Network Streaming of Dynamic 3D Content with On-line Compression of Frame Data

Giuseppe Marino Paolo Simone Gasparello Davide Vercelli Massimo Bergamasco Franco Tecchia
Scuola Superiore Sant’Anna PERCRO

ABSTRACT
Real-time 3D content distribution over a network requires facing several challenges, most notably the handling of the large amount of data usually associated with 3D meshes. The scope of the present paper falls within the well-established context of real-time capture and streaming of OpenGL command sequences, focusing in particular on data compression schemes. However, we advance beyond the state-of-the-art improving over previous attempts of “in-frame” geometric compression on 3D structures inferred from generic OpenGL command sequences and adding “inter-frame” redundancy exploitation of the traffic generated by the typical architecture of interactive applications.

Index Terms: I.3.2 [Computing Methodologies]: Computer Graphics—Graphics Systems (Distributed/Network Graphics); H.4.m [Information Systems]: Information Systems Applications—Miscellaneous

1 INTRODUCTION
In this paper, we propose a method to allow efficient distribution of OpenGL streams over the Internet by means of various form of data compression. This work stems from our previous research in the context of cluster-based immersive visualisation; in this field, intercepting OpenGL or Direct3D calls by means of custom system drivers and distribute them over a network has proved to effectively decouple the application architecture from the underlying rendering system, allowing for a reconfigurable rendering nodes arrangement.

While previous work in this field [5][6][10] focused on multicasting command streams over Local Area Networks, our goal was to extend this kind of cluster-based rendering architectures to work over WAN networks too, as this would allow for a new generation of network streamed 3D applications. We propose here an approach that addresses the low-bitrate requirement and the real-time constraint challenges using a combination of inter-frame and intra-frame compression of OpenGL command streams as well as using light-weight compression/decompression schemes.

2 RELATED WORK
A number of approaches have been proposed to stream graphical commands over a network in a distributed rendering architecture. WireGL [5], is one of the most relevant examples. It is a software system developed to distribute the graphic load of an OpenGL application to a cluster of machines. A graphical command stream is generated by means of a custom device driver library capable of intercepting the OpenGL API calls invoked by a generic application.

The software project Chromium [6] is a further refinement of the WireGL concept. It inherits the interception mechanism of its predecessor, but system configuration become more flexible, allowing to build a graph that describes the complete network structure and can be used to configure each single node.

AnyGL [10], a large-scale hybrid distributed graphics system, deals with the problem of managing the large amount of data generated by a OpenGL streaming application, introducing the concept of data compression for the first time. In this case geometry data redundancies are reduced by using simple position predictors. Vertex normals [4] and colors are compressed in similar ways.

In our approach we propose the adoption of state-of-the-art geometry compression techniques [3]. In particular, we concentrate on the mesh connectivity compression. We adopt the valence driven approach, first introduced by Touma and Gotsman [9] and subsequently improved by Alliez and Desbrun [2]. In this technique the output symbol stream is the sequence of the vertex valence values, whose order is given by the conquest process. The mesh connectivity is compressed on average with 1.5 bits per vertex for regular meshes.

Differential compression is another approach to face the issue of data transmission. It consists in representing the differences between a source and a target files in a compact way. This technique is mostly based on the diff algorithm [7]. It produces, given a target and a source file, a patch containing a sequence of delete, change and append operations. The mathematical base of the diff algorithm is the solution of the longest common subsequence (LCS) problem. In our work we employ a Google code open source implementation of the algorithm [1].

3 CAPTURING OPENGL STREAMS
We organised our work around our own software system for the capture and distribution of OpenGL calls performed by a graphical application. Our system exploits a Chromium-like network scheme: there is a master node, where the graphical application is running, and a number of slave nodes, that receive the OpenGL stream generated by the master node and broadcasted into the network. The master node uses a custom device driver interposed between the application and the OpenGL system library, capable of intercepting any call performed. Every time a call is intercepted and before its execution, the custom driver creates a ghost command code to be streamed over the network. In this way each slave, once received...
the stream, can replicate all the OpenGL calls, reconstructing the master’s OpenGL state and graphical output.

4 THE OVERALL ARCHITECTURE
Once the OpenGL calls have been intercepted with the mechanism described in the previous section, they are passed to the packetizer module (Figure 2). This component is in charge to encode all the information about performed calls and then stores them into a command buffer. If the frame contains geometry description commands, a geometric compression module will handle them, and send the compressed output to the packetizer. Once this process has been completed, the buffer content is compared against the previous frame by the diff module, as described in section 6. The output of this operation is then compressed with a general purpose compressor (LZO or Zlib) and sent over the network. On the slave node the original data is reconstructed by decompression phases performed in reverse order with respect to the master side.

5 COMPRESSING FRAMES GEOMETRY
Transferring complete 3D model descriptions over the Internet may represent a serious problem. If models are large and/or the network link is not fast enough, slave node applications may stutter, being interrupted by long pauses spent for model synchronization.

We approach this issue by introducing a mesh encoder component, based on the combination of several general purpose and specific known compression techniques. Connectivity and geometry data is recognised by analyzing a sequence of OpenGL API calls, and then compressed with our scheme.

The valence driven approach [9][2] drives a mesh traversal using connectivity information, and outputs symbols whenever new topological elements are encountered. As soon as a vertex gets conquered, its attributes are quantized and predicted with the paralelagram rule [9], so that only differences are transmitted. A final compression stage provides entropy minimization by means of a fast adaptive context-based arithmetic coder [8].

6 EXPLOITING FRAME TO FRAME COHERENCE
The OpenGL frame structure of a typical 3D application begins with some frames dedicated to the description of the 3D objects composing the scene. A lot of data is generated at this point, and it can be compressed as described in the previous sections. After that, if the 3D model is no longer modified, the following frames often consist in the exploration of the model itself. Each frame is mainly a collection of drawing calls and calls to change the camera placement. It is clearly visible how the “exploration frames” are similar: they only differ by the position of the camera.

To exploit frame-to-frame coherence we use a diff algorithm, as described in Section 2. With this technique it is possible to avoid sending a large percentage of each frame data (all the geometry information). In many cases frames which size is of the order of several kilobytes can be represented with just a few dozens of bytes.

7 MEASUREMENTS
We have conducted formal testing of the compression methods: measurements were taken using as a test-bed a chromium-like cluster rendering system that we have developed in house over the years. We selected three test applications to be rendered on our framework, selected for their visual complexity, in order to cover a wide range of real-life scenarios: a simple scene with just a small number of moving objects, a complex CAD model being manipulated in real-time to a complex scenario traversed by the camera with a first-person perspective. We connected two computers over a network (Figure 1): at one side of the connection (A) runs the original application, whose OpenGL command stream is intercepted and sent to the slave node (B) for being remotely executed. The goal of our test is to evaluate if the generated traffic is compatible with typical Internet bandwidth, therefore we measured the size of the data chunks in various conditions. Preliminary results show that geometric compression reduces the traffic load at application bootstrap (and every time some new geometry is kicked-in), while frame-to-frame compression minimizes the amount of incremental data that needs to be exchanged as frames advance, exploiting the high redundancy in data flow for interactive applications. Numbers obtained suggest an average reduction for the first frame to 1/3 of the original data size and for subsequent frames to 1/8.

8 CONCLUSIONS
We have presented a method to allow efficient distribution of real-time generated content using on the fly compression and decompression of OpenGL command streams. We advance beyond the state-of-the-art improving over previous techniques of in-frame geometric compression on 3D structures inferred from generic OpenGL command sequences and adding inter-frame redundancy exploitation of the traffic generated by the typical architecture of interactive applications. Measurements reveal for this combination of techniques a very effective reduction of network traffic obtained with a modest CPU overhead.

REFERENCES