Rendering on Tiled Displays using Advanced Stream Caching

Mario Lorenz, Guido Brunnett and Marcel Heinz
Computer Graphics and Visualization, Chemnitz University of Technology, Germany
{ Mario.Lorenz | Guido.Brunnett }@informatik.tu-chemnitz.de, Marcel.Heinz@s2001.tu-chemnitz.de

Abstract

To render complex scenes on tiled displays efficiently modifications to the Chromium framework have been proposed that lead 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 a transparent stream cache can be used that accelerates the multicast communication channel. It can be profiled that applications that do not use display lists or VBO's to control the rendering put a heavy load on the client host because the cache control has to compute a checksum for the identification of repeated command sequences. In this paper we analyse the possibilities to improve the stream caching for unmodified GL applications. As our main result we introduce a new GL extension to control the stream cache with minimally modified applications.

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

1. Introduction

Since today's *tiled displays* are driven by clusters of PC's parallel rendering becomes necessary. In general, parallel rendering processes running on different cluster nodes take advantage of the cluster best, but the design of parallel rendering applications is often difficult and expensive because one has to deal with complex data distribution strategies and synchronization models. As an alternative to redesign existing applications a special parallel rendering interface can be implemented that allows running 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 of this approach is the internal handling of very large data streams that carry simple graphical commands delivered by the applications to the virtual pipeline.

In this paper we briefly recall recent optimizations [LB06] integrated in the *Chromium* framework [HHN*02] to achieve non redundant network communication of most stream parts using stream cached multicasting. Since the necessary computation of cache ident checksums may limit the acceleration of scenes that contain mainly small or deforming objects the main focus of this paper lies on advanced stream caching. We introduce a GL extension intended for application based control of the stream cache. Using this extension it is possible to overcome the bottleneck caused by repeated computations of cache id's. Since only minimal changes to an existing graphical application are necessary to increase the overall performance significantly this approach may be interesting for all users who need very easy and fast porting of existing code to a rendering cluster for driving a tiled display.

2. Related Work

Based on Stanford's *WireGL* [HEB*01] the widely known *Chromium* framework was designed. This software allows building a parallel rendering system for high performance remote rendering [HBEH00]. In Chromium a client/server architecture based on the concatenation of so called software nodes running stream processing units (SPU's) is realized. All nodes may be placed on different cluster hosts. Using the so called *Tilesort-* and *Render-*SPU's, a Sort-First architecture [MCEF94] can be configured to allow native OpenGL programs to drive a tiled display [MUE95].

Unfortunately, the overall graphical performance of such clusters does not scale very well. It can be seen that the frame rate becomes lower with every tile added to the cluster. If objects are visible on many tiles the frame rate may also be negatively affected. This behaviour is caused by the design of the Tilesort-SPU and the network layer. The Tilesort-SPU implements a screen space decision method for splitting the incoming command stream into several streams [BHH00] that are sent to the servers with point-to-point (*unicast*) connections only. This implies the redundant transmission of stream parts that must be sent to different nodes. With respect to the overall performance this is very harmful.

3. Previous Work

To improve the overall performance of Chromium the most important task was the effective elimination of redundant transmissions. This reduced the consumed bandwidth as well as the clients processor load because less buffers need to be packed and sent. First, we integrated *Multicasting* into Chromium by the implementation of a

© The Eurographics Association 2007.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

new connection type that allows sending incoming GL commands to all render servers at once. To transfer geometry buffers through the multicast channel we designed a new SPU (*OPT-SPU*) to replace the Tilesort-SPU. This works in the following way. Most parts of the incoming stream are encoded to packed buffers and forwarded to all tile servers using the UDP Multicast channel. Thereby the consumed bandwidth will be significantly reduced. The so called serializing instructions, e.g. viewport settings, are handled separately using the state tracker [BHH00]. Since the expensive tile-sorting of geometry blocks is no longer necessary we implemented a conventional frustum culling method to avoid situations where servers will process geometric primitives outside their viewports. It has the same effect as tile-sorting: no geometry outside the tile borders is drawn, but it runs on the low utilized servers.

Unfortunately, Multicasting cannot be used to eliminate the redundant data transmission 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 approach is to compile a display list to encapsulate suitable objects. The next time the object is detected by the client, it will be replaced in the outgoing stream by a small instruction. Unfortunately the compilation of the lists take some time, so short pauses may occur.

Therefore we developed a stream cache without the application of display list functionality. In contrast to the display list approach our cache works transparently on packed buffers. It stores cached packer buffers in the server context. The set of cache control functions will be processed when received buffers are unpacked. Figure 1 shows the components of our stream cache.

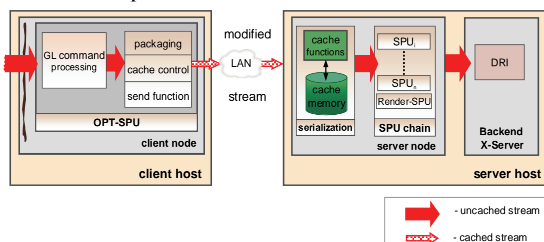

Figure 1: *Structure of the Stream Cache*

Note that the cache content of all servers is identical. We placed the cache control function in the OPT-SPU's flush function that is responsible for the transmission of filled buffers. Buffers are identified with a checksum. When a cached buffer is detected in the incoming stream next time its content will be ignored. Instead only the small *glCallCacheCR()* command will be transmitted to the servers which initiates the extraction of the object from the storage area of the cache at the server side.

To accelerate the rendering of large cached objects we developed a cache extension based on OpenGL's VBO functionality. The basic idea is as follows: A method running as part of the cache identifies suitable cached Chromium buffers using a cost function and transforms the covered GL commands to VA and EA structures. After that a VBO will be compiled in the background. The next time the cached data set is called, the cache control function initiates the rendering of the related VBO instead of unpacking the cached Chromium buffer.

The transparent stream cache reduces the consumed network bandwidth significantly. It is suitable to improve the rendering performance of most types of scenes. A detailed description of the methods introduced in this section can be found in [LB06] and [LBH07].

4. Advanced Stream Caching

The most difficult and expensive task in transparent stream caching is the repeated identification of already cached packer buffers. However, even fast identification functions impact the CPU utilization of the client host significantly. That is a problem because the client's CPU utilization is always a bottleneck. But, caused by the design of a stream cache this task must be processed for every buffer to be handled. Therefore we designed different functions that work on a set of sample data from the binary buffer content and its 3D bounding box coordinates. A discussion of the results can be found in [LBH07].

The rendering performance will benefit from the transparent stream cache if the related block can be called from the storage area instead of transmission. On the other hand, the cache may lower the rendering speed if the related block will be cached and only called once. The decision to cache a geometry block can only be made based on a simple heuristic, e.g. count of commands and data size. There is no other way because the cache has no information whether the block will be transmitted again. Additionally, in the case of a cache collision the cache cannot estimate how long the already cached object will be still needed. Therefore a heuristic is used to decide to cache the new block or to leave the cached one on its position. Obviously, a stream cache will accelerate the transmission of native, even complex GL-streams significantly but it will never reach the performance of an application based acceleration using display lists or VBO's.

If the application would have the chance to manage the cache objects itself, the repeated computation of checksums would no longer be necessary. This will eliminate the drawbacks of the automated transparent stream cache. The goals of such an application controlled stream cache are a significant acceleration of the overall rendering speed and the elimination of artefacts resulting from the rendering of incorrect identified geometry by the identification function. Remember that the identification function only may use samples of the buffers and this may lead to identification errors. The application may render both, managed cache objects and native geometry blocks mixed. The cache would handle native blocks in the usual way, but managed blocks directly. Managed blocks will produce no overhead because the identification is given by the application.

© The Eurographics Association 2007.

Such a stream cache has different advantages. First, the identification of already cached objects is given. This will improve the overall rendering performance of the cluster most because cached objects can be called directly with the given identifier. In this case the application does not specify the geometry data as sequences of vertex, color, normal and texture commands. Instead, a simple cache control command will be specified. Therefore it is not necessary that the packer encodes the geometry commands into a geometry buffer. Obviously this reduces the work load of the client significantly and all other tasks will benefit.

Next, the application has the chance to mark objects that will not benefit from caching, e.g. such that will be drawn only once or that change their geometry. This will avoid the expensive computation of checksums by the cache control function as well as the substitution of other, suitable objects in the case of a hash collision. Additionally, the application can send control commands to remove objects from the cache which will no longer be needed. This will reduce the memory utilization as well as the count of hash collisions.

The application based cache control does not replace the transparent stream cache but it is a very useful extension. As described above Chromium based rendering clusters have no information about the structure of the 3D scene and the rendering process. Based on GL streams only suitable data sets, e.g. *glBegin()-glEnd()* blocks can be analyzed and managed with a significant overhead. In contrast, the application can decide based on the structure of the scene and the information from the rendering process which objects should be cached, how long they should remain in the cache and which objects should not be cached. Especially the knowledge about the future of scene parts (which Chromium does not have) is very important for caching. Since the application often knows which objects will be always visible and which ones will be rendered only for a short time, it is able to manage the cache more efficient as a transparent stream cache ever can. This allows specific cache strategies and, of course, best acceleration.

The drawback of this method is the need for a modification of the graphical application which is not always suitable. In addition, our stream cache handles only geometry stored in *glBegin()-glEnd()* blocks. The method is not applicable if the application specifies the geometry different from that.

5. Cache Control API

The application based cache control was developed with respect to the following aspects:

1. Minimal code modifications of existing GL-applications: The cache control functions are easy to integrate. There is no need for major changes of the render process and of the internal data representation of 3D-objects. The specification of *glBegin()-glEnd()* blocks remains unchanged. Furthermore, there are no restrictions for the specification of vertex data as given in the case of vertex arrays. This minimizes the complexity of conceptual modifications particularly.
2. Simple and efficient API: The cache control interface is simple to use. The application uses only a small set of functions to perform all necessary management tasks. In particular, the specification and handling of cache objects is very efficient.
3. Flexibility and extensibility: The design of the API in form of a GL extension simplifies the integration of the functions into the Chromium framework. All functions are handled by Chromium in the usual way. This allows the operation of user-specific SPUs in front of the OPT-SPU. The integration of additional API-functions requires minimal code changes.

The API functions are used to manage cache objects in a special storage area on the server side of the stream cache. The configurable storage area is organized based on consecutive identifier numbers. Each identifier references exactly one object managed by the application using API functions. The API allows an object to be stored, called from the cache and deleted. In the trivial case the application can fill the storage area of the cache with objects numbered from 1 to *GL_MAX_CACHE_ID_CR* in ascending order. No hashing is necessary. Note that the identifier *0* is reserved to mark objects that should be handled in a particular way described later in this section.

Since Chromium is seen from the application as a usual implementation of the GL interface the cache control API has been designed and implemented in form of a GL extension (*GL_CR_cache_control*). The extension is defined in the namespace of Chromium and uses values from the reserved Chromium enumeration interval (*0x8AF0-0x8B2F*). It provides the following functions:

```
typedef void (APIENTRY *glCacheBeginCRProc) (GLuint id);
typedef void (APIENTRY *glCacheEndCRProc) (void);
typedef void (APIENTRY *glCacheCallCRProc) (GLuint id);
typedef void (APIENTRY *glCacheDeleteCRProc) (GLuint id);

extern void APIENTRY glCacheBeginCR(GLuint id);
extern void APIENTRY glCacheEndCR(void);
extern void APIENTRY glCacheCallCR(GLuint id);
extern void APIENTRY glCacheDeleteCR(GLuint id);

#define GL_MAX_CACHE_ID_CR 0x8B2F .
```

First, the application should ask the cache for the number of cache objects. This is done using the *glGet()* function with the argument *GL_MAX_CACHE_ID_CR*. A returned value of *0* indicates a disabled stream cache.

The specification of a new cache object is done using the function pair *glCacheBeginCR(id)* and *glCacheEndGL()*. Between the calls of these functions an arbitrary number of *glBegin()-glEnd()* blocks containing the GL commands allowed there may be specified. *glCacheBeginCR(id)* addresses the related cache object using the argument *id*. If there is no cache object with the identifier *id* when *glCacheBeginCR(id)* is called, all subsequent GL commands will be stored by the packer in geometry buffers until *glCacheEndCR()* is received. The filled buffers will be transmitted through the multicast channel to all servers in parallel and stored in the stream cache. Finally, the stored cache object is called to initiate the rendering of the geometry. If the cache already contains an object with the

© The Eurographics Association 2007.

identifier *id* when *glCacheBeginCR(id)* is called, all subsequent GL function calls will be ignored until a corresponding *glCacheEndCR()* is received. Instead of packing the GL commands only the small *glCallCacheCR(id)* is packed and transmitted. This initiates the rendering of the cached geometry at the server side.

The selective *non-caching* of geometry is specified calling g*lCacheBeginCR(0)*. In this case no data are handled by the cache. This means that no checksum is computed and no packed buffers are stored in the cache. Instead of that the commands are packed into Chromium buffers and sent to the servers in the usual way until *glEndBeginCR()* is called. In contrast to the selective caching of geometry any GL commands are allowed here. In this way a set of geometric objects that should not be cached, e.g. deformable solids, and the related state changes can be transmitted to the servers efficiently.

The function *glCallCacheCR(id)* is used to extract the cache object *id* from the storage area at the server side directly. It is functionally equivalent to the command sequence *glCacheBeginCR(id); ... glCacheEndCR();* but much more efficient. This sequence ignores all GL commands between the function calls in the case the object is already cached, whereas *glCallCacheCR()* is represented by one call only. While the sequence is intended for a minimal integration of the cache control into existing applications as shown below, *glCallCacheCR()* requires a more complex management of cache identifiers within the application. The instruction *glCacheDeleteCR(id)* is used to remove the object with the identifiers *id* from the cache at all servers. After this another object can be stored with the freed *id*.

All API functions handle errors in the usual way of GL. For instance, calling *glCallCacheCR()* with an invalid *id* (if no related object is held by the cache or if the *id* value is out of range) will generate *GL_INVALID_VALUE*. A forbidden function call, e.g. calling *glCacheDeleteCR()* between *glCacheBeginCR()* and *glCacheEndCR()* results in a *GL_INVALID_OPERATION* error.

The following example shows a typical function to draw a part of the scene using some *glBegin()-glEnd()* blocks:

```
void DrawObject (...)
{
        ...
        /* set OpenGL state */
        ...
        /* specify geometry as sequence of
         * Begin/End-Blocks */
        glBegin(...);
        ...
        /* geometry data */
        ...
        glEnd();
        ...
        glBegin(...);
        ...
        /* geometry data */
        ...
        glEnd();
}
```

The next code shows an integration of the cache control functions that requires only minimal code modification to perform a specific caching of suitable objects. In this case all geometry data is specified and packed every time the function *DrawObject()* is called but in contrast to the unmodified code the expensive computation of checksums that is necessary for the identification of the geometry data is not performed by the cache.

```
static bool cache; /* indicates an active cache control */
void DrawObject (...)
{
          ...
          /* set OpenGL state */
          ...
          if (cache) {
                    glCacheBeginCR(object_id);
          }
          /* specify geometry as sequence of
           * Begin/End-Blocks */
          glBegin(...);
          ...
          /* geometry data */
          ...
          glEnd();
          ...
          glBegin(...);
          ...
          /* geometry data */
          ...
          glEnd();
          if (cache) {
                    glCacheEndCR();
          }
}
```

Such a modified function *DrawObject()* is also used to bypass the stream cache for the contained geometry. To achieve the so called non-caching the argument *object_id* has to be *0*. The overall rendering performance benefits mostly if the cache control extension is used in the following manner because no GL commands are sent to Chromium if the object is already stored in the cache:

```
static bool cache; /* indicates active cache control */
void DrawObject (...)
{
        static int object_cached=0;
        ...
        /* set OpenGL state */
        ...
        if (cache && object_cached) {
                glCacheCallCR(object_id);
        } else {
                if (cache) {
                        glCacheBeginCR(object_id);
                }
                if (!object_cached) {
                        /* specify geometry
                           as sequence of
                         * Begin/End-Blocks */
                        glBegin(...);
                        ...
                        /* geometry data */
                        ...
                        glEnd();
                        ...
                        glBegin(...);
                        ...
                        /* geometry data */
                        ...
                        glEnd();
                }
                if (cache) {
                        glCacheEndCR();
                        object_cached=1;
                }
        }
}
```

It differs from the minimal variant in the handling of the flag *object_cached*. It can be seen that the application has to manage the cache state for all related objects. In complex rendering libraries this cannot be done as simple as in this example. Therefore the first implementation may be used to get better results fast.

© The Eurographics Association 2007.

Our strategy is comparable to the integration of display list functionality into complex programs. In contrast to our approach the internal handling of display lists in Chromium is very complex. Chromium handles display lists with a so called display list manager in two different ways: First, a list can be stored at the client. The manager inserts the contained data into the stream sent to the servers. This means that all data will be transmitted every time a list is called by the application. This is inefficient. As the second option, Chromium will forward the entire list to the servers. This seems to be efficient. But, display lists are not limited to store geometry. A list may also contain commands for changing the GL state. This may be complicated if some kind of culling is performed to avoid geometry to be drawn that is not visible on the related tile. The consequence is that for all display lists the state has to be tracked which an expensive task is. Furthermore, we found that consumer graphics cards have major problems compiling a large number of display lists. This leads to inconvenient interrupts of the continuous rendering.

6. Results

We use a render cluster of 12 PC's (Intel Core 2 Duo CPU, 2GB RAM, Nvidia GeForce 7800). The application and the Chromium client node are running on an additional PC. The 13 PC's are connected with 2 switched Gigabit Ethernet networks whereof one is exclusively used for multicast transmissions. The cache and the OPT-SPU are integrated in the Chromium distribution 1.9.

A simple test program shown in Figure 2 was implemented to draw triangulated objects using a single *glBegin()-glEnd()* block. For each vertex the 3D coordinates and the coordinates of the surface normal are specified using *glNormal()* and *glVertex()* commands.

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

First we measured the frame rate for a set of objects with different complexity each with various cache settings. The unmodified program was used to benchmark the cluster without caching as well as with transparent stream caching (w/ and w/o automatic VBO generation). The transparent stream cache was configured to perform the computation of identifiers based on the binary content of whole buffers. In this case the hash function *ROLLADD* [LBH07] applies each byte of the buffer to build a checksum. To determine the acceleration of the overall rendering performance caused by the application based cache control we modified the application to support both variants as shown in section 5. Table 2 shows the measured frame rates. It can be seen that the application of the cache control improves the performance significantly. The minimalist modification of the OpenGL application program using pairs of *glCacheBeginCR()-glCacheEndCR()* achieves a speed-up between 10% and 30% compared to our transparent stream cache with activated VBO generation. As expected, the reasonable code modification using the *glCallCacheCR()* method generates the best rendering performance. The acceleration is between 160% and 350% whereas complex objects benefit mostly. The cache control extension achieves best results in combination with automatic VBO generation by the transparent stream cache. Running the render cluster with this configuration a speed-up between 250% and 1250% was measured. The results marked with * represent the maximum frame rate that our servers can render limited by the graphics hardware.

The columns named morphing geometry contain the results for the selective non-caching of major scene parts. To measure the acceleration of this method we modified randomly the vertex data continuously to ensure that the related *glBegin()-glEnd()* block will occur only one-time. In this situation the rendering was between 15% and 30% faster compared to usual stream caching due to the elimination of cache overhead, e.g. computation of useless checksums. In addition, a significant reduction of memory utilization was observed.

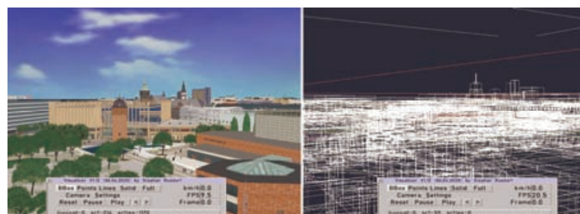

Figure 3: *The urban scene*

A second test was done rendering the urban scene shown in Figure 3. The scene has a completely different characteristic. It consists of about 140.000 objects with low complexity, e.g. parts of buildings, streets, trees and so on. But, the rendering of the scene graph requires a very high number of GL state changes including texture swapping. Since the given relation between object complexity and state changes is disadvantageous the use of the cache control extension cannot achieve such high speed-up values like in the previous test program. As shown in Table 1 a continuous walk-through runs as twice as fast as the rendering based on multicast transmissions without and about 40% faster than with transparent stream caching.

scene	w/o GL_cache_control_CR		w/ GL_cache_control_CR
	w/o stream caching	w/ stream caching	(variant glCallCacheCR())
urban model	6,7	8,3	11,5

Table 1: *Frame rates for the urban scene*

© The Eurographics Association 2007.

scene characteristics			static geometry							morphing geometry	
			w/o GL_cache_control_CR			w/ GL_cache_control_CR					
						minimal. modification		w/ glCallCacheCR()			
scene file	number of triangles	number of vertices	w/o stream caching	w/ stream caching	w/ caching + VBO gen.	w/ stream caching	w/ caching + VBO gen.	w/ stream caching	w/ caching + VBO gen.	w/ stream caching	non-caching
Hawaii	19602,0	10000,0	48,0	124,0	199,0	134,0	236,0	199,0	550*	23,0	30,0
Club	33700,0	16864,0	30,0	74,0	127,0	86,0	164,0	144,0	580*	14,9	17,3
Body	38756,0	19927,0	25,0	69,0	122,0	76,0	148,0	123,0	560*	12,5	14,4
Bones	45750,0	23623,0	22,0	60,0	103,0	67,0	131,0	116,0	570*	10,2	12,7
Bunny	69451,0	34834,0	14,7	40,0	62,0	48,0	78,0	90,0	560*	7,3	8,5
Horse	96966,0	48485,0	10,7	29,0	51,0	33,0	64,0	60,0	440,0	5,2	6,3
Football	121784,0	61742,0	8,3	21,0	41,0	24,0	52,0	49,0	470,0	4,2	4,9
Picard	146036,0	73014,0	6,3	12,0	19,0	16,0	23,0	43,0	239,0	3,2	3,8
Plate	599999,0	300970,0	1,7	5,6	7,8	6,2	10,5	14,0	84,0	1,4	1,8

Table 2: *Frame rates for different triangulated objects*

7. Conclusion

The application of a stream cache in combination with multicast transmissions within a Chromium render cluster enables a significantly higher rendering performance possible. Furthermore the consumed network bandwidth and the CPU utilization especially at the client host will be considerably reduced. But, the acceleration is limited by different aspects, mainly caused by the fact that a usual stream cache has not enough information about the graphical scene to perform an optimal cache strategy. This leads to computational and memory overhead. Unfortunately there is no chance to overcome these problems if the application cannot be modified because complex stream parts have to be identified continuously based on expensive checksums.

These drawbacks can be eliminated if the graphical application manages parts of the stream cache. This includes primary the selective caching and calling of parts of the graphical scene. For this purpose we developed an easy to use GL extension which acts as an interface to our stream cache that was implemented for the acceleration of multicast transmissions within the Chromium framework.

The *GL_CR_cache_control* extension can be used for a minimal invasive modification of existing applications. The most important advantage of this approach is that it makes no revision of the internal data representation necessary because any *glBegin()-glEnd()* block can be processed. This simplifies the integration of the cache control compared to the application of vertex arrays notable. Remember that vertex arrays and vertex buffer objects can only be used for the storage of vertex data. In this case the geometry needs to be specially organized. Additionally, the GL state has to be managed in a special manner. This may lead to a major redesign of the applications. Hence, vertex arrays and VBO's are the most inflexible, but also the most powerful way to accelerate the rendering.

From Chromium's point of view the introduced application based cache control is a good compromise between the effort for implementation and the achieved acceleration of the overall rendering speed. Scenes containing objects with complex geometry, e.g. laser scanned data, will benefit from the described method best.

8. References

[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.

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

[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.

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

[LB06] LORENZ M., BRUNNETT G.: Optimized Visualization for Tiled Displays. Eurographics/ACM SIGGRAPH Symposium Proceedings, Parallel Graphics and Visualization 2006, pp. 127-130, Eurographics Association, 2006.

[LBH07] LORENZ M., BRUNNETT G., HEINZ M.: Driving tiled displays with an extended chromium system based on stream cached multicast communication. To be printed in Special Issue of the Parallel Computing Journal, Vol. 33, ISSN: 0167-8191, 2007.

[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.

© The Eurographics Association 2007.