Real-Time Network Streaming of Dynamic 3D Content with In-Frame and Inter-Frame Compression

Paolo Simone Gasparello, Giuseppe Marino, Filippo Bannò, Franco Tecchia, Massimo Bergamasco

PERCRO, Scuola Superiore Sant'Anna, Pisa, Italy

ABSTRACT
Real-time 3D content distribution over a network (either LAN or WAN) has many possible applications, but 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. Measurements reveal for this combination of techniques a very effective reduction of network traffic and a CPU overhead compatible with the requirements of interactive applications, suggesting a significant application potential for Internet-based 3D content streaming.

KEYWORDS: Distributed Rendering, Distributed Applications, Remote Graphics, Delta Compression, Geometric Compression

1 INTRODUCTION
In this paper, we propose a method to allow efficient distribution of OpenGL streams over the Internet by means of various forms 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 distributing them over a network has proved to effectively decouple the application architecture from the underlying rendering system, allowing for an arrangement of reconfigurable rendering nodes, usually with no requirement to change the intercepted application.

While previous work in this field [1][2][3] focused on multicasting command streams over Local Area Networks, our goal was to extend this kind of cluster-based rendering architecture to work over WAN networks too, as this would allow for a new generation of network streamed 3D applications. The two main challenges we faced were: achieving a sufficient data compression ratio to allow low-bitrate network streaming, and ensuring that the process of data compression and decompression can happen in real-time and is compatible with the strict time constraints usually imposed by real-time generated content. We propose here an approach that addresses these challenges using a combination of inter-frame and intra-frame compression of OpenGL command streams as well as using light-weight compression/decompression schemes.

The paper is organised as follows. After a survey about the related works on the topic in Section 2, Section 3 briefly details the general technique of OpenGL command capturing and streaming and Section 4 illustrates the overall architecture of our system. Section 5 describes how in-frame geometric compression is achieved, while Section 6 is devoted to frame-to-frame compression. Finally, in Section 7 some significant numerical measures are reported. They are taken using a test-bed based on a Chromium-like cluster rendering system that we’ve developed in house over the years.

2 RELATED WORK
A number of approaches have been proposed to stream graphical commands over a network in a distributed rendering architecture. WireGL [1], 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. Collected data is then sent over a LAN network to one or more servers, each of them in charge of managing a different view of the scene or a different portion of the target framebuffer.

The software project Chromium [2] is a further refinement of the WireGL concept. It inherits the interception mechanism of its predecessor, but system configuration becomes more flexible, allowing the building of a graph that describes the complete network structure and can be used to configure each single node. Three kinds of task can be assigned to each machine of the network: they can be required to manipulate the incoming instruction stream in a custom way, or to split the stream into several sequences (Transformation), or to merge more sequences into a single one (Serialization). A special node, called the Mothership, deals with the initialization of the whole system, and it is capable of dynamically reconfiguring the system components. To achieve the maximum flexibility, instead of being constrained by standard static configuration files, the Mothership supports dynamic reconfiguration by means of special scripts written in Python.

AnyGL [3], a large-scale hybrid distributed graphics system, deals with the problem of managing the large amount of data generated by an 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 colours are compressed in similar ways. A command-code compression scheme is also implemented.

In our approach we propose the adoption of state-of-the-art geometry compression techniques. In particular, we concentrate on the mesh connectivity compression. A survey of the recent advances in compression of 3D meshes can be seen in [5] and [6]. All those techniques are generally grouped into two major categories: single-rate and progressive algorithms. Single-rate techniques compress a 3D mesh by visiting its surface in a
deterministic way, and “conquering” all the geometric elements as they are encountered. The stream of symbols emitted can be driven by triangle [7][8], edge [9] or vertex [10] properties.

The most effective known single-rate algorithm is the valence driven approach, first introduced by Touma and Gotsman [10] and subsequently improved by Alliez and Desbrun [11]. In this technique the output symbol stream is the sequence of the valence valence values, whose order is given by the conquest process. The mesh connectivity is compressed on average with 1.5 bits per vertex (bpv) for regular meshes, while the theoretical upper bound obtained by the worst-case analysis is 3.24 bpv, proving the “near-optimality” of this approach [12][11][13].

Progressive compression, on the other hand, allows for a mesh to be transmitted first as a coarse, low-detailed version. As more data reaches the receiver, it performs a mesh reconstruction by incrementing the level-of-detail until reaching the required quality. Many progressive compression approaches are based on the Hoppe progressive meshes [14][15].

In his work, he proposes to generate a sequence of levels-of-detail by iterating a simplification algorithm (as an example see [16]). Progressive encoders compress the coarse base mesh and the series of reversed simplification operations. For this purpose several approaches have been proposed to encode efficiently the vertex split sequence. Two of them are the PFS [17] and the CPM [18][19]. By the way, the effectiveness of single-rate algorithms is moderately better.

Spectral compression [20][21] is a different kind of progressive approach. It describes a notably effective compression method to encode the set of vertex attributes, whilst relying on one of the single-rate techniques seen above to encode the connectivity data. It is well known [22] that this method is considerably more effective when compared with Touma and Gotsman’s algorithm. But the high complexity required to compute the Laplacian eigenvectors on both encoder and decoder side makes this approach infeasible for an implementation in real-time applications.

In literature, the issue of efficient data transmission is usually faced in two different ways. The data differencing approach consists of representing the differences between source and target files in a compact way. On the other hand, data compression compresses all the data in a single file. Data differencing techniques are mostly based on the diff algorithm [23]. It produces, from a given “target file” and a “source file”, a so called patch (or “delta file”), containing a sequence of delete, change and append operations. Replying the same operations on the source file content it is possible to reconstruct the target file.

The mathematical base of the diff algorithm is the solution of the longest common subsequence (LCS) problem. Considering two sequences of items, it consists in finding the longest set of items that appears in both the original sequences and in the same order.

The algorithm is commonly employed for showing the differences between two different versions of the same ASCII file. Most of the source code management environments are based on it, in order to save disk usage. In fact, since subsequent versions of source code usually differ only by a small number of lines, it is more convenient to store just the differences rather than the full versions.

In [24], Korn and Vo showed that the concept of data compression could be thought as a special differencing operation in which the source data is empty. They proposed Vdelta, a delta compression algorithm based on the Lempel-Ziv77 [25] approach. Afterwards, they proposed a newer version of the algorithm, called vcdiff [26].

In our work we employed open-vcdiff [27], a Google Code open source implementation of the vcdiff algorithm.

3 Capturing OpenGL Streams

We organised our work around our own software system for the capture and the 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 runs, and a number of slave nodes, that receive the OpenGL stream generated by the master node, broadcast over the network. The master node uses a custom device driver interposed between the application and the OpenGL system library, capable of interpreting 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 it has received the stream, can replicate all the OpenGL calls, reconstructing the master’s OpenGL state and graphical output.

The amount of network traffic generated per second in this way could be large, especially for complex 3D applications. Our aim is to distribute 3D content over the Internet in a way similar to video streaming. We look for a real-time transmission, even if real-time synchronisation between sender and receiver is not required. So, in order to preserve video smoothness, several frames are buffered before visualising the first one. Nonetheless the network load is still a factor to take into account. A large amount of OpenGL traffic per second results in buffer underflow, causing frame freezes that can last for a long time. So, in order to send an OpenGL stream over the Internet, it is very important to significantly reduce the size of every frame.

Prior to the application of data compression, the structure of the stream as naively generated by the master custom driver is rather redundant. In fact, in any interactive application there are a number of calls performed several times using the same parameter values. In addition, a 16 bit number is associated with each of the functions, regardless of the frequency they are called with. To eliminate this kind of redundancy, each frame can be compressed using a general purpose algorithm (such as zlib and LZO). Using this solution, each frame’s size can be significantly reduced, but for internet streaming applications this is still not sufficient. In our approach we propose to apply incremental encoding to OpenGL streams: the similarities between subsequent frames can in this way be exploited as most of the standard video compressors currently do.
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 on model synchronization. In practice, transferring a complete representation of a complex 3D environment made up of thousands of polygons, may take several minutes on a slow network connection.

To have an idea of the traffic generated by the description of a 3D model consider for example an application using the OpenGL indexed elements drawing mode with GL_TRIANGLES primitives, conceptually equivalent to a face-vertex incident table. Assume for the moment that each vertex is only described by its 3D-world coordinates \( x, y, z \). In this kind of representation a triangle corresponds to a set of three vertex indices. Suppose one uses the type GLfloat to represent vertex coordinates and type GLint for the indices. A mesh with \( v \) vertices and \( t \) triangles would require a traffic of at least (byte):

\[
M = t \cdot 3 \cdot \text{sizeof}(\text{GLfloat}) + v \cdot 3 \cdot \text{sizeof}(\text{GLint})
\]

\[
M = 12 \cdot t + 12 \cdot v = 12 \cdot (t + v)
\]

As a direct consequence of the Euler characteristic of a mesh [28] it can be stated that:

\[
t \approx 2v \Rightarrow M \approx 12 \cdot (2v + v) \approx 36v
\]

The above expression shows that a simple mesh needs approximately 36 bytes per vertex, or, technically, 288 bits per vertex (bpv). However, 3D model descriptions often include OpenGL commands associated to others per vertex attributes: normals, colours, texture coordinates, or further generic attributes.

OpenGL allows for a slightly more efficient mesh description by using GL_TRIANGLES_STRIP primitives, but redundancy is still an issue. In our work, an “in-frame” geometric compressor has been introduced. It can turn the previously mentioned 288 bpv representation into a form with only 30 bpv on average. By means of this specific compression technique, the traffic generated is drastically reduced, allowing a fast 3D model distribution even over a poor performance network like the Internet.

The approach used in the design of our mesh encoder is based on a combination of a set of well known compression techniques, both general purpose and specific, each one giving a tangible contribution to decreasing the memory requirements for model storage. A 3D mesh can basically be considered as the conjunction of two types of data: vertex attributes and connectivity. Vertex attributes are represented by vertex coordinates, normals, colours, texture coordinates or by anything that can be specified with a glVertexPointer command. Connectivity is represented by the vertex indices list, that indicates how vertices are combined to form surfaces. Generally, both data flows are treated separately and follow different data processing paths. Nevertheless, their interaction is very important during certain steps, as explained below. At the end of the process the two flows join in a unique data stream to fill in a single memory buffer. Figure 4 illustrates the architecture and the functional scheme of the compressor.

In our implementation, the meshes of triangles are recognised by analysing a sequence of OpenGL commands. OpenGL allows explicit specification of the mesh connectivity when indexed element commands are used (glDrawElements or...
glArrayElement). By means of other drawing modes (immediate mode or with glDrawArrays for example) the mesh connectivity can be inferred through a process of vertex clustering: vertex indices are assigned in such a way as to repeat the same index for those vertices whose attributes have the same values.

Some functional block of the process requires the input model to be a manifold. Therefore, in the first stage, the mesh topology is checked and eventually adapted [29] to comply with the manifold constraint. In this phase, vertex attributes data and connectivity data are separated to feed the second stage of the process, where a compound of specific operations are performed. This results in a stream of output symbols which have a statistical distribution suitable for an entropy based encoding (the third stage).

The VBE block (Valence Based Encoding) [10][11] is the kernel of the second stage. Using connectivity information, this component performs a complete mesh traversal (or conquest) and outputs symbols whenever new topological elements are encountered. Geometry coordinates, colours, normals and texture coordinates are analysed according to the VBE conquest process. Whenever a vertex gets conquered, its attributes are first quantised and then compared with the result of a simple first order predictor, which is based on the parallelogram rule [30]. Only the differences are transmitted, so that the statistical distribution becomes a Gaussian-like curve centred around zero. Quantisation is the sole lossy compression component of the entire process and the quality of the result depends on the number of bits chosen for quantisation indices. The value of the quantisation factor can be configured as a fixed value, or can be heuristically selected for each set of primitives. It can be thought that quantisation may be avoided to improve quality, however this would be wrong, because quantisation is the key to have a zero cumulative error prediction. The third stage provides entropy minimisation using a fast adaptive context-based arithmetic coder [31]. In Figure 4 the different statistical contexts are represented by the SC blocks.

![Figure 4. The architectural scheme of the 3D mesh compressor](Image 101x518 to 249x730)

**5.1 Compressing Textures**

Textures in OpenGL are usually described as an uncompressed matrix of pixels. It is easy to figure out how fast the traffic load can grow if they are transmitted in the same raw format they are managed by OpenGL.

The topics about image compression are widely discussed in literature, and a vast amount of solutions have been proposed, both lossless and lossy. It is well known that images such as logos, composed by large contiguous areas having exactly the same colour, are more suitable to be compressed with lossless algorithms (i.e. PNG [32]). On the other hand, pictures taken with cameras are better compressed with lossy algorithms, for example JPEG [33] or the recent WebP [34].

Our system makes use of libraries capable of both lossless and lossy image compression algorithms, but the feature to selectively decide which one is better to apply for each single texture is currently missing. For this reason, from now on we suppose that all the textures are compressed with JPEG lossy algorithm, with a quality factor configurable by user. In our tests, this choice led to better results in terms of average effectiveness. This allows to significantly reduce the size of the textures with negligible losses to the image quality.

**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. During these frames, geometric data is usually stored inside display lists or vertex arrays. 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 not modified, the following frames often consist of only the exploration of the model itself. Each frame is mainly a collection of drawing calls and calls that change the camera placement. It is clearly visible how the “exploration frames” are similar: they only differ in the position of the camera. So, once the drawing calls are sent for the first time (inside the OpenGL stream associated to the first frame), the only information really needed to reconstruct the subsequent frames are the updated camera positions.

To exploit frame-to-frame coherence we use a diff algorithm, as described in Section 2. Figure 5 illustrates how the diff operation is performed by our system. The OpenGL stream associated to each “exploration” frame can be seen as the sum of its geometry

![Figure 5. Working scheme of the diff compression in theXVR Network Renderer. Dashed arrows represent packet sending over the network](Image 319x596 to 559x730)
first frame under the form of display lists, vertex arrays and texture objects. On the other hand, Frame 2 provides a good insight into the amount of data crossing the network after the creation of those entities. Finally, the differences in size between Frame 2 and Frame 3 suggest how much redundancy could be detected between two consecutive (and similar) frames.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Frame 0 Size</th>
<th>Frame 1 Size</th>
<th>Frame 2 Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>33,375</td>
<td>4,602</td>
<td>4,602</td>
</tr>
<tr>
<td>Motorbike</td>
<td>6,177,468</td>
<td>1,705</td>
<td>1,705</td>
</tr>
<tr>
<td>Roller Coaster</td>
<td>18,359,992</td>
<td>18,410</td>
<td>18,410</td>
</tr>
</tbody>
</table>

Table 1. Frame size [bytes] with only LZO compression

<table>
<thead>
<tr>
<th>Applications</th>
<th>Frame 0 Size</th>
<th>Frame 1 Size</th>
<th>Frame 2 Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>10,663</td>
<td>4,602</td>
<td>4,602</td>
</tr>
<tr>
<td>Motorbike</td>
<td>2,093,678</td>
<td>1,705</td>
<td>1,705</td>
</tr>
<tr>
<td>Roller Coaster</td>
<td>7,655,571</td>
<td>18,410</td>
<td>18,410</td>
</tr>
</tbody>
</table>

Table 2. Frame size [bytes] with LZO plus in-frame compression

<table>
<thead>
<tr>
<th>Applications</th>
<th>Frame 0 Size</th>
<th>Frame 1 Size</th>
<th>Frame 2 Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>33,375</td>
<td>3,931</td>
<td>3,931</td>
</tr>
<tr>
<td>Motorbike</td>
<td>6,177,468</td>
<td>208</td>
<td>208</td>
</tr>
<tr>
<td>Roller Coaster</td>
<td>18,359,992</td>
<td>92</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 3. Frame size [bytes] with LZO plus inter-frame compression

<table>
<thead>
<tr>
<th>Applications</th>
<th>Frame 0 Size</th>
<th>Frame 1 Size</th>
<th>Frame 2 Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>10,663</td>
<td>3,931</td>
<td>3,931</td>
</tr>
<tr>
<td>Motorbike</td>
<td>2,093,678</td>
<td>208</td>
<td>208</td>
</tr>
<tr>
<td>Roller Coaster</td>
<td>7,655,571</td>
<td>92</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 4. Frame size [bytes] with LZO plus in-frame and inter-frame compression

Looking at the results in the tables, we can easily figure out a few interesting facts: geometric compression allows a significant reduction of the data size only for the first frame of the sequence (data chunk size is in this case reduced to about 1/3 of the original size in all three scenarios), while it seems not to impact the subsequent frames. This is reasonable, as in our applications most of the geometric entities are assembled right at the beginning. On the other hand, frame-to-frame coherency detection using the diff algorithm drastically reduces the data traffic the two computers need to exchange in order to render the same frame, both for the Motorbike and for the Roller Coaster applications. This is justified by the fact that the differences between consecutive frames of these applications are minimal: for the Motorbike the two frames are largely identical, with the exception of a few roto-translations of some components of the displayed geometry, while in the Roller Coaster application the only dynamic part of the scene is the position of the camera. Only in the Spheres applications,

7 Measurements

We’ve conducted formal testing of the compression methods: measurements were taken using as a test-bed a Chromium-like cluster rendering system we’ve developed in house over the years [35]. We selected three significant test applications (see Figure 6) to be rendered on our framework. Applications were 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 (Spheres App), a complex CAD model being manipulated in real-time (Motorbike App) and a complex scenario traversed by the camera with a first-person perspective (Roller Coaster App).

We connected two computers over a network as in Figure 1: the original application runs on one side of the connection (Computer A). Its OpenGL command stream is intercepted by our special driver. Frame data is organised in chunks (one chunk for frame) and sent to the slave node (Computer B) in order to be remotely executed. The goal of our first test is to evaluate if the generated traffic is compatible with typical Internet bandwidth. Therefore, we measured the size in bytes of the data chunks in various conditions, as reported in the Tables 1, 2, 3 and 4.

Note that we reported in the tables data relative to the first three frames only: this is because, in many VR frameworks, most of the geometry and texture data crosses the network only during the first frame under the form of display lists, vertex arrays and texture objects. On the other hand, Frame 2 provides a good insight into the amount of data crossing the network after the creation of those entities. Finally, the differences in size between Frame 2 and Frame 3 suggest how much redundancy could be detected between two consecutive (and similar) frames.

Figure 6. Some screen-shot of the test applications

(G) and camera setting (C) sections. G sections are the same for all the frames, while the C sections are frame dependent. While the first frame is sent unmodified, for each of the subsequent frames an incremental packet is generated and sent over the network. With this technique it is possible to avoid the transmission of a large percentage of the frame data. In many cases, frames with sizes of the order of several kilobytes can be represented with just a few dozen bytes.

This technique is especially beneficial for static geometry, but can still be convenient in the case of a model whose geometry changes over time, that is the typical case of 3D modelling software. The diff algorithm is in fact capable of detecting modifications done at vertex or attribute level, and to put just these differences into the OpenGL stream.
where the dynamic objects are bouncing around, shadows are dynamic, and the camera is animated, the differences between consecutive frames can't be significantly reduced. The two compression schemes therefore appear to be truly complementary: geometric compression reduces the traffic load at the application bootstrap (and every time some new geometry kicks-in), while frame-to-frame compression minimises the amount of incremental data that needs to be exchanged by the nodes as frames advance, exploiting the high redundancy in data flow for interactive applications.

A second set of tests were conducted on the system to see how much the compressing and decompressing processes impacts the overall rendering performance. The fundamental question was: is compression/decompression compatible with real-time rendering? Tables 5 and 6 report our findings. We measured the “frame execution time” with compression/decompression turned ON and OFF, i.e. the time needed for the rendering to be fully executed both on the master node A and on the slave node B, as in our architecture the two nodes are synchronised around the swap-buffer operation. This allows us to find out how much the compression and decompression codecs are slowing down the overall rendering process. All test were performed on a Microsoft Windows Vista 32 Bit machine, Pentium Intel Core 2 Duo T9400 / 2.53 GHz (Dual-Core) with 4GB of RAM and an ATI Mobility Radeon HD 3650 - 512 MB video-card.

<table>
<thead>
<tr>
<th></th>
<th>Frame 0</th>
<th>Frame 1</th>
<th>Frame 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>12.6673</td>
<td>4.0159</td>
<td>3.95106</td>
</tr>
<tr>
<td>Motorbike</td>
<td>1157.56</td>
<td>27.4411</td>
<td>29.8014</td>
</tr>
<tr>
<td>Roller Coaster</td>
<td>20,523.3</td>
<td>41.9717</td>
<td>32.6774</td>
</tr>
</tbody>
</table>

Table 5. Frame rendering time [ms] with only LZO compression

<table>
<thead>
<tr>
<th></th>
<th>Frame 0</th>
<th>Frame 1</th>
<th>Frame 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>80.5976</td>
<td>7.16921</td>
<td>4.81178</td>
</tr>
<tr>
<td>Motorbike</td>
<td>9.440.13</td>
<td>17.3465</td>
<td>18.9491</td>
</tr>
<tr>
<td>Roller Coaster</td>
<td>32,382.5</td>
<td>54.3147</td>
<td>28.9925</td>
</tr>
</tbody>
</table>

Table 6. Frame rendering time [ms] with LZO plus in-frame and inter-frame compression

Again, results allow for simple interpretations: while it helps reducing network traffic, geometry compression results in increasing significantly the time to process geometry-intensive frames. On the other hand, we have to remember that, for Internet based applications, longer processing times are compensated by shorter network transfers due to reduced data size, and that, in any case, long geometry processing time is usually only a concern at application start-up. From the measurements it also emerges that exploiting frame-to-frame compression with the diff algorithm does not really have a huge impact on the overall performance. On the contrary, the benefits of having a reduced data size seem to improve the rendering time even over the LAN we used for taking our measurements.

8 CONCLUSIONS AND FUTURE WORK

In this paper we've 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 of 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. For this combination of techniques, measurements reveal a very effective reduction of network traffic obtained with a modest CPU overhead. This suggests a significant application potential of the technique whenever the amount of data bandwidth is limited, such as in the case of Internet-based 3D streaming.

We are at present expanding the current research by investigating techniques for progressive transmission of frame data [18][20] containing 3D model descriptions. The goal is to distribute the heavy bursts of such data on a set of subsequent frame chunks, still maintaining usability and interaction constraints during the transmission phase.

An enhanced version of the texture compression module is being planned where each texture is analysed in advance to dynamically determine which kind of image compression algorithm is more suitable to apply in order to improve the effectiveness of the process.

Another possible improvement could be the introduction of a state tracker able to make the system robust in case of data loss. Data packets containing OpenGL state modifications could be sent over TCP connections, while the other ones could be sent with faster UDP datagrams. This mechanism would be similar to the one used in most of the video compression algorithms.

ACKNOWLEDGEMENTS

This work was funded by the project BEAMING, acknowledging the financial support of the Seventh Framework Programme for Research of the European Commission DG-INFSO-D2, under grant number 248620.

REFERENCES