Development of a Distributed Scene Toolkit Based on Open Inventor

Jan Pečiva

2002

Abstract

A distributed scene toolkit may be very useful for development of shared virtual environment applications, for example 3D multiplayer games, because time-sensitive distributed applications cause programmers many extraordinary consistency problems. In this thesis we will present the basis of such a toolkit, its design and implementation based on the popular Open Inventor library. The core idea is that one computer is the scene server and all others are the clients. The scene server manages the scene graph and can be used for viewing and modifying the scene as clients do. The client computers are like replica managers. They hold copies of the server's scene graph and make all the synchronization work. The toolkit implements algorithms for maintaining a scene consistency. The Network interface provides a convenient access to network functions and the whole toolkit should help programmers in the development of 3D graphics distributed applications.

Keywords :

distributed scene, distributed graphics, distributed virtual environment, computer supported cooperative work, virtual reality, scene graph, Open Inventor

Contents

1 Introduction **3**
- 1.1 Project Aims . . . 3
- 1.2 Open Inventor . . . 3
- 1.3 Usability . . . 4

2 Design **5**
- 2.1 Overview . . . 5
- 2.2 Introduction to Open Inventor Architecture . . . 5
- 2.3 General Design . . . 6
- 2.4 Modifications Handling . . . 7
- 2.5 Consistency Handling . . . 8

3 SoField Extensions **10**
- 3.1 Overview . . . 10
- 3.2 Synchronization of a Field's Value Over Network . . . 10
- 3.3 Process of Connecting of one Field to the Another . . . 11
- 3.4 Disconnect and Cancel Operations . . . 12
- 3.5 Public Network Connection Related Functions . . . 13
- 3.6 Additional SoField Internal Data . . . 14

4 SoFieldContainer Extensions **15**
- 4.1 Overview . . . 15
- 4.2 Synchronization of a Field Containers Over Network . . . 15
- 4.3 Process of Connecting of one Container to the Another . . . 16
- 4.4 Disconnect and Cancel Operations . . . 17
- 4.5 Public Network Connection Related Functions . . . 18
- 4.6 Additional SoFieldContainer Internal Data . . . 18

5 SoGroup Extensions **19**
- 5.1 Overview . . . 19
- 5.2 Synchronization of Graphs Over Network . . . 19
- 5.3 Process of Connecting of one Scene Graph to Another . . . 21
- 5.4 Disconnect and Cancel Operations . . . 21
- 5.5 Public Network Connection Related Functions . . . 22
- 5.6 Additional SoFieldContainer Internal Data . . . 22

6 SoDS Global Class **23**
6.1 Overview . . . 23
6.2 Public SoDS API . . . 23
6.3 Connecting Other Computers . . . 26

7 Network Layer **27**
7.1 Overview . . . 27
7.2 Handling of Scene Modifications . . . 27
7.3 General Design . . . 28
7.4 Mechanism of Sending Messages . . . 29
7.5 Handling the Scene Consistency . . . 30
7.6 Limited-Bandwidth Networks . . . 30
7.7 Process of Receiving Messages . . . 30
7.8 Abstract Message Classes . . . 30
7.9 Network Messages . . . 32

8 Results **39**
8.1 Overview . . . 39
8.2 Testing . . . 39
8.3 Conclusion . . . 41
8.4 Future Work . . . 41

Chapter 1

Introduction

1.1 Project Aims

This work aims to develop a toolkit for distributed shared virtual scenes. The work connects the world of 3D graphics and the world of distributed systems. The world of 3D graphics is represented by Inventor toolkits. Open Inventor is the de facto standard for visualization in the scientific and engineering community. On the other side, the world of distributed systems gives some principles for designing a toolkit for distributed 3D graphics.

The toolkit for distributed 3D graphics may be very useful, especially with internet and high speed local networks. On the one hand, vertical distribution can be used for parallel processing of a scene increasing performance by distributing tasks among computers. On the other hand, horizontal distribution for applications means that users can work together, cooperatively, possibly over large distances. Successful areas are Distributed Virtual Environments (DVEs), Computer Supported Cooperative Work (CSCW) and multiplayer 3D computer games running over networks.

This work presents the design of such a toolkit giving the programmer an easy-to-use API for programming distributed 3D graphics applications. We will present the design of Network Layer and the network protocol, and their integration into Coin, a free library that implements Open Inventor 2.1 API.

1.2 Open Inventor

Open Inventor is an OpenGL-based, retain-mode, 3D model scene-graph rendering and interaction library, which has became de facto standard in the scientific and engineering community. The first version of Open Inventor was released by SGI in 1991 and after that it has been developed for another seven years. Because of its well designed API Open Inventor still lives and is developed by other companies. In 2001, TGS released a new thread-safe version and opened the world of multiprocessing for Open Inventor. My work is a step in a different direction, a step towards distributed processing. I have used the word 'step', because this area is too large and development of a full toolkit will take a long time. So, this thesis describes the basis of such a toolkit.

1.3 Usability

With growing importance of networks and high-speed internet, the importance of the distributed systems also grows. People want to be connected together. They may want not only to communicate together through network, but to see one another and also to share the same virtual environment with others. A great development has been made on 3D multiplayer computer games which are typical distributed 3D applications very popular today.

Developing of a distributed 3D application is a hard problem. The network latency among the computers is usually tens of milliseconds, a typical value is 40ms. In the 40 milliseconds the application can render four frames (when 100 FPS) and still may not have some actual scene data from the other computers. So, it is a question how to perform, for example, collision detection on the objects we do not know their actual position. This leads to complex problems of relaxed consistency models. This inspired me to develop a library that will make a programmer's life easier. The library, presented here, should be useful in two points:

- It removes from a programmer all low-level network troubles and handles most of communication for him.
- It automatically synchronizes the scenes among computers while keeping the scene consistency.

Chapter 2

Design

2.1 Overview

The purpose of this work is to design Open Inventor API extension and implement it into Coin, a free library that implements Open Inventor API. So, in this chapter, we will describe the theoretical basis on which the work is built, then the concept of the design. The implementation is presented in succeeding chapters.

2.2 Introduction to Open Inventor Architecture

A scene in Open Inventor consists of nodes. Each node represents a geometry, property or grouping object. Because of grouping objects, we can organize nodes in a hierarchical structure called a directed acyclic graph. We will call it simply 'scene graph'. The scene graph is an representation of the scene in Invetor. Note, that if we use the term 'scene graph', it may mean a whole scene graph or any of its subgraphs or just a single node.

Nodes are C++ objects and there are three types of nodes:

- Shape nodes represent 3D geometry.
- Property nodes represent appearance and other qualitative characteristics of the scene.
- Group nodes are containers that collects nodes into graphs.

Each node usually contains some data items called fields. Fields are data encapsulation providing many mechanisms for Inventor application, like automatic read and write to files, detection of changes and connecting one field to the another.

Performing an action on a scene graph means to traverse an Inventor's scene graph. Typical traversing is done from top to bottom and from right to left. This way is usually performed rendering actions. There are also actions for writing the scene to the file, for picking objects along a ray, for computing the bounding box of the scene graph and some others.

Fields can be connected to the other fields. If a field is connected to the other field, it means one is master and other slave and each time the master is modified, the slave is notified about the modification to re-synchronize its value.

The notification mechanism is very important in Inventor. It is an advanced topic that is not completely important for Inventor users. But it is important for programmers extending

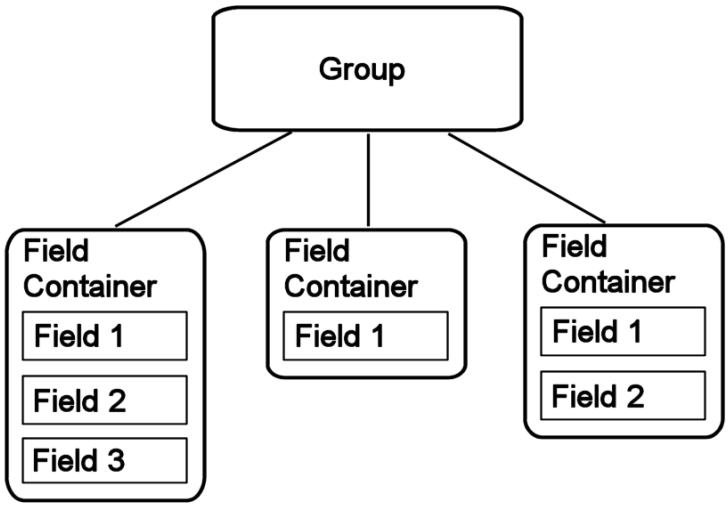

Figure 2.1: A simple Inventor graph

Inventor's functionality and also for this work. Whenever user modifies something in the scene graph, the notification is started. For example, when user modifies the field containing the geometry of some scene object, notification starts. The node within field is placed is notified and notification propagates up to the root of the scene graph. When the root node of the scene graph is notified, the scene is scheduled to be rendered to update the picture on the screen. So, the scene is rendered only, if something is changing in the scene. Another advantage is, that rarely changing parts of the scene graph we can use some caching mechanisms, like render-caching in OpenGL display lists. And if some modification happens, the notification mechanism tell the cache to update its content.

More informations about Inventor are in The Inventor Mentor book [8], some additional informations are in The Inventor Toolmaker [9].

2.3 General Design

The question is What is a distributed shared scene graph? It is a scene graph shared among some computers. Each computer keeps its own copy of the scene graph, which must be kept synchronized with the others. The best solution is to implement the distributed shared scene graph to behave as if it was stored in distributed shared memory (DSM) [1]. In this case it will be nearly transparent for the programmer using such a scene graph whether he is writing a local or distributed application. However, for realization of DSM we have to use sequential consistency for all reads and writes to the DSM on the all computers. This is impossible to be done with sufficient performance, because of network latencies among computers. Therefore some less restrictive consistency model must be used.

My work, however, did not take the route of complex consistency models, but I made some restrictions to the concept of distributed shared scene graphs. If we make all shared scene graphs on all computers only readable (not writeable) except one that we make readable and writeable, we find a system that is easy to gain consistency for. In my work I will call such a configuration

'master-slave', because there is one writeable master scene graph and the rest are read-only slave graphs keeping everything synchronized with the master. Unfortunately, it looks like a great limitation to be able to modify the scene graph on one computer only. But we can use a simple workaround. We can send a modification request from slave to master scene graph each time we want to modify slave. And the master replies whether he has modified the value or not. A reason for refusing the modification may be, for example, if we are utilizing collision detection, to try to move one object into another. The collision detection is usually centralized, because only the 'scene server' knows valid positions of all objects in the scene. Until the reply, the slave scene graph asking for the modification may contain both — old or new value, depending on programmer's choice.

Master-slave architecture perfectly fits the original Open Inventor design where we can find connect operation from one field to another. In Inventor, connecting one field to another means one is the master and when its value is changed, all slaves are notified about this to re-synchronize their values. I will use same idea for my network connection. Moreover, I will extend the idea of connecting fields to connecting nodes and also graphs. We can not see anything like this in Inventor API because connecting graphs only has its meaning when it is over a network. Locally it is handled by multiple references to nodes.

The question is For what can be master-slave architecture be used for? General answer is: For each client-server designed application. These are most of today's 3D games and VR systems. So, it is not a problem. Also, special usage may be distributed rendering of very complex scenes or another scene processing.

2.4 Modifications Handling

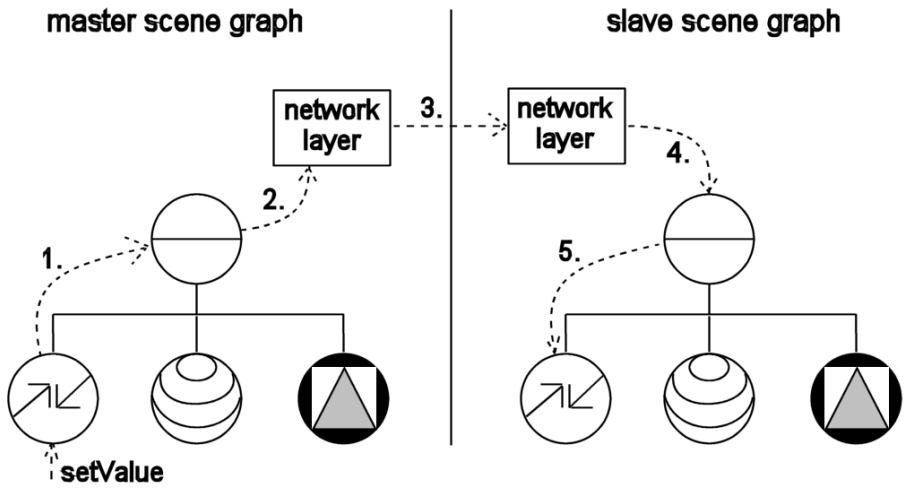

Figure 2.2: Handling of a scene modification

In figure 2.2 there is a simple example showing the way modifications are going to keep the slave scene graph synchronized. There are two scene graphs on two computers. One graph is master, second is slave. The scenes are composed of four nodes — one separator and three its children, first from left is some transform node, then some appearance node and right node is shape. Master-slave relation, in this example, is made by connection between the separator in

the master scene graph and the separator in the slave scene graph. The example is showing the way the transform node modification is going to keep slave scene graph synchronized with the master scene graph:

1. The scene is modified and the notification starts. In this stage when separator is notified, it realizes that it has some 'network-slave' connections.
2. The separator notifies Network Layer about the modification and Network Layer does the rest.
3. The modification is stored in a network stream. The stream is sent and then received by another computer. This can last some tens of milliseconds.
4. The modification is read from the stream and sent to the slave separator.
5. The separator updates the value of the slave transform node.

2.5 Consistency Handling

Even though I am not using any consistency models in my work yet, there are some consistency problems. The problems come from generic Inventor behavior, because a scene graph is not always consistent.

For example, there are different nodes for storing geomery coordinates (SoCoordinate3) and texture coodinates (SoTextureCoordinate2). When an user is modifying geometry of some scene object by adding some coordinates to SoCoordinate3 node and textures are used for the object, the scene is inconsistent until texture coordinates are also modified to fit the number of coordinates in SoCoordinate3 node. A rendering of such inconsistent scene may cause the geometry of the object to be not rendered and on weak implementations it may cause an application to crash. This problem is not important on non-distributed scene graphs, because it is simply a programmer's mistake to render an inconsistent scene. But if we are working with the distributed scene graph, the master is producing a stream of modifications and slaves are receiving the stream and applying the modifications to their scene graphs. But because of the network communication, some modifications may yet arrived, some may be on the way. In our example, the SoCoordinate3 modification may be yet received, but the SoTextureCoordinate2 one may be still on the way. Applying the SoCoordinate3 modification without SoTextureCoordinate2 will break the scene consistency. This problem must be avoided.

We can use two equal solutions: To send modifications in chunks and ensure that applying all messages in the chunk at once does not break the consistency. The second solution is to put a special message to a stream sent to slave to tell it 'at this moment the scene is consistent', so it can apply all received modifications.

For finding a solution we need to answer for a simple question: When can we be sure that the scene is consistent in a properly working Inventor application? Generally never, because a programmer may use some very strange programming techniques. But we will expect that the scene is consistent at least just before and after the scene is rendered. This is true in the most of cases. So, each time before and after the scene is rendered, Network Layer takes all master scenes modifications and sends them in one great chunk to all slaves. All the slaves take the modifications before and after scene graph rendering and update their scene graphs.

This solution is sufficient in the most of cases. For special application demands it is possible to customize this behavior, for example, to perform updates more often. More details about handling network messages and the scene consistency will be in the chapter 7 about Network Layer.

Chapter 3

SoField Extensions

3.1 Overview

All extensions of the SoField concern the synchronization of fields over network. They introduce some basic principles that will be used and extended later by the SoFieldContainer and SoGroup classes (chapters 4 and 5).

Connecting one field to the other means to make one slave and the other master. The master has always a valid value and all modifications to its value are sent to all its slaves. If a user wants to modify a slave field, the field must ask the master field for validating the modification. This a powerful solution that make it easy for a programmer to keep the scene consistency up.

3.2 Synchronization of a Field's Value Over Network

For updating values of slave fields, we must use a little bit different principle than Inventor uses. Because of network latency we will not send a notification to slaves to invalidate their values and later during read operation perform real updating of their values. Instead, we will send a new value immediately.

The process of a modification at the master consists of five steps:

1. An user modifies the field's value.
2. The notification is performed (on the local scene graph).
3. The check for field's network connections is performed; if there are no slaves connected to this field then return.
4. The check is performed whether the value has been really changed; if not, then return.
5. Call to Network Layer for updating all network slave fields.

First two steps are generic Inventor behavior. In third step there is a test for network-slave connections. If none, then can be nothing done and the field behaves like genuine Inventor field. At fourth step we proceed in the case that field value has been really changed. So, touching or writing same value has no effect on network connections. At step five all necessary information

are passed to Network Layer. Network Layer will send 'new value' message to all slave fields. More detailed description of Network Layer will be described in the chapter 7.

There is a problem how to transfer fields value. We need general principle for all types of field. This can be very easily done by built-in mechanism for writing and reading fields. So, we use SoInput and SoOutput classes with buffers set to memory. Then we read or write fields value. There is a problem with amount of transferred data. If a field is modified, we are transferring whole field content. This is not a problem in most cases. But sometimes we are modifying large multiple value fields or texture data. There can be done additional research to minimize amount of the transferred data, to avoid, for example, the whole texture being transferred if its single pixel is modified.

The process of updating slave's value:

1. Network Layer decodes message and sends it to target field.
2. Field checks connection status; if connection is no longer active then return.
3. Update field's value.

Step 2 is especially important because connection may be closed on slave's side, but it take some time to deliver disconnect request to the master and also there may be messages on the way. So, all messages received after disconnect operation and before disconnect acknowledge message from master must be ignored.

Yet, there are some additional extensions related to SoField synchronization based on connecting scene graphs, instead of single fields. They will be described in the chapter 4 and 5 that will explain SoFieldContainer and SoGroup extensions.

3.3 Process of Connecting of one Field to the Another

There are some differences between Inventor's connectFrom and our network version. First noticeable difference is that value of slave field is not immediately available. New value is present at the very moment connection is acknowledged from master's side. Until this, the old field's value is present. Testing for invalid field's value can be made by function netInvalidValue(). Another difference is that connection may be refused if target field does not exist, has been destroyed or is of different type. If connection is refused, field's value are left unchanged. Fields must be exactly of same type. No conversions are supported, but it may be not a problem to implement this if there will be a serious reason for it.

The process of the connecting:

1. The slave sends the connection request to the master.
2. The master accepts/refuses the connection, the answer is sent to the slave.
3. The answer received by the slave, the connection is established/canceled.
4. If the connection is established, the field's value is updated.

Connecting functions:

```
void netConnectFrom(int ip, SoField *master)
void netConnectFrom(int ip, SoFieldContainer *container, const char *fieldName)
void netConnectFrom(int ip, SoFieldContainer *container, SbName &fieldName)
void netConnectFrom(int ip, SbName &nodeName, SbName &fieldName)
void netConnectFrom(int ip, SbName &nodeName, const char *fieldName)
void netConnectFrom(int ip, const char *nodeName, SbName &fieldName)
void netConnectFrom(int ip, const char *nodeName, const char *fieldName)
```

Connection request is specified by IP address and some field's identification on target computer. Easiest and most safe field's identification is by node and field name (case 4 to 7). Field's identification by its pointer (case 1) or container's pointer and field's name (case 2, 3) is potentially dangerous. We must be sure that no-one destroys an object at target computer pointed by our pointer, otherwise application may crash.

3.4 Disconnect and Cancel Operations

Disconnect operation executed on a slave field disconnects it from its master. On the other side, Cancel operation, performed on master, disconnect all its slaves from it. Both operations are implemented in a way that it is transparent for user whether he uses local disconnect or network disconnect or cancel operation.

After performing disconnect or cancelConnection function field is behaving as if it was unconnected. It ignores all network messages that may be 'on the way' before disconnect acknowledge comes. Then slave field is sure that no more messages come from the master and completely deletes the connection.

The process of the disconnecting:

The slave side after calling netDisconnect function:

1. The disconnect request is sent to the master.
2. The reference counter of the slave's container is incremented.
3. The slave field marks the connection as 'closing' and any incoming messages through the connection are ignored.
4. The field's behavior is restored to be 'like an unconnected field'.

The master side after receiving the disconnect request:

5. The network connection at the master field is removed.
6. The disconnect acknowledgement is sent to the slave.

The slave side after receiving the disconnect acknowledgement:

7. The slave removes the connection and unreferences its container.

The process of canceling a connection is the same, only master and slave roles are exchanged.

Important point of disconnecting is incrementing and decrementing of container's references counter. It is very important for field that has not completely closed connections not to be destroyed. This is the easiest way to handle this. The danger comes from network messages that must not be sent to non-existing fields otherwise application may crash.

Disconnect function prototypes:

```
void netDisconnect()
void netDisconnect(int ip, SoField *master)
void netDisconnect(int ip, SoFieldContainer *container, const char *fieldName)
void netDisconnect(int ip, SoFieldContainer *container, SbName &fieldName)
```

CancelConnection function prototypes:

```
void netCancelConnection(int ip, SoField *field)
void netCancelConnection(int ip, SoFieldContainer *container,
    const char *fieldName)
void netCancelConnection(int ip, SoFieldContainer *container,
    SbName &fieldName)
void netCancelConnections()
```

For convenience there is one function for closing all network connections:

```
void closeNetwork()
```

3.5 Public Network Connection Related Functions

`bool netInvalidValue()`
: The function returns true in the case that the field has been connected but value has not been updated yet. Until new value comes, old value of the field is present.

`bool hasNetConnections()`
: The function returns true, if the field is connected through the network, or someone is connected to it through the network.

`bool isNetMaster()`
: The function returns true if some field is connected to this field over network.

`bool isNetSlave()`
: The function returns true if this field is connected to some field over network.

`int getNetMasterNum()`
: It returns the number of connections this field is connected from. Currently, returned value can be only 0 or 1, no netAppendConnection() is implemented yet.

`SbBool getNetMasterConnections(SoNetConnectionList &list)`
: In list it returns the list of connections this field is connected from.

`const SoNetConnectionList* getNetMasterConnections()`
: It returns the list of connections this field is connected from.

`int getNetSlaveNum()`
The function returns the number of fields that are connected to this field.

`SbBool getNetSlaveConnections(SoNetConnectionList &list)`
In list it returns the list of connections to this field.

`const SoNetConnectionList* getNetSlaveConnections()`
The function returns the list of connections to this field.

3.6 Additional SoField Internal Data

Because of extending of functionality we need also to add some data in the SoField class. However, amount of the data must be minimized because SoField objects are ubiquitous in Inventor applications.

Our solution increased the size of SoField class only by 8 bytes (on 32-bit architectures), although there are four data items that we need to be stored somewhere around SoField class:

`netMasters`
List of connections to the masters.

`netSlaves`
List of connections to the slaves.

`currentStream`
Pointer to the buffer where actual field's content is stored.

`scheduledMsgs`
Internal pointer used by Network Layer.

Most painful are first two items, because lists are classes and take a lot of memory even if they are empty. However, all Inventor implementations, as far as I know, do not store connection related informations directly in SoField class, because most of fields do not have any connections. Rather, if connection informations need to be stored for a certain field, additional memory is allocated. So, we can place our lists to such on-demand allocated memory. The resting two items — both pointers taking 4 byte each at 32-bit architectures, should be placed directly in SoField class. The reason is that if user connects large scene graph, all fields at the scene graph need to use this two data items. If we place the items to the on-demand allocated memory, all fields in the connected graph will allocate it, so we will waste too much of memory.

Chapter 4

SoFieldContainer Extensions

4.1 Overview

SoFieldContainer extensions concern synchronization of all the container's fields. This is a necessary step to synchronizing whole scene graphs as it will be described in the chapter 5 about SoGroup extensions.

The connecting of one SoFieldContainer the other means to make one slave and other master. All we need is to synchronize all fields of the SoFieldContainer. So, fields of the master container has always valid values and fields of the slave container is synchronized with master's values. As we can see, connecting one container to the other one has same effect as connecting all slave container's fields, one by one, to their master fields, but the connecting of the SoFieldContainer is more convenient and saves memory resources.

4.2 Synchronization of a Field Containers Over Network

If we are connecting some large structures, we need to use some general mechanism for getting modifications from the master structure and sending them through network to all slaves. In well designed Inventor API we can find easily a solution. We can use notification to detect that something has been changed and also to locate the source of notification. In the case of connected container, notification source is, in most of cases, one of its fields.

Process of modification on master side:

1. An user modifies some field's value.
2. The field's container is notified about the modification.
3. If the container does not have any network-slave connections, do nothing and continue in the notification.
4. Locate the source of the notification; this information is accessible through SoNotList class passed as a parameter while notifying.
5. If the notification source is a field, tell the field about all network-slave connections at this container.

6. Continue in the notification (go to the step 3).
7. Check whether value of the field has been really changed; if not, then return.
8. The modified field call Network Layer to send the 'new value' message through all network-slave connections.

The containers process of synchronization is the extension of field's synchronization process (see the section 3.2). We are using notification to detect and locate modified field and then we pass all connection-related data to the field. After notification is done, the field calls Network Layer for send 'new value' message through all network-slave connections, this include the field's connections and also connections of its container.

There is a problem, how to identify fields in the container for sending 'new value' messages. We are simply using field's name string, because field's name is unique in one container. A small disadvantge of this is increased network bandwidth. Another solution can be to use an integer giving the index of the field that has been modified. However, using field's names is the most safe solution among all Inventors, therefore it is used.

Process of modification on slave side:

1. Network Layer decodes the message and send it to the 'slave container'.
2. The container checks the connection status; if the connection is no longer active, then return.
3. The container finds target the field.
4. The value of the target field is updated.

On the slave side we are using nearly same process as field's modifications do (see the section 3.2). Only difference is that 'new value' message is passed to a container and the container responsible for locating and updating of the appropriate field.

4.3 Process of Connecting of one Container to the Another

A process of connecting of containers is nearly the same as connecting of fields (see the section 3.3). Containers connecting together must be exactly of same type. After connectFrom function call, all fields in the container remain with their old values until the connection is fully established by an acknowledge message. At the very moment connection is established, all fields get a new value. There are consistency reasons for this. If the connection is refused, all fields are left unchanged.

Process of the connecting:

1. The slave sends connection request to the master.
2. The master accepts/refuses the connection, the answer is sent to the slave.
3. The answer is received by the slave, the connection is established/canceled.
4. If the connection is established, values of all fields are updated.

Function prototypes:

```
void netConnectFrom(int ip, SoFieldContainer *fc)
void netConnectFrom(int ip, const char *name)
void netConnectFrom(int ip, const SbName &name)
```

4.4 Disconnect and Cancel Operations

Disconnect and cancel operations on containers have same meaning as their field's versions (see the section 3.4), here we will speak only about differences and extensions. At the moment of disconnection updating of container's fields are stopped and their values are no longer changed by the connection. Even, if container is about to destroy, it must wait for disconnect acknowledge from all its network 'friends'.

Process of disconnecting:

The slave side after calling netDisconnect function:

1. The disconnect request is sent to the master.
2. The reference counter of the slave container is incremented.
3. The slave container marks the connection as 'closing' and any incoming messages through the connection are ignored.
4. The container's behavior is restored to be 'like an unconnected container'.

The master side after receiving the disconnect request:

5. The network connection at the master container is removed.
6. The disconnect acknowledgement is sent to the slave.

The slave side after receiving the disconnect acknowledgement:

7. The slave removes the connection and unreferences itself.

The process of canceling a connection is the same, only master and slave roles are exchanged.

Disconnect function prototypes:

```
void netDisconnect()
void netDisconnect(int ip, SoFieldContainer *fc)
```

CancelConnection function prototypes:

```
void netCancelConnections()
void netCancelConnection(int ip, SoFieldContainer *fc)
```

For convenience there is one function for closing all network connections including also all connections of container's fields:

```
void closeNetwork()
```

4.5 Public Network Connection Related Functions

`bool netInvalidValue()`
The function returns true in the case that the field has been connected but value has not been updated yet. Until new value comes, old value of the field is present.

`bool hasNetConnections()`
The function returns true, if the field is connected through the network, or someone is connected to it through the network.

`bool hasFieldsNetConnections()`
The function returns true, if any of the container's fields has network-master or network-slave connections.

`bool isNetMaster()`
The function returns true if some field is connected to this field over network.

`bool isNetSlave()`
The function returns true if this field is connected to some field over network.

`int getNetMasterNum()`
It returns the number of connections this field is connected from. Currently, returned value can be only 0 or 1, no netAppendConnection() is implemented yet.

`SbBool getNetMasterConnections(SoNetConnectionList &list)`
In list it returns the list of connections this field is connected from.

`const SoNetConnectionList* getNetMasterConnections()`
It returns the list of connections this field is connected from.

`int getNetSlaveNum()`
The function returns the number of fields that are connected to this field.

`SbBool getNetSlaveConnections(SoNetConnectionList &list)`
In list it returns the list of connections to this field.

`const SoNetConnectionList* getNetSlaveConnections()`
The function returns the list of connections to this field.

4.6 Additional SoFieldContainer Internal Data

There are two data items added to SoFieldContainer class:

`netMasters`
List of connections to the masters.

`netSlaves`
List of connections to the slaves.

Because only few containers usually have network connections, there are only two pointers placed directly to the SoFieldContainer class and when we need to store some connections in the container, we allocate the appropriate list.

Chapter 5

SoGroup Extensions

5.1 Overview

SoGroup extensions is based on the SoFieldContainer ones (see the chapter 4), because SoGroup is derived from SoFieldContainer. Additionally, SoGroup extends the behavior of SoFieldContainer to take care about its children nodes.

If one group is synchronized over network, all its fields and children must be synchronized. The fields are synchronized in the manner presented in the chapter 4 about SoFieldContainer extensions. The child adding and removing from the group must be handled. Moreover, all children must be synchronized in the same way as they alone have been connected. In the case of SoGroup child it means recursive behavior. So, whole scene graph under our heading group is synchronized.

5.2 Synchronization of Graphs Over Network

If a SoGroup object has some slave connection, we need to synchronize whole scene graph under the SoGroup. So, we utilize Inventor notification mechanism and during this process we need to find 'the notification path', or simply the path from the connected SoGroup object to the initiator of the notification. Then, in the case of a scene graph structure modification, we have to send the path from the connected SoGroup to the modified object and the data related to the type of modification. In the case of field's value modification, we are sending the path to field's container, the field's name and new value data.

Getting a notification path is a problem because each Inventor can implement the notification in a little bit different way. So, I have tried to make independent implementation. It simply make calls to special notification handling functions for each notified object in the scene graph when the object is entered and left by notification. This can be easy done by modifying SoBase and SoField notify method, because most of objects in the scene graph are derived from them. The two handling functions, one when object is entered, one when leaving, keep the record of the notification path. Actually, for the best performance, nothing is done until we find some node with network connections. Then, when notification is going back and stack is popped, the notification path is built.

If a field has been modified, after notification it receives list of notification paths. These paths represents paths from nodes, containing network-slave connections, to the container of

the modified field. Usually, only one notification path is received, but it is not a general case, because some nodes on the notification path can also have network-slave connections or because of multiple references. So, the field takes all network-slave connections from all connected nodes found during the notification and it sends 'new value' messages through them. Three things is needed for creating such messages: the notification path, the name of the field and the new value data. The 'new value' messages, generated by the SoField and SoFieldContainer extensions (chapters 3 and 4), is handled together with these messages until entering Network Layer. Then different message formats are used.

When the scene graph structure is modified, some little bit different things happen. After notification process is done, group object, whose children list has been modified, receives list of notification paths. The rest of work depends on the type of modification. Simplest operation is removing a child. All we need is to send the path to the group whose children are modified and index of removed child. If we are inserting a single node, we must send the type of inserting node and values of all its fields. In the case of inserting a large scene graph we must send whole scene graph structure and values of all fields in the scene graph.

There should be implemented a special mechanism of handling multiple references to be correctly transferred from the master scene graph to all slave graphs. This is not currently implemented in our 'research' code, because it will need deeper digging in the Inventor's source code to get an efficient solution. Generally, we should test all nodes in the graph, that is about to be inserted, for network-slave connections. If we find some connections, we should compare IP addresses of the connections and connections of connected scene graph we are inserting to. The same IP addresses means multiple references and we must handle them by special network messages when connecting and disconnecting.

The process of the modification on the master side:

1. An user modifies something in the scene graph headed by a group with network-slave connections.
2. The notification is performed and the list of paths to nodes with network-slave connections is returned.
3. The modification is handled.
4. Call Network Layer for sending appropriate messages to all network-slaves graphs.

The step 3 depends on the type of the modification:

- Field modifications are handled by preparing the 'new value' message data for sending.
- Removing something from the scene graph are handled simply by getting index of the removed child.
- Inserting something to the scene graph are handled in three steps:
    1. Multiple connection checking of each node in the inserted scene graph.
    2. Storing inserted scene graph structure in the stream in the order to reconstruct it on all slaves.
    3. Preparing to send all fields values in the inserted scene graph.

The process of the modification on the slave side:

1. Network Layer decodes the message and send it to the appropriate SoGroup object.
2. The group object checks the connection status; if the connection is no longer active, then return.
3. The group finds the target object for the modification by the path information from the message
4. The modification is applied.

5.3 Process of Connecting of one Scene Graph to Another

The connecting one scene graph to another is composed of three steps. At first, we establish the connection between the heading node of the master scene graph to the heading node of the slave scene graph. At second, we must copy the master's scene graph structure to the slave. At third, we must copy values of all fields from the master to the slave. The second and the third step must be made as one atomic operation because of consistency reasons. How this is handled will be described in the capter 7 about Network Layer.

The process of the connecting:

1. The slave sends connection request to the master.
2. The master accepts/refuses the connection, the answer is sent to the slave.
3. The answer is received by the slave, the connection is established/canceled.
4. If the connection is established, the scene graph structure is synchronized and values of all fields are updated

The function prototypes are same as for SoFieldContainer:

```
void netConnectFrom(int ip, SoFieldContainer *fc)
void netConnectFrom(int ip, const char *name)
void netConnectFrom(int ip, const SbName &name)
```

5.4 Disconnect and Cancel Operations

Disconnect and cancel operations are using same algorithms as their SoFieldContainer's versions (see the section 4.4). At the moment of the disconnecting, although the connection is not completely deleted, scene graph behaves like unconnected and no changes coming through the network are applied. Note, that when we disconnect a scene graph, some parts of it may still be connected. Because of multiple references they may still be parts of other connected graphs.

The process of the disconnecting:

The slave side after calling netDisconnect function:

1. The disconnect request is sent to the master.
2. The reference counter of heading group is incremented.
3. The slave container marks the connection as 'closing' and any incoming messages through the connection are ignored.
4. The scene graph's behavior is restored to be 'like an unconnected scene graph'

The master side after receiving the disconnect request:

5. The network connection at the master heading group is removed.
6. The disconnect acknowledgement is sent to the slave.

The slave side after receiving the disconnect acknowledgement:

7. The slave removes the connection and unreferences the heading group.

The process of canceling a connection is the same, only master and slave roles are exchanged.

Disconnect function prototypes (same as for SoFieldContainer):

```
void netDisconnect()
void netDisconnect(int ip, SoFieldContainer *fc)
```

CancelConnection function prototypes (same as for SoFieldContainer):

```
void netCancelConnections()
void netCancelConnection(int ip, SoFieldContainer *fc)
```

For convenience there is one function for closing all network connections of whole scene graph including all node's connections and all fields connections:

```
void closeNetwork()
```

5.5 Public Network Connection Related Functions

The functions are completely the same as in the SoFieldContainer, see the section 4.5.

5.6 Additional SoFieldContainer Internal Data

The functions are completely the same as in the SoFieldContainer, see the section 4.6.

Chapter 6

SoDS Global Class

6.1 Overview

SoDS (Distributed Scene) global class gives a programmer an access to the network functions. It is Inventor manner to not leave global functions alone, but to group them to global classes like SoDS class (scene database) or SoSensorManager class (sensors).

SoDS class is a public interface to the internal Network Layer functions. Network Layer will be described in the chapter 7.

6.2 Public SoDS API

- initialization/finalization functions:

```
void init(bool initNetwork = true)
void done()
bool isNetworkOpen()
bool openNetwork()
void closeNetwork()
```

- connection management:

```
void connectComputer(uint32_t ip)
void disconnectComputer(uint32_t ip)
void dropAllNonEstablishedConnections()
void disconnectAllComputers()
```

- processing functions:

```
void processNetwork()
void processReceivedMessages()
void performSending()
```

- getting network informations:

```
const SbString& getHostName()
uint32_t getHostIP()
SoMFString* getComputerNames()
SoMFUInt32* getComputerIPs()
```

- callbacks:

```
void setComputerListChangedCallback(SoDSComputerListChangedCB *f,
    void *userData = NULL)
void setConnectQueryCallback(SoDSConnectQueryCB *f, void *userData = NULL)
void setUserMsgCallback(SoDSUserMsgCB *f)
```

The function descriptions:

- initialization/finalization functions:

`void init(bool initNetwork = true)`
: Initializes all internal data structures. If initNetwork parameter is true, it also initializes network by calling openNetwork function.

`void done()`
: Closes network and frees all allocated resources.

`bool openNetwork()`
: Initializes network. So, other computers can connect to this computer and this computer can make connections to the others.

`void closeNetwork()`
: Closes network. All connected fields and containers in all scene graphs in the application are disconnected. Then connections to the other computers are closed and network resources are freed.

`bool isNetworkOpen()`
: Returns true, if network has been successfully initialized by openNetwork().

- connection management:

`void connectComputer(uint32_t ip)`
: Establish connection between this computer and computer identified by ip. The connection is not established immediately. There must be done some connecting work. After everything is done, connection is append to the global connection list.

`void disconnectComputer(uint32_t ip)`
: Disconnect computer identified by ip. At first, all fields and containers network connections are disconnected. Then connection is safety closed.

`void dropAllNonEstablishedConnections()`
: All connections which are currently trying to be established or their establishing process is in the progress are dropped.

`void disconnectAllComputers()`
: Connections to all computers are closed. It is nearly same operation like calling disconnectComputer() for all the connections.

- processing functions:

  `void processNetwork()`
  Performs sending and receiving network operations. It is equal to call processReceivedMessages() and performsending().

  `void processReceivedMessages()`
  All messages received by network and waiting to be processed and applied to the scene graph are processed.

  `void performSending()`
  All messages waiting to be send are sent.

  `void sendMessage(uint32_t ip, int msgId, void *buf, int size)`
  Sends user message to computer identified by IP. The computer must be in the global connected computer list. The message, specified by a pointer and a size, is marked by msgId parameter and sent to the target computer.

- getting network informations:

  `const SbString& getHostName()`
  Returns host name of this computer.

  `uint32_t getHostIP()`
  Returns IP address of this computer.

  `SoMFString* getComputerNames()`
  Returns global SoMFString field containing names of all connected computers.

  `SoMFUInt32* getComputerIPs()`
  Returns global SoMFUInt32 field containing IP addresses of all connected computers.

- callbacks:

  `typedef void SoDSComputerListChangedCB(void * userdata)`

  `void setComputerListChangedCallback(`
  `SoDSComputerListChangedCB *f, void *userData = NULL)`
  Sets the callback that is called each time global computer list is modified. The computer list is changing each time some computer is successfully connected or disconnected.

  `typedef bool SoDSConnectQueryCB(uint32_t ip, const char* hostName,`
  `void *userData, int userDataSize, int &errorCode)`

  `void setConnectQueryCallback(SoDSConnectQueryCB *f,`
  `void *userData = NULL)`
  Sets the callback that is called each time some computer tries to connect to this computer. User is allowed to accept or denies to the computer to connect. If no callback is registered, all computers can connect to this computer.

  `typedef void SoDSUserMsgCB(int id, uint32_t ip, void *buf, int size)`

  `void setUserMsgCallback(SoDSUserMsgCB *f)`
  Sets the callback that is called each time some connected computer sends user message. If no callback is registered, message is ignored.

6.3 Connecting Other Computers

To have fully established connection between two computers requires some conditions to be satisfied:

1. To get the agreement to connect from the other computer.
2. To have a safe network connection.
3. To be sure about the same types assigned to Inventor classes on both computers.
4. To have synchronized Inventor's time on both computers.
5. To pass the measurement of the network bandwidth and ping times.

In first step we are asking another computer whether it accepts our connection. There is an user defined callback on the target computer and user may refuse the connection. If the connection is accepted we need to establish safe network connection. To have the safe network connection means not to worry about lost packets and incoming packet order. This can be done in the software wrapping OS sockets. The third point is important if we are using different types of Inventors. It is not much usual, so it is currently nothing done for this. The fourth step is very important for running engines and animations based on global Inventor's time, because we want to they work same on the all computers. The last step should get an idea about the network connection performance, because this information is important for Network Layer. After all this is passed, the connection is fully established and appended to the global connection list.

Currently, the connecting computers algorithms are in the 'before realization' state, because it is on the edge of this topic. Instead, we are using PVM toolkit [2] for handling all network communication.

Chapter 7

Network Layer

7.1 Overview

Network Layer is the core network implementation providing convenient API interface for the rest of the distributed scene implementation. It is composed of three parts:

- network message classes
- low-level networking
- network management with public API by SoDS global class

Note that all Network Layer API is private and used throughout the distributed scene extension. Only public API is accessible through SoDS global class (see the chapter 6).

Network Layer has several important functions:

- It provides the hi-level network API for other distributed scene modules.
- It contains the rich set of network messages and several abstract classes for deriving additional messages.
- It automatically handles messages sending and receiving.
- It provides the convenient mechanisms for keeping the scene consistency up.

7.2 Handling of Scene Modifications

In figure 7.1 we can see the modification process from the section 2.4 with special focus on Network Layer. In the figure we can see two scene graphs. The left is master, the right slave. The process of the modification of the field in the transform node follows:

1. An user modifies the field.
2. The heading group of the scene graph is notified about the scene graph modification.
3. The modified field receives the path starting at the heading group and terminating at the field's container. It also receives the list of all network-slave connections of the heading node.

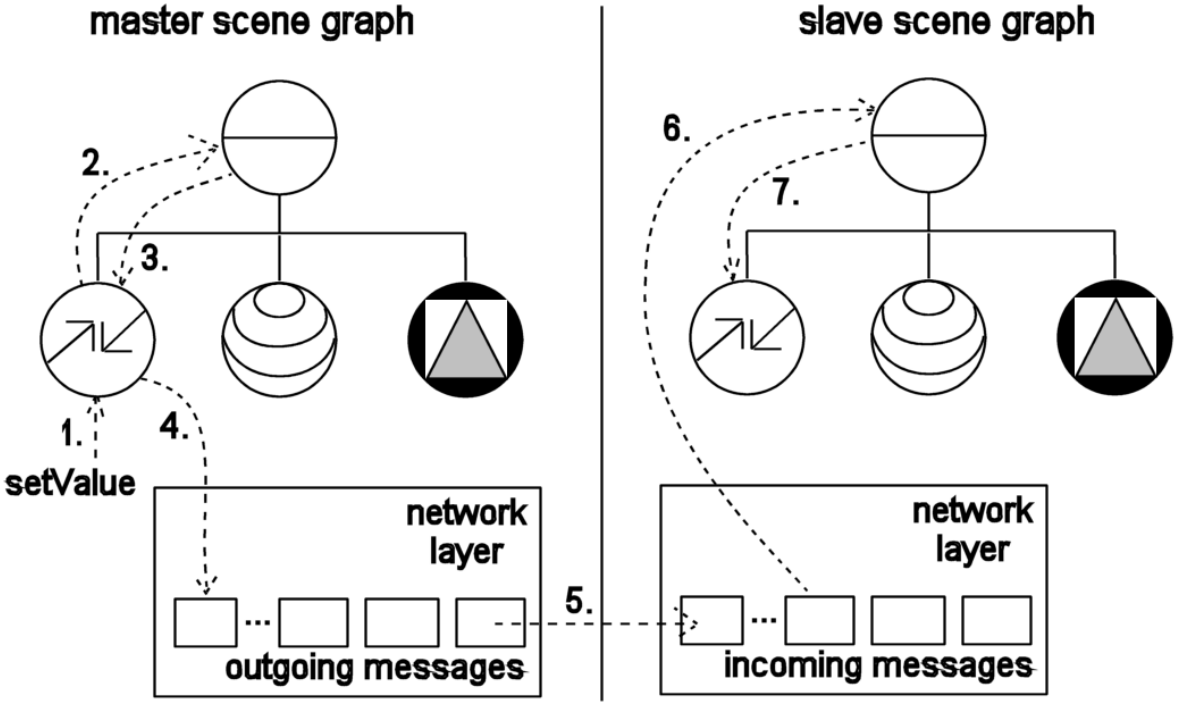

Figure 7.1: Handling of a scene modification

4. All appropriate data is passed from the field to Network Layer. The network message object is created and put to the queue.
5. All messages are sent to all their computer recipients.
6. After receiving, messages are sent to their destination objects in the scene graph.
7. The destination object applies message 'new value' message to the appropriate field.

The steps 1,2 and 3 has been well described in the section 2.4 about modifications handling. The fourth, fifth and sixth step will be described in this chapter that is focused on the basic distributed scene related problems. The seventh step again in the section 2.4.

7.3 General Design

Typical Inventor applications work in two stages (figure 7.2a):

1. Application processing and modifying of the scene
2. Rendering of the scene

Inventor applications using our network extension will work usually in four stages (figure 7.2b):

1. Application processing and modifying of the scene

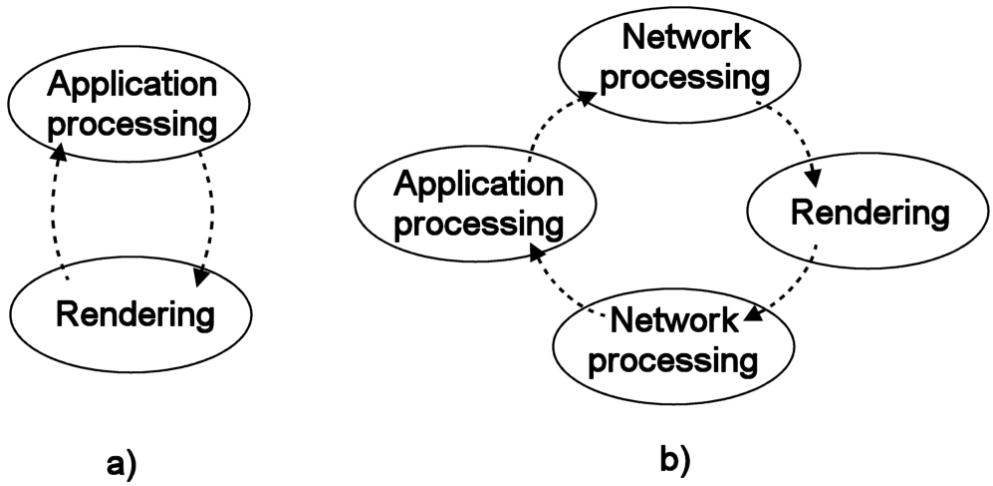

Figure 7.2: The application processing loop: a) without network b) with network

2. Network processing — sending and receiving the scene modifications
3. Rendering of the scene
4. Network processing — sending and receiving the scene modifications

We are updating the scene two times to keep the scene in the most possible actual state. This handles most of all cases. Anyway, if an user has some special needs, he can handle network processing by himself. There are three functions in global SoDS class for handling this: performSending, processReceivedMessages and processNetwork. For more information about the functions see the section 6.2.

The network is handled by main application thread, so we do not have problems with locking and interfering among threads. Anyway, multithreaded version should be not a great problem, because Network Layer strictly separates its data and the data of the scene graph.

7.4 Mechanism of Sending Messages

We do not pass messages directly to the network sockets, instead we are scheduling them in the queue and sending them together. There are two reasons for this:

- Sending large stream can be much more efficient then sending many small streams.
- We can easily detect multiple modifications of one field and send only the latest modification.

When a message is about to be scheduled in the outgoing network queue, the queue is first searched for messages that can be removed because they will lose their meaning by scheduling the message. For example, older modification of the same field may be in the queue. All this can be done only if we keep some consistency dependencies.

All messages are put in the queue until SoDS::performSending() is called. Then the sending starts. For each target computer we create list of messages from the queue. Then we store them to the streams and send the streams to their computers.

7.5 Handling the Scene Consistency

Some discussion about scene consistency has been given in the section 2.5. Now, because of the design of the message handling presented in the previous sections in this chapter, the handling of the consistency is quite easy. We will expect, the scene is consistent at the moment before and after scene is rendered. At these times the network is processed and all messages in the outgoing queue are sent. Because the scene is consistent at the time of processing, we can put 'consistency valid' message at the end of the queue. Each computer, when it receives this message, can be sure that applying all previous messages coming from the same computer do not break the scene consistency.

7.6 Limited-Bandwidth Networks

Sometimes there may be a problem with a network bandwidth. A programmer can make a very 'modification extensive' application that will need more network bandwidth than is available. The same problem can arise when too many computers are connected to one computer or only dial-up connection is available.

The most terrible scenario is that a server is making very many modifications and its sending queue is getting terribly long. Time from a modification on the server to updating all slaves takes many seconds. Finally, server's outgoing queue take all server's free memory and application crashes.

We should handle these problems. Some optimization work can be done on outgoing message queue. For example, we can remove 'consistency valid' messages of the long message queue to remove some consistency constraints and purge possibly many 'new value' messages. All this is for future research.

7.7 Process of Receiving Messages

Incoming messages stays in OS buffers until network is processed. It is done before and after rendering. All streams are read from the socket and they are processed. Messages are read from them and if 'consistency valid' message is found, all previous messages of the stream are applied to the scene graph.

7.8 Abstract Message Classes

I am using message classes derived from NSMessage (Network Sent Message) for sending and message classes derived from NRMessage (Network Received Message) for decoding and applying messages on the target computer.

```
class NSMessage
{
public:
  NSMessage* getNext() const;
  NSMessage* getPrev() const;
```

```
  virtual uint32_t getOneTargetIP() const;
  virtual bool isTargetingIP(uint32_t ip) const;
  virtual bool isEmpty() const;

  virtual void onSchedule();
  virtual void write(uint32_t ip, std::ostream &out);

  virtual ~NSMessage();

  void send();
};
```

`NSMessage* getNext() const;NSMessage* getPrev() const;`
The functions used for browsing in the outgoing message queue.

`virtual uint32_t getOneTargetIP() const = 0;`
The function returns one of the message target ip addresses.

`virtual bool isTargetingIP(uint32_t ip) const = 0;`
The function is testing whether ip is one of the target ip addresses of the message.

`virtual bool isEmpty() const = 0;`
The function returns false, if the message has been sent to all its target ip addresses.

`virtual void onSchedule();`
This virtual function does nothing by default, but it can be overridden by descendants to perform some actions when message is scheduled. It is called immediately before the message is put to the queue.

`virtual void write(uint32_t ip, std::ostream &out) = 0;`
The function writes the message to the stream specified by out parameter. When message is written, target ip address is removed from list of message targets. If no more messages rest, calling isEmpty() will return true.

`virtual ~NSMessage();`
Virtual destructor. It frees all allocated resources.

`void send();`
Schedules the message to be sent. It calls onSchedule() virtual function and then inserts message to the outgoing message queue.

```
class NRMessage
{
public:
  NRMessage* getNext() const;
  uint32_t getIP() const;
  void setIP(uint32_t ip);

  virtual void read(std::istream &in) = 0;
```

```
  virtual void apply() = 0;

  static void processMessages(std::istream &in, uint32_t ip);

  NRMessage(uint32_t ip);
  virtual ~NRMessage();
};
```

`NRMessage* getNext() const;`
The function returns next message in the incoming message queue.

`uint32_t getIP() const ;`
The function returns the ip address the message comes from.

`void setIP(uint32_t ip);`
The function for setting the message ip address.

`virtual void read(std::istream &in) = 0;`
The function reads the message from the stream.

`virtual void apply() = 0;`
The function 'executes' the message. Its content is applied to the scene graph. If there is a response, it will be passed to the outgoing message queue.

`static void processMessages(std::istream &in, uint32_t ip);`
The function creates appropriate message object, reads it from the stream and executes the message object.

`NRMessage(uint32_t ip);`
Constructor. It sets message ip address where the message comes from.

`virtual ~NRMessage();`
Virtual destructor. It frees all allocated resources.

7.9 Network Messages

- System messages:
  MSG_END_OF_STREAM
  MSG_PING
- Connecting computers:
  MSG_COMP_REG_REQ
  MSG_COMP_REG_ACK
  MSG_COMP_UNREG_REQ
  MSG_COMP_UNREG_ACK
- User messages:
  MSG_USER_MESSAGE

- Fields messages:
  MSG_FIELD_CONNECT_FROM_BY_PTR
  MSG_FIELD_CONNECT_FROM_BY_NAME
  MSG_FIELD_UPDATE_PTR
  MSG_FIELD_DISCONNECT_REQ
  MSG_FIELD_DISCONNECT_ACK
  MSG_FIELD_CANCEL_REQ
  MSG_FIELD_NEW_VALUE
- Containers messages:
  MSG_CONTAINER_CONNECT_FROM_BY_PTR
  MSG_CONTAINER_CONNECT_FROM_BY_NAME
  MSG_CONTAINER_UPDATE_PTR
  MSG_CONTAINER_DISCONNECT_REQ
  MSG_CONTAINER_DISCONNECT_ACK
  MSG_CONTAINER_CANCEL_REQ
  MSG_FIELD_CONNECT_FROM_BY_CONTAINER
  MSG_CONTAINER_NEW_VALUE

The message descriptions:

- System messages:

`MSG_END_OF_STREAM`
: The message is present on the each stream sent through the network. It serves also as 'consistency mark'.

  Message data:

  `< no data >`

`MSG_PING`
: The message is used for measuring ping times among computers. Not implemented yet.

  Message data:

  `time of sending`

- Connecting computers:

`MSG_COMP_REG_REQ`
: The message is sent to the another computer to ask it for establishing the connection. This is the only one message together with MSG_COMP_REG_ACK that can be sent without the non-established safe network connection.

  Message data:

  `char[] hostName`
  : network name of the requesting computer

  `void[] userData`
  : user data

`int userDataSize`
data size

`MSG_COMP_REG_ACK`
The message is the answer for receiving MSG_COMP_REG_REQ. If connection is accepted, accept is true and the safe network connection is established. Otherwise, accept is false and the connection is not established.

Message data:

`bool accept`
true, if the connection has been accepted

`int errorCode`
error code value; can be set independently whether the connection has been accepted or not

`void[] userData`
user data

`int userDataSize`
user data size

`MSG_COMP_UNREG_REQ`
The sending computer is asking the another computer for closing the safe network connection and to disconnect. There is not possible to refuse the message. This message must be the last sent message of the sending computer, because the connection is supposed to be closed by the sending computer.

Message data:

`< no data >`

`MSG_COMP_UNREG_ACK`
The message means the connection is closed and no more messages will arrive from the sending computer.

Message data:

`< no data >`

- User messages:

`MSG_USER_MESSAGE`
This is user defined message. A user can send his own messages. The message is marked with id identifier for user convenience. Data are specified as an array of chars of specified size.

Message data:

`int id`
message id

`char[] data`
data buffer

`int dataSize`
data size

- Fields messages:

  `MSG_FIELD_CONNECT_FROM_BY_PTR`
  : This is the connection request based on the destination field address.

    Message data:

    `SoField *dest`
    : destination field pointer

    `SoField *src`
    : source field pointer

  `MSG_FIELD_CONNECT_FROM_BY_NAME`
  : This is the connection request based on the field's name and name of its container.

    Message data:

    `char[] containerName`
    : destination field's container name

    `char[] fieldName`
    : destination field name

    `SoField *src`
    : source field pointer

  `MSG_FIELD_UPDATE_PTR`
  : The message is response to the MSG_FIELD_CONNECT_FROM_BY_NAME message, because for the full established connection we need to know the pointer of the target object.

    Message data:

    `SoField *dest`
    : destination field pointer

    `SoField *src`
    : source field pointer

  `MSG_FIELD_DISCONNECT_REQ`
  : The slave field is asking master for disconnecting.

    Message data:

    `SoField *dest`
    : destination field pointer

    `SoField *src`
    : source field pointer

  `MSG_FIELD_DISCONNECT_ACK`
  : This is a common answer for MSG_FIELD_DISCONNECT_REQ and MSG_FIELD-_CANCEL_REQ messages.

    Message data:

    `SoField *dest`
    : destination field pointer

`MSG_FIELD_CANCEL_REQ`
The master field is asking the slave field for disconnecting.

Message data:

`SoField *dest`
destination field pointer

`SoField *src`
source field pointer

`MSG_FIELD_NEW_VALUE`
The new value message sent by master field to the slave field.

Message data:

`SoField *dest`
destination field pointer

`char[] data`
new value's data

`int dataSize`
data size

- Containers messages:

`MSG_CONTAINER_CONNECT_FROM_BY_PTR`
The message is connection request based on pointer to destination object. The connecting objects can be SoFieldContainer's descendants. This includes SoGroup, so this message is also used to connect graphs.

Message data:

`SoFieldContainer *dest`
destination container pointer

`SoFieldContainer *src`
source container pointer

`MSG_CONTAINER_CONNECT_FROM_BY_NAME`
The message is connection request based on name of destination object. The connecting objects can be SoFieldContainer's descendants. This includes SoGroup, so this message is also used to connect graphs.

Message data:

`char[] containerName`
name of the destination container

`SoFieldContainer *src`
source container pointer

`MSG_CONTAINER_UPDATE_PTR`
The message is response to the MSG_CONTAINER_CONNECT_FROM_BY_NAME message, because for the full established connection we need to know the pointer of the target object.

Message data:

`SoField *dest`
destination container pointer

`SoField *src`
source container pointer

`MSG_CONTAINER_DISCONNECT_REQ`
The slave is asking master for disconnecting.

Message data:

`SoField *dest`
destination container pointer

`SoField *src`
source container pointer

`MSG_CONTAINER_DISCONNECT_ACK`
This is a common answer for MSG_CONTAINER_DISCONNECT_REQ and MSG-_CONTAINER_CANCEL_REQ messages.

Message data:

`SoField *dest`
destination container pointer

`MSG_CONTAINER_CANCEL_REQ`
The master is asking the slave for disconnecting.

Message data:

`SoField *dest`
destination container pointer

`SoField *src`
source container pointer

`MSG_FIELD_CONNECT_FROM_BY_CONTAINER`
This is the connection request to connect one field to the another based on the pointer to container and the destination field name.

Message data:

`SoFieldContainer *container`
pointer to the container

`char[] fieldName`
name of the target field

`SoField *src`
source container pointer

`MSG_CONTAINER_NEW_VALUE`
The new value message sent by master to the slave. The target field is found by path and field name.

Message data:

`SoFieldContainer *dest`
: destination container/heading node pointer

`path`
: path from the heading node to the field's container

`fieldName`
: name of the target field

`char[] data`
: 'new value' data

`int dataSize`
: data size

Chapter 8

Results

8.1 Overview

The purpose of the distributed scene toolkit is to give a programmers convenient API for development of shared distributed scenes. We have chosen to implement our distributed scene extension into Coin, a library that fully implements Open Inventor 2.1 API.

We have designed an easy-to-use API extension that is following the clear Open Inventor API design. We have choose to use master-slave architecture for synchronization of scene graphs, nodes and fields. Some investigation has been done to maintain consistency of distributed scenes.

We have verified that the robustness of Open Inventor can handle all demands for realization of the distributed scene. Moreover, there has been no need to use any hacks into Inventor's design.

8.2 Testing

I have developed an application for testing and demonstration purposes of my distributed scene extension. It is simple space scene viewer with implemented collision detection and one additional computer-driven ship.

For a testing purposes I have run the application on two computers and made connection among the scene graphs of the computer-driven ships, so the slave ship has been synchronized as it received the update messages from the master ship on the other computer. A screenshot is shown in figure 8.1.

There is another test shown in figure 8.2. I have established the connection between the root-nodes of the scene graphs of the two computers. So, the whole scene graphs are kept synchronized. We can see the same rotation of the planets and the same position and direction of the viewer. The slightly strange differencies between the positions of the computer-driven ships is caused by the 'latency' among the applications (non-accelerated OpenGL and very low Frames Per Second — FPS).

Finally some additional testing has been made to some special cases, like deletion of one of the computer-driven ships, to prove there is not a problem with Inventor design and the connection can be safely closed. We have not discovered any such problems and all special cases are handled in regular ways without using special hacks into Inventor implementation.

Figure 8.1: The testing and demonstration application: synchronizing positions of the two space ships between the two applications. Because of the screenshot the applications has been run on the same computer.

Figure 8.2: The testing and demonstration application: synchronization of the two scenes. We can see the same rotation of the planets, the same position and direction of the viewer and 'latency' caused differencies among the computer-driven ships (because about 1 FPS).

8.3 Conclusion

In the past time a programmer of distributed 3D graphics applications must hardly implement the difficult network functionality and then fight with objects synchronization, distributed objects interaction and the scene consistency.

Now, a programmer is saved from all network troubles, because all network functionality is implemented for him. The objects synchronization is handled in elegant ways and there are introduced some mechanisms for a programmer that can help him much to face the distributed scene pecularities.

The easiness of development Inventor local 3D applications is now extended by distributed abilities, so development of distributed 3D applications should be no longer hard work. I hope, this extension move the people's horizon of possibilities yet a little bit further.

8.4 Future Work

The aim of this work, to design a toolkit for distributed 3D graphics, has been accomplished. It means to touch many internal Inventor implementation principles to see posibilities of the design of the toolkit. Finally, our 'research' implementation proved that the ways of the design are right.

The first steps in the future research will probably be to turn our 'research' implementation to the 'public' one. It means especially to implement some convenient functionality that has not been in the center of the scope of this research work.

We expect additional reseach at the area of consistency models for the distributed virtual scenes. This, we hope, should minimize the amount of consistency problems a programmer must take care about.

Finally, some classes should be developed for making the development of the distributed scenes easier. There should be some motion related classes to make objects' motion smooth and to solve some collision detection related problems in distributed scenes.

Bibliography

[1] George Colouris, Jean Dollimore, Tim Kindberg, *Distributed Systems Concepts and Design,* Addison-Wesley, 1996

[2] PVM (Parallel Virtual Machine), http://www.csm.ornl.gov/pvm/pvm_home.html

[3] Gerd Hesina, Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics, 1999, http://www.cg.tuwien.ac.at/research/vr/div/

[4] V. Krishnaswamy, M. Ahamad, M. Raynal, D. Bakken, Shared State Consistency for Time-Sensitive Distributed Applications, in Newsletter of the Technical Commitee on Distributed Processing Fall 2001

[5] Systems in Motion, Coin3D, http://www.coin3d.org/

[6] SGI, Open Inventor, http://www.sgi.com/software/inventor/

[7] Template Graphics Software, Open Inventor, http://www.tgs.com/

[8] Josie Wernecke, *The Inventor Mentor,* Addison-Wesley, 1994

[9] Josie Wernecke, *The Inventor Toolmaker,* Addison-Wesley, 1994

[10] SGI, *OpenGL Performer Getting Started Guide,* 2000, http://techpubs.sgi.com/library/manuals/3000/007-3560-002/pdf/007-3560-002.pdf

[11] SGI, *OpenGL Performer Programmer's Guide,* 2001, http://techpubs.sgi.com/library/manuals/1000/007-1680-070/pdf/007-1680-070.pdf

[12] Václav Dvořák, *Advanced Computer Architecture,* Vysoké učení technické v Brně, 1997

[13] SGI, *The OpenGL Graphics System: A Specification (Version 1.3)*, 2001, http://www.opengl.org/developers/documentation/version1_3/glspec13.pdf