Optimized Visualization for Tiled Displays

Mario Lorenz and Guido Brunnett
Computer Graphics and Visualization, Chemnitz University of Technology, Germany
{ Mario.Lorenz | Guido.Brunnett }@informatik.tu-chemnitz.de

Abstract
In this paper we present new functionality we added to the Chromium framework. When driving tiled displays using a sort-first configuration based on the Tilesort stream procession unit (SPU) the performance bottlenecks are the high utilization of the client host caused by the expensive sorting and bucketing of geometry and the high bandwidth consumption caused by a significant amount of redundant unicast transmissions. We addressed these problems with an implementation of a true point-to-multipoint connection type using UDP multicast. Based on this functionality we developed the so called OPT-SPU. This SPU replaces the widely used Tilesort-SPU in typical Sort-First environments. Tile-sorting and state differencing is not necessary because Multicasting allows us to send the geometry to all server nodes at once. Instead of tile-sorting a conventional frustum culling method is used to avoid needless server utilization caused by rendering of geometry outside their viewports. This approach leads to significant lower processor and memory load on the client and a very effective utilization of available network bandwidth. To avoid redundant transmissions of identical command sequences that are generated by the application several times we put a transparent stream cache into the multicast communication channel. In addition, frustum and hardware accelerated occlusion culling methods may be used to eliminate unnecessary transfer of invisible geometry. Finally, a software based method for synchronization of buffer swap operations at all servers was implemented. In a nutshell, for the first time an appropriate combination of our optimizations makes it possible to render large scenes synchronously on an arbitary number of tiles at nearly constant performance.

Categories and Subject Descriptors (ACM CCS): I.3.2 [Computer Graphics]: Distributed/network graphics

1. Introduction

The development of so called *tiled displays* has been initiated by users wishes for very large screens with very high resolutions. In most cases each tile is projected by one beamer driven by one graphics card integrated in one node of a clustered rendering system. To present stereo images a second projector may be needed per tile. Building up a tiled display using mainstream projection hardware is a difficult task because one has to deal with mechanical and optical problems to guarantee the tiles will form a continous frame. To provide seamless overlaps between adjacent tiles mostly *edge blending* is done. This is realized by usage of expensive precision mechanics in conjunction with signal processors. In addition to the hardware an application or toolkit is needed to render the parts of the frame synchronized and parallel on the cluster. Using scene distribution, parallel rendering processes running on different nodes take the advantage of the cluster best, but the design of parallel rendering applications is often difficult and expensive.

As an alternative to redesign existing applications a special parallel rendering interface can be implemented that allows to run native applications on tiled displays [ISH98]. In this case a library has to manage the parallel rendering process on the cluster. In other words, a parallel OpenGL library has to simulate a virtual GL-Pipeline with a frame buffer resolution identical to the resolution of the entire tiled display. The main problem is the internal handling of very large data streams that carry simple graphics commands delivered by the applications to the virtual pipeline. For instance, the GL stream of a single colored, lighted and textured sphere represented by the glutSolidSphere(1,10,10) command contains two triangle fans and eight quad strips with a total data size of about 7100 kBytes taking five UDP datagrams. It is clear that common network hardware is by far not fast enough to allow continuous rendering of complex scenes.

In this paper we present optimizations of the *Chromium* framework. We introduce fast, non redundant network communication of most stream parts as well as situation dependent stream modifications, e.g. caching and culling. In summary, the optimization methods we present in this paper enhance the usability of Chromium to run unmodified OpenGL applications on clusters dedicated to drive very large tiled displays. The over-all performance is much better than with the widely used Tilesort based configurations.

2. Related Work

Stanford's WireGL [HEB*01] was one of the first research projects were a system architecture for high performance remote rendering [HBEH00] was developed. Based on WireGL the widely known *Chromium* framework was designed [HHN*02]. It is a software to build a parallel rendering system. Using the so called *Tilesort* stream processing unit (SPU), a Sort-First architecture [MUE95] can be configured to allow native OpenGL programs to render to a tiled display. But, the internal data handling of Chromium has some serious drawbacks causing slow frame rates and poor synchronization of tile servers.

In Chromium a client/server architecture based on the concatenation of so called software nodes is realized. All nodes may be placed on different cluster hosts. The framework contains a network library allowing point-to-point (unicast) connections between nodes only. The application sends graphics commands to the client calling functions of the *libcrfaker*. This replacement of the GL library is the entry point to the network of Chromium stream transportation and manipulation units. Every node contains a chain of stream processing units (SPU) that implement functions to handle GL commands. Within a SPU-chain the command stream is handled with subsequent function calls. To encode stream commands in network packets for transportation between nodes the framework contains the packer library. It also includes the functions needed to decode received buffers on the server side. The unpacker converts received commands to sequent calls of functions implemented in the SPU chain. The last SPU of a server's chain may either render to a graphics pipeline or send packed buffers to subsequent servers. However, when a tiled display is driven each server node runs a render-SPU at the end of his chain.

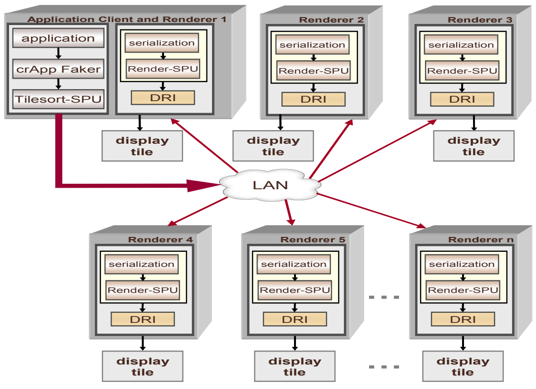


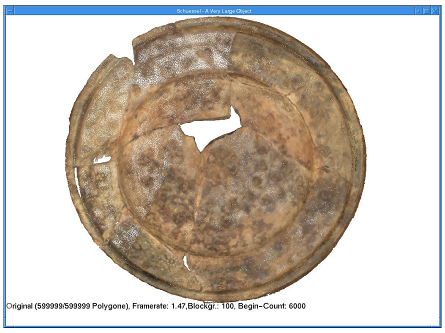
Figure 1: *Sort-First cluster based on Tilesort-SPU*

Figure 1 shows a typical configuration of a graphics cluster running Chromium to drive a tiled display. In this case each tile is driven by one server node running on a dedicated PC. Additionally, one PC also hosts the client node executing the application and the faker library as well as the Tilesort-SPU at the end of his chain. The Tilesort-SPU implements a set of functions for splitting the incoming command stream into several streams. In the trivial case this is done by encoding the commands into network buffers using the packer and sending filled buffers to all servers. Each server decodes the received command buffers and maps the contained commands to functions of the SPU-chain. Finally the render-SPU at the end of this chain maps the functions to the GL hardware driver. A detailed description of the Chromium Framework can be found in [HHN*02].

3. Drawbacks of typical Sort-First Architectures

If a parallel rendering system is dedicated to drive a tiled display, two important demands must be satisfied. First, it should render the incoming command stream fast and correct. Second, to provide a coherent impression of images all tiles of each frame have to be displayed synchronously. In addition, if shutter hardware is used for the presentation of stereoscopic image pairs, the timing of the video signals has to be synchronized.

As described in the previous section the Chromium framework can be configured to realize a parallel rendering system that drives a tiled display. However, the design of tile-sort based architectures leads to important drawbacks that will be described in the next paragraphs. To benchmark Chromium we rendered typical scenes with different characteristics on clusters consisting of 1 to 12 nodes. Figure 3 shows the frame rate when a rotating plate (Figure 2) of about 600k colored triangles organized in 6000 glBegin() - glEnd() blocks is rendered.



Original (599999/599999 Polygone), Framerate: 1.47,Blockgr.: 100, Begin-Count: 6000

Figure 2: *A plate composed of 600k triangles*

This is a typical application that requires a large, high resolution display because single lines cannot be seen if the object is displayed in wireframe mode at 1280x1024. If the same object is viewed at 3840x2560 it is possible. The diagram shows the loss of performance if more tiles were added to the cluster. In general the system becomes slower with an increasing number of tiles. Tile-sorting is mostly faster than the so called Broadcast mode if the distribution of objects in scene fits well to the tiling. For instance, using a 3x1 tiling, the plate may only be visible at the center tile. Then the plate can be rendered nearly as fast as drawing one tile only because most parts of the stream have to be sent to one server only.

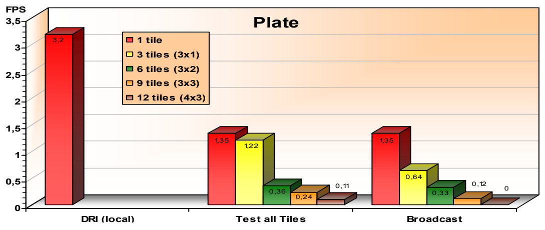


Figure 3: *Tilesort results rendering the plate*

A walk-trough style application (Figure 4) sequentially rendering images of a mid-complex urban scene runs in the Test-All-Tiles mode about half as fast as local DRI based rendering (Figure 5). The client ran out of memory in Broadcast mode with more than 5 servers.

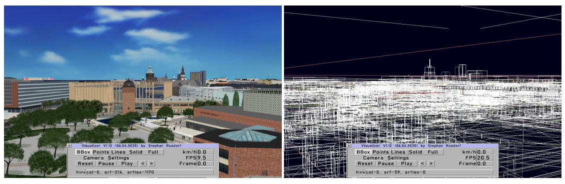


Figure 4: *The urban scene*

Unfortunately, the poor syncronization of tiles especially at low frame rates makes the navigation difficult or completely impossible. Surprisingly we found out that the Test-All-Tiles mode creates artefacts (wrong colored objects) and Broadcast does not.

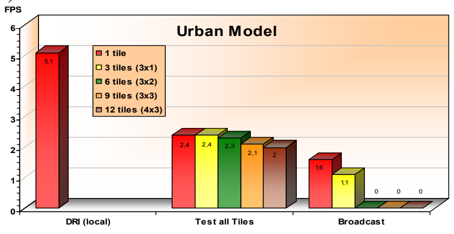


Figure 5: *Tilesort results rendering the urban scene*

As expected, the rendering performance decreases with every new tile added to the cluster. When DMX is used to draw to the application window, the frame rate decreases with the number of tiles including parts of the window. If a lot of geometry is drawn only on few tiles, an optimized bucketing mode brings slightly better results than Test-All-Tiles. Unfortunately the faster bucketing modes, e.g. Uniform Grid, can only be used with special setups and mostly not in combination with Distributed Multiheaded X (DMX).

Rendering our test scenes at adequate frame rates using a Tilesort based architecture is only possible if the display consists of up to 6 tiles. For a larger number of tiles this is only possible if the application makes extensive use of GL display lists. In general, low frame rates will be tolerated by users if the image tiles are displayed synchronously. Because we have no expensive hardware that supports framelocking, the tiles were updated "round robin" or random instead of synchronous. This leads to a complete loss of visual impression. Hence, a closer look to the functionality of Chromium is necessary.

A so called bucketing mode is implemented in the Tilesort-SPU. Bucketing uses the projected 2D bounding rectangle of the figure the commands would draw to decide on which server node a incoming GL command sequence must be processed. In most cases, in particular if scenes are drawn to the application window using DMX, the projected bounding rectangle is tested against all tiles (Test-All-Tiles). If the bucketing detects that a command sequence has to be sent to several servers this is done sequentially.

Because it is useless to update the GL state of servers that do not receive geometry, so called state differencing is done. The state tracker implements the logic of the GL system state handling. Two states S_i and S_j can be tested to detect changes. If S_i and S_j are different, a minimal set of GL commands is generated to update S_i to S_j. The Tilesort-SPU uses the state tracker to optimize the streams sent to servers. By this way only the minimal command set will be transmitted to keep the server state up to date. Additionally the Tilesort-SPU can answer state queries without asking the servers.

Two drawbacks of Chromium can be identified. First, if geometry has to be drawn on several tiles it will be sent sequentially to each server using unicast connections. For complex scenes rendered on many tiles this causes a significant amount of redundant communication and, of course, an inefficient utilization of available network bandwidth. In the worst case each geometry has to be sent to each server. Second, the processor utilization of the system that hosts the client node is always very high. This is mainly caused by the demands of the application, the cost-expensive bucketing algorithm, the packaging and state tracking functions. In summary the main bottlenecks of Chromium based systems are the poor utilization of available network bandwidth and the high load of the client host.

4. Optimizations

We analyzed various approaches to eliminate the drawbacks described in the previous section. It is our opinion that only a combination of different methods is able to increase the overall performance of the Chromium framework significantly. Obviously the most important task is the effective elimination of redundant transmissions. This should cause a speedup if geometry encoded in packed buffers is visible on more than one tile, is broadcasted or is drawn at least two times by the application, e.g. rendering stereo pairs. The elimination of redundancy reduces the consumed bandwidth as well as the clients processor load because less buffers need to be packed and sent. The free client ressources can be used for additional optimizations e.g. occlusion culling on the client side.

4.1. Multicast transmissions

Multicasting can be used to send network packets to a group of recipients at once. To integrate Multicasting into Chromium a new connection type must be implemented that allows to send incoming GL commands to all render servers. However, this is quite difficult because a lot of commands need to be treated specially, e.g. frustum and viewport related instructions. These functions are called serializing commands because they force the *parallel* transmission to stop and initiate a *serial* communication. Therefore the data handling of sending and receiving components has to be modified to benefit from this new communication channel. Also, state tracking is useful to avoid network communication if the application queries state entries.

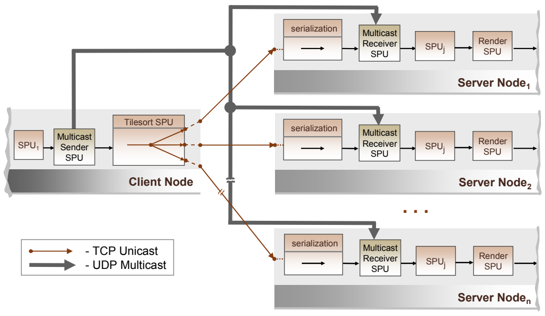


Figure 6: *SPU based multicast integration*

The easiest approach to integrate Multicasting into the Chromium framework is the implementation of special SPUs. One of these acts as sender and has to be placed in the client's SPU chain in front of the Tilesort-SPU. A second Multicast SPU acts as receiver and will be inserted at the head of the server's SPU chain. We used the following approach: the sender will cut all appropiate parts out of the incoming stream, transmits them in parallel to a group of servers using the multicast channel and finally the receivers will insert the cut commands at the original positions into the stream. It is necessary to synchronize part of the stream transmitted via multicast with the unicast streams. Special commands are inserted into the stream forwarded to the Tilesort-SPU to manage this problem. These commands mark the positions where multicast sequences were cut out and must be re-inserted at the server side. Figure 6 shows the communication channels used for this approach.

Alternatively Multicasting can be integrated by modifying the Tilesort-SPU and the server nodes. The aim is a revised version of the Tilesort-SPU which will send appropiate buffers using the multicast channel. Only serializing commands will be transmitted through unicast connections. Because both streams are handled by the Tilesort-SPU no special synchronization commands are necessary. On the other hand, the modification of Tilesort would be difficult because the internal data handling was optimized for existing unicast connections. Figure 7 illustrates this approach. The advantage of the SPU based approach is that only minimal modifications of the existing code are necessary. On the other hand, the multicast connection can only be used by the integration of additional SPU's. This leads to an overhead because additional function calls become necessary for all commands that need to be processed with respect to system state.

We implemented both variants. The SPU approach leads to slightly faster rendering because in the second approach we were not able to eliminate some inefficient data handling within the Tilesort-SPU. On the other hand, a satisfying synchronization of the image generation between tiles could not be realized. This is because of the way the concatenated Tilesort-SPU handles buffers that contain serializing commands. Therefore we designed a new SPU from scratch dedicated to replace Tilesort-SPU. Important features of the OPT-SPU will be descibed later in this paper.

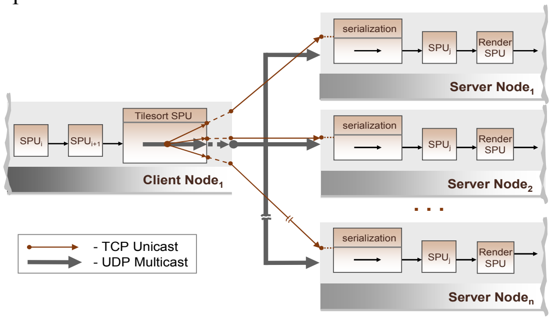


Figure 7: *Network layer based multicast integration*

Our implementation of the multicast connection type uses UDP sockets. Therefore it can only be used on IP Ethernet. Because UDP transmissions are not secured by design like TCP, we developed a simple and fast protocol. This is used by sender and receiver nodes to keep the multicast communication consistent. The sender serially numbers all packets sent. Thereby receivers can detect the loss of packets or a wrong order of received packets. Typically, the sender transmits a configured count of packets and will get receipts from all receiving servers of the related multicast group. In the case of lost packets the transmission will continue until a configured credit limit is reached. If a server node still misses some packets it will send a reorder message to the UPD sender through a secure TCP channel. In this case the sender stops the multicast transmission immediately and initiates the retransmission of the missing packets to the related node through the TCP connection. Instead of establishing new TCP sockets we use the existing TCP channels supported by the Chromium network layer to secure the multicast channel as well as to communicate multicast related information e.g. joining the multicast group and so on. Figure 8 shows the message flow if a packet is lost.

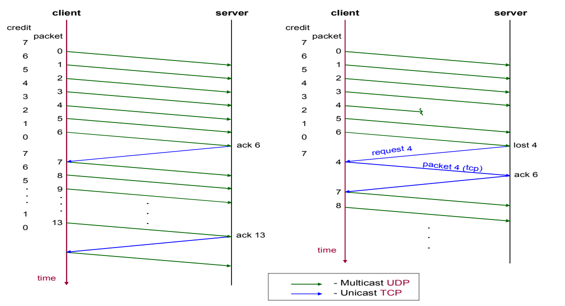


Figure 8: *Protocol used to secure a multicast channel*

In addition to the implementation of the multicast connection type we had to modify some other parts of the chromium framework. These are the receive()-function called by all nodes to accept and handle incoming multicast packets, the configuration manager to allow the definition of multicast related settings and the nodes itself to make use of the multicast connection type. At this time, only one multicast channel is supported per running instance of Chromium. It is managed by the first server configured to the OPT-SPU. Typically this channel is used to send suitable packed buffers to all servers in parallel. Furthermore, servers can only join one multicast group. This restricts the servers to accept multicast buffers from one sender only. Remembering Molnar's classification [MCEF94] of parallel rendering systems, only Sort-First configurations can be built using our implementation. In this case the client and all server nodes join a unique multicast group. Theoretically each server may join different multicast groups. This would allow Sort-Last configurations, but for our purpose to drive tiled displays with a Sort-First cluster the implemented functionality is sufficient. As will be described in the next section, for Sort-First clusters it is better to transmit each geometry buffer to all servers immediately instead of separately by tile-sorting.

4.2. Replacement of Tilesorting by Culling

We modified the Tilesort-SPU to transfer geometry buffers through the multicast channel. Analyzing this SPU we were pleased to see very effective handling of the incoming stream to optimize unicast transmissions. Unfortunately, the integration of a new "bucketing mode" for Multicasting causes many changes and leads to confused and sometimes ineffective code. Therefore we decided to write a new SPU supporting the Multicast mode only.

The way it works is pretty simple. Most parts of the incoming stream are encoded to packed buffers and forwarded to all tile servers using the Multicast channel. This includes all GL commands except the serializing instructions. Most commands causing state changes were recorded using the state tracker only for the server controlling the multicast connection. Some state commands need to be handled different for each server, e.g. viewport settings. The most important fact is that all blocks containing geometry will only be transmitted in parallel through the Multicast channel. Thereby the consumed bandwith will be significantly reduced. Additionally, we transmit all texture data and other suitable commands through the multicast channel. Multicast buffers will be flushed if no free space is left or if a serializing command has to be sent.

Our way to handle the incoming GL stream generated by the application makes the expensive classification of bounding rectangles against tile borders unnecessary. It is no serious drawback to transfer all GL primitives to all tile servers regardless of the geometric situation. First, the Multicast transmission causes no extra costs if the geometry would be rendered outside the tile one server is responsible for. Second, if the object or the camera will be moved later, this geometry may be visible on these tiles. In this case, if the multicast transmission is combined with stream caching as described in the next section, it may be very useful to immediately transmit all geometry rendered for the first time.

We use a conventional frustum culling method to avoid situations where servers will process geometric primitives outside their viewports. In contrast to tile-sorting our frustum culling procedure runs on the server side. It has the same effect as tile-sorting: no geometry outside the tile borders is drawn. However, the work is done on the low utilized servers and large geometry buffers are sent only once in parallel. In this way the server sided frustum culling takes a lot of load from clients running the OPT-SPU and the application. Consequentely the client can process incoming streams faster.

Finally, we integrated a similar functionality to avoid that geometry buffers outside the display borders will be processed. The client side frustum culling method can be activated by configuration if the application does not implement its own sophisticated frustum culling function.

The results presented in section 6 show a dramatic acceleration reached by Multicasting.

4.3. Synchronized Rendering

In general, the actual frame will be rendered to the back buffer whereas the content of the front buffer is displayed. When SwapBuffers() is called, the back buffer becomes visible. Unfortunately, the back buffer does not become visible immediately because first the graphics pipeline has to be flushed to guarantee that the rendering of the actual frame is finished. The time each server needs to finish the rendering of the received command streams varies. Therefore the synchronization of the moment when SwapBuffers() is called by the Render-SPU does not solve the problem.

To provide a coherent impression of images, especially for low frame rates, we developed a software method to synchronize the visible buffer swap between all servers. Basically, all Render-SPU's will intercept the call of SwapBuffers(). When SwapBuffers() is received, a call of glFlush() will be done instead. glFlush() blocks until all previous commands were executed by the hardware. Next, the Render-SPU sends an acknowledgement through a TCP connection back to the OPT-SPU. When the OPT-SPU has received this acknowledgements from all servers, a subsequent call of SwapBuffers() is initiated by sending a message through the Multicast channel to trigger a synchronized call of SwapBuffers().

We added a configureable threshold value to define a minimum frame rate. The synchronization is activated when the rendering performance becomes lower than the threshold value. We noticed that values of 10 deliver good results.

4.4. Stream Caching

If the geometric situation requires that identical GL commands have to be transmitted to a group of servers this can be managed using Multicasting. As described in the last sections, the parallel transmission to all servers needs only the time a single transmission to one server will take. Unfortunately, Multicasting cannot be used to eliminate the redundant traffic caused by identical command sequences in the GL stream. For instance, parts of the scene may be used several times to render a frame. The effective transmission of repeated stream parts can be managed with stream caching.

An implementation of a stream cache was presented in [DKK02]. The main idea of this approch is to compile a display list to encapsulate suitable objects. A checksum is used to identify blocks of geometry (glBegin/glEnd). The cache is controlled using the computed checksum. For each part of the stream to be cached a new display list is compiled. To do this, a new display list identifier is allocated on the related server. Later, all data will be sent and compiled into the list by the graphics driver. The next time the object is detected at the client, it will be replaced in the outgoing stream by a call to glCallList(). Therefore only this small instruction needs to be transmitted to each server. Once the list is compiled the rendering of the geometry can be initiated by transmissions of small commands.

Unfortunately the compilation of the lists may take some time, so short pauses may occur on cheap hardware. We noticed that our graphics hardware (Radeon X600) has problems to handle a large number of lists. The time for display list compilation increases significantly when the driver needs to swap lists between the onboard memory and the system context. Therefore we developed a stream cache without the use of display list functionality. The stream cache works transparently on packed buffers. This way it can be used with both multicast and unicast connections.

We placed the cache control function in the OPT-SPU's flush function that is responsible for the transmission of filled buffers. A checksum is created for each buffer that contains an configurable amount of data. If a suitable buffer is found a free cache position will be computed. If one is found the buffer is sent immediately encapsulated in the cache control command crCache(). This command is inserted into the outgoing stream at the objects position.

We experimented with different hash functions widely used to compute unique and secure checksums, e.g. MD4 and MD5. We found that these functions are too slow to be used for our purpose. So we developed a fast hash function to identify the geometry buffers. The checksum is computed based on the coordinates of the 3D bounding box provided by the packer library and the binary buffer content of each D-th word. D is an increment of the pointer that addresses memory words to be read by the hash function. The larger the increment is chosen the smaller are the costs of the hash function. On the other hand, large increments would cause the function to ignore small differences in the packed buffer. Good results were achieved with odd increments lower than 7.

Another hash function is used to compute a cache position for new buffers in the hash table. If a collision occurs the next position is tried. If this is also occupied, a cost function is called to decide if the older object should be removed from the cache or the caching of the new object should be denied. The cost value is computed from statistic information stored with the object, e.g. age, size, total count of calls and last frame the object was drawn.

The storage area of the cache is a part of the server node. The implementation was added to the so called serialization function. The set of cache control functions will be processed when received buffers are unpacked. The command set includes functions for caching and calling buffers and for cache control. Figure 9 shows the components of our stream cache.

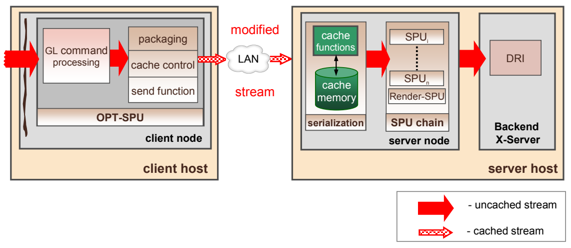


Figure 9: *Structure of the Stream Cache*

In contrast to the display list approach our cache stores packed buffers in the server context. Note that the cache content of all servers is the same. When cached buffers are called, each buffer is decoded as it would be received. No time is needed to compile lists and the cache works as fast as possible as long as enough memory is available on the server side. Look at the results to see the speed-up when the stream cache is used.

4.5. Occlusion Culling

In general, occlusion culling methods are used to avoid the rendering of geometry probably hidden by other non-transparent objects. Potentially it can be used to reduce the complexity of the incoming GL stream on the client side. The choice of a suitable occlusion culling method depends very strongly on the structure of the scene. For streaming purposes two appoaches can be used.

First, each 3D bounding box of a packed geometry buffer can be queried subsequent to the frustum culling procedure. This can be done using the occlusion query extensions supported by common graphics cards. To use this functionality a local rendering context is required. Each non transparent object is rendered in the local context as fast as possible, e.g. without lighting and texturing. Based on the content of the frame buffer the occlusion query method can decide the potential visibility of new buffers by simulated rendering of the bounding box. If pixels would be created, the geometry needs to be drawn to the local context and has be multicast to all servers. Because of high latency we did a threaded implementation of the function used to handle the local context.

Second, each frame can be evaluated using a color histogram. In contrast to the method describe above, a exact set of visible, non transparent objects is computed. It is independent from the order the geometry is placed in the stream to be processed. Unfortunately this approach has some drawbacks. First, the stream data of a complete frame must be stored at the client side. Second, the whole scene needs to be rendered to the local context. This is done as fast as possible, e.g. without lighting, texturing and aliasing. Each glBegin() / glEnd() block is drawn using a unique color value. At the end of a frame, the color buffer contains the exact visibility set identified by color values. The main problem is that the visibility set cannot be processed until the last geometry was drawn. Also, a slow readback of the entire frame buffer is necessary. Based on the read pixels the computation of the histogram takes place. This histogram is finally used to assemble the outgoing command stream to render only the visible set of blocks from the stored, incoming frame. To avoid unuseful state changes because geometry was cut off, we use state differencing as described in [BHH00].

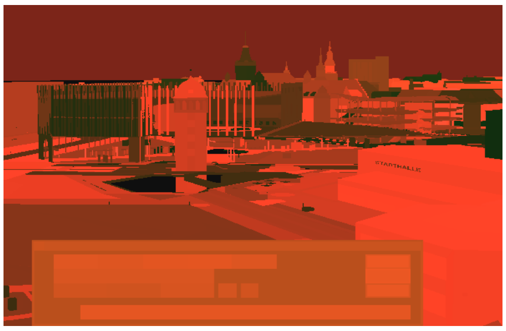


Figure 10: *Occlusion Histogram of the urban scene*

We did a threaded implementation of the histrogram approach. One thread handles the incoming stream. This includes the local rendering and the assembly of the buffer structure to store the frame. The second thread computes the histrogram from the pixel data rendered in the first thread. A third thread assembles the send buffers using the histrogram and state differencing. Additionally this thread sends the buffers to all servers using Multicast and if necessary the unicast channels. The threaded implementation allows the parallel computation of two frames on dual core hardware. Figure 10 shows the occlusion histogram of a given scene.

5. Optimized Cluster Configuration

In this section we describe the configuration of our rendering cluster using the optimizations presented above. The cluster is dedicated to drive a display with 12 tiles aligned in a 4x3 matrix. The entire cluster runs on SuSE Linux 10.0, XFree 4.3 and DMX. The version of the Chromium framework we use is 1.7.

We use mainstream PC's, each equiped with PentiumD processor, 2GB memory and ATI X600 graphics card. Our approach to minimize unicast communication by tracking most parts of system state only for the first server causes all hosts to be identically equiped and configured. In contrast to a default Chromium configuration all hosts of our cluster are connected to two Gigabit Ethernet switches. Both networks are configured with separate private address ranges. This was done because we noticed that the ongoing traffic generated from OS, X and DMX interferes with the multicast transmissions. This causes a high count of lost packets to be retransmitted by related unicast communication. The total throughput of the multicast channel is best if it runs on a dedicated physical network.

The second difference to common configurations based on Tilesort is that we use a separate host to run the client node. We moved the client node to an extra PC because of two reasons. First, the processor load of a shared PC typically used to run the application, the client, one server node and mostly the DMX server is naturally very high. As described in section 2 the client's CPU load is a significant bottleneck. Second, if a host is shared it has to act as multicast sender and receiver. This finally leads to a poor performance of the multicast channel because most switches have problems to handle both streams on a port.

The OPT-SPU supports rendering to an application window displayed on a spanning DMX screen. The DMX server runs on the server node driving the lower-left tile. Hardware devices used for interaction are connected to the DMX server host. A special python script is used to autostart Chromium. Because network port numbers are randomly generated the parallel execution of several applications on the cluster is supported.

Additionally we added a procedure to the Render–SPU allowing quad-buffered frame sequential stereo streams generated by applications written for shutter hardware to be rendered on passive stereo displays. To do this, per tile two servers are used to render the stereo image pair. Both servers show the same part of the spanning DMX workspace. Only the drawing areas of the application window overlapped by the GL windows rendered directly to the background X-Servers differs between stereo pairs. To achieve this, the rendering to the left and right buffers is tracked and subdivided.

6. Results

We measured the effects of each implemented method isolated as well as for useful combinations. Therefore scenes with different characterizations were rendered. The figures contain selected measurements to be suitable to show the most important advantages of the work we have done. The frame rate measured when rendering on one server directly using DRI instead of Chromium is included in some diagrams for reference.

Figure 11 shows the frame rate rendering the plate fullscreen (Figure 2) for a different number of tiles configured. In contrast to Test-all-Tiles and Broadcast the performance of Multicast is nearly constant. The effective bandwidth utilized transfering the stream through the multicast channel is about 550MBit/s. In contrast, the bandwidth consumed running Test-All-Tiles is only about 250MBits/s because the client's CPU load is at 100%. This means, that even though the network would be able to carry more data to achieve faster rendering, the client cannot pack buffers fast enough. Note, because the plate is visible on the center tile only when rendered to 3 servers, the rate reached with Test-All-Tiles is nearly as high as for one tile only.

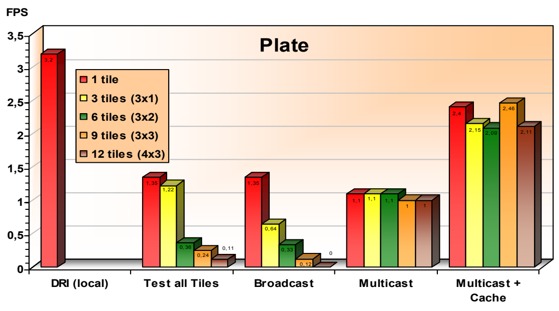


Figure 11: *Results rendering the plate fullscreen*

Similar values have been measured rendering with 12 servers configured. Figure 12 shows the framerate of the same program when the window has been scaled to be displayed on different tiles of the DMX workspace.

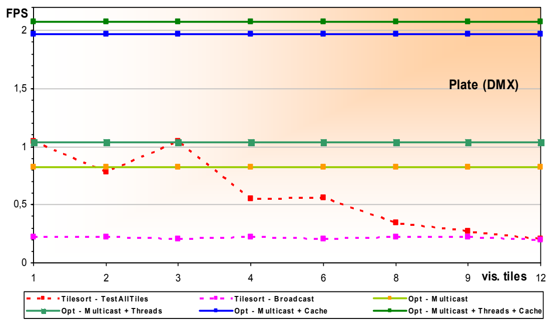


Figure 12: *Rendering plate to a tiled DMX window*

Additionally both diagrams include the frame rate measured with Multicasting in combination with stream caching. Because the object is rotating only, the cache delivers best results. The lack of performance compared to local rendering is mostly caused by the costs of the cache control functions, e.g. computation of checksums. The total bandwidth utilized is about 6,5 MBits/s instead 550MBit/s without caching.

Figure 13 shows the results of the urban scene (Figure 4). The entire urban model is drawn for generation of each frame because the application integrated frustum and occlusion culling methods are disabled. It contains about 180k polygons textured with 75MByte of image data. The rendering performance goes up once more when our additional optimization methods (frustum and histogram occlusion culling) were used. Because knowledge about the structure of scene is not available the frame rate of scene graph based culling methods processed by the application (16 FPS) cannot be reached. Once more, note the nearly constant frame rate of our Multicast modes for different cluster sizes and remember that the tile update by swapping the GL buffers occours synchronously.

We measured the time a frame update will need from the moment the first tile becomes visible until the last tile is displayed using a video camera. For the situation of Figure 13 the Broadcast mode needs 13 video frames (0.52s) to update all tiles. Test-all-Tiles need 0,4s. In contrast, all tiles were updated running Multicast modes within the duration of one video frame (0.04s). We found, that users will not tolerate delays of more than 0.06 to 0.075 seconds.

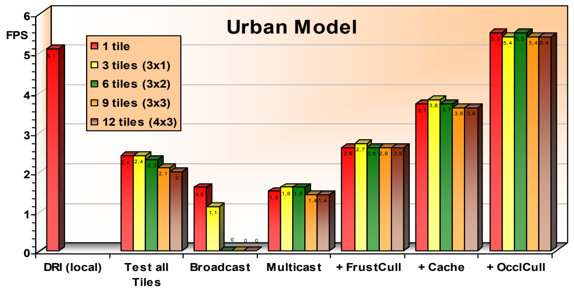


Figure 13: *Results rendering the urban scene*

Finally, in Figure 14 the relation between the client's CPU utilization and the network throughput is shown. The measurement was done rendering a microscope with different level of complexity (resolution factors of GLU quadric primitives). First, note that Test-All-Tiles is not able to fully utilize the available Gigabit network whereas the OPT-SPU does. Second, note that the cache reduces the client's processor load significantly. Because at one level of complexity a static geometry is rendered, the gap between cached and native Multicasting is caused by the load of packaging and sender routines only.

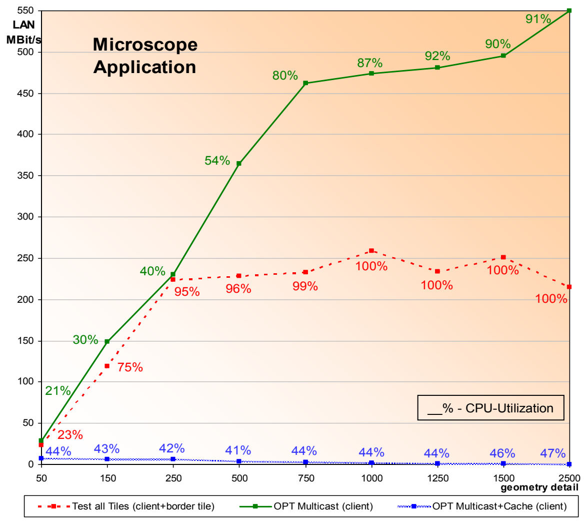


Figure 14: *Throughput and client's CPU utilization*

7. Conclusion

The most important improvement that we achieved is the nearly constant rendering performance of the graphics cluster when additional tiles are added. Also, different scenes will be rendered at nearly constant frame rates independent of the size of the application window displayed on spanning DMX workspaces. This has been achieved mostly by the integration of Multicasting into the Chromium framework.

Second, a software based method was developed to synchronize the rendering of all tiles. It is triggered by messages to control the visible effect of SwapBuffers() transmitted in parallel to all servers with Multicasting.

Third, the implementation of a memory based stream cache allows very large scenes to be rendered with adequate performance. The necessary bandwith to transfer the command stream of a frame was significantly reduced. Also, the cache mostly reduces the client's CPU load. This leads to better rendering speed overall.

The combination of these new features improves the performance of all applications. We developed additional methods to accelerate applications that do not support frustum and occlusion culling. Finally a fast converter was implemented to enable applications written for shutter hardware to be displayed on passive stereo displays. This can be done in combination with DMX too.

8. References

[ISH98] IGEHY H, STOLL G, HANRAHAN P.: The design of a parallel graphics interface. *Proceedings of SIGGRAPH 1998*, pages 141-150, 1998.

[HBEH00] HUMPHREYS G, BUCK I, ELDRIDGE M, HANRAHAN P.: Distributed rendering for scalable displays. *IEEE Supercomputing 2000*, 2000.

[HEB*01] HUMPHREYS G, ELDRIDGE M, BUCK I, STOLL G, EVERETT M, HANRAHAN P.: WireGL - A scalable graphics system for clusters. *Proceedings of SIGGRAPH 2001*, pages 129-140, 2001.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK F., AHERN S., KIRCHNER P., KLOSOWSKI J.: Chromium: A streamprocessing framework for interactive rendering on clusters. *In Proceedings of SIGGRAPH*, pages 693–702, 2002.

[BHH00] BUCK I, HUMPHREYS G, HANRAHAN P.: Tracking graphics state for networked rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 87-95, 2000.

[MUE95] MUELLER C.: The sort-first rendering architecture for high-performance graphics. S*ymposium on Interactive 3D Graphics*, pages 75–84, 1995.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[DKK02] DUCA N., KIRCHNER P.D., KLOSOWSKI J.T.: Stream Caches: Optimizing Data Flow. In *Visualization Clusters. Commodity Cluster Visualization Workshop*, *IEEE Visualization*, 2002.

[BHH00] Buck I., Humphreys G., Hanrahan P.: Tracking graphics state for networked rendering. In *Eurographics/ SIGGRAPH Workshop on Graphics Hardware*, 2000.