7 Conclusion

To conclude this paper we would like to recapitulate of what we consider to be essential for scene description language for multi-user distributed virtual reality simulation. We have showed that it is not enough to provide language only for the geometry definition of the entities. In VR simulation, there are many more factors which make the environment "real" for human participant. Very important is a behavior and interaction with the entities. The tools for describing arbitrary complex behavior and interaction should be available in the scene description language or its extension. This give us a possibility to describe somewhat "live" virtual environment, in which the human observer can participate. This is closely related to an extensibility issue of the language. The language should allow extensions in all aspects of functionality.

The authors of the language ought also to take into consideration the fact, that the hosts involved in the simulation are not connected into a local subnet, but can be placed all over the world. One could say it is more a problem related to a communication layer of the simulation. As we have shown the communication can be significantly reduced by adding more autonomy into the entities description (dead reckoning) and also by "smart" subdivision of the scene (multicast zones). If the scene is subdivided accordingly to its topology, the communication demands can be significantly reduced. In our research we promote the idea of supporting both formats:

- undivided world database
- world database divided into the multicasting zones

The scene subdivision reduces not only the amount of communication, but also can positively influence the memory requirements for storing the scene on each host. The time needed for the rendering of the scene subset is clearly shorter then dealing with the whole scene. Each host has to store and render only those parts, which are in AOI of its user.

We don't consider this paper to be a complete and exhaustive illustration of all the problems related to MUDVRS. We have chosen only those, which we believe are related to the language for the virtual environment scene description.

References

[1] Bell, G., Parisi, A., Pesce, M.: *The Virtual Reality Modeling Language.* Version 1.0 Specification. Silicon Graphics Inc., 1995.

[2] Fuchs, H., Kedem, Z., H.: *On visible surfaces generation by a priory tree structures.* Computer Graphics (Proc. SIGGRAPH '80), 14(3), July 1980, pp. 124–133.

[3] Stamoe, D., Roehl, B., Eagan, J.: *Virtual Reality Creations.* Waite Group Press, CA, 1993.

[4] Škrášek, J., Tichý, Z.: *Základy aplikované matematiky 3.* SNTL, Praha, 1990.

[5] Teller, S., J., Sèquin, C., H.: *Visibility preprocessing for interactive walkthroughs.* Computer Graphics (Proc. SIGGRAPH '91) 25(4), 1991, pp. 138–148.

[6] Teller, S., J.: *Computing the Antipenumbra Cast by and Area Light Source.* Computer Graphics (Proc. Annual conference Series), 1993, pp. 236–246.