Understanding Next-generation VR: Classifying Commodity Clusters for Immersive Virtual Reality

Alexander Streit*
QUT

Ruth Christie†
QUT

Andy Boud‡
VR Solutions

Abstract

Commodity clusters offer the ability to deliver higher performance computer graphics at lower prices than traditional graphics supercomputers. Immersive virtual reality systems demand notoriously high computational requirements to deliver adequate real-time graphics, leading to the emergence of commodity clusters for immersive virtual reality. Such clusters deliver the graphics power needed by leveraging the combined power of several computers to meet the demands of real-time interactive immersive computer graphics.

However, the field of commodity cluster-based virtual reality is still in early stages of development and the field is currently ad-hoc in nature and lacks order. There is no accepted means for comparing approaches and implementers are left with instinctual or trial-and-error means for selecting an approach.

This paper provides a classification system that facilitates understanding not only of the nature of different clustering systems but also the interrelations between them. The system is built from a new model for generalized computer graphics applications, which is based on the flow of data through a sequence of operations over the entire context of the application. Prior models and classification systems have been too focused in context and application whereas the system described here provides a unified means for comparison of works within the field.

CR Categories:

Keywords: Virtual and Augmented reality, Real-time Graphics, Computer Clusters

---

*e-mail: a.streit@qut.edu.au
†e-mail: r.christie@qut.edu.au
‡e-mail: andyb@vrsolutions.com.au

1 Introduction

Whereas traditional displays present a window to another world [Sutherland 1965], an immersive display seeks to impart a feeling of being *inside* this other world. This is achieved through interactive "true to scale" 3D graphics encompassing as much of the human visual system as possible. Rendering graphics suitable for the demands of virtual reality applications, using a computer cluster built with commodity hardware, is the topic of this paper.

Stanford University, in conjunction with the U.S. Department of Energy [Houston et. al. 2002], created one of the first commodity-based graphics clusters in 1999. This cluster was composed of 32 nodes and ran the WireGL [Humphreys et. al. 2001] and Chromium [Humphreys et. al. 2002] software suites. Being an early adopter, several issues were encountered and a white paper was published that described hardware limitations with the primary requirement being better throughput on commodity bus systems [Houston et. al. 2002].

Samanta et. al. provide several reasons for using a commodity based graphics cluster [Samanta et. al. 2000]. A summary of reasons identified in [Samanta et. al. 2000] is:

- Lower-cost: a favorable price-to-performance ratio when compared to traditional high-end, custom-designed rendering systems.
- Technology tracking: the performance for commodity components has been improving more rapidly than custom-designed, high-end hardware.
- Modularity & flexibility: networking protocols provide for easier reconfiguration of rendering systems as well as supporting heterogeneous compositions.
- Scalable capacity: Since each PC in a cluster has its own CPU, memory, and AGP bus, scalability is better than a traditional system where the memory and I/O subsystem are shared by all graphics pipelines.

Existing computer graphics systems have been shown to follow a general order [Whitman 1992, Eldridge 2001]. The application supplies a series of vertex data, which is transformed into viewing space data and clipped against the viewing volume (Geometry Processing). The transformed vertices are then rasterized, a process that converts 3D polygons into image space information. Finally, shading parameters are incorporated to produce the resulting pixels (Pixel-level Processing). In the case of an interactive application, these pixels are displayed to the user.

Molnar et. al. describe *the sorting classification system* for parallel computer graphics, which is based on "where the sort from object coordinates to screen coordinates occurs" [Molnar et. al. 1994] (pp. 23). As shown in Figure 1, this can occur at one of three stages, producing three categories: sort-first, sort-middle and sort-last. The "Geometry Processing" operation refers to object space operations prior to rasterization, whereas the "Pixel-level processing" occurs in image space and deals with individual pixels.

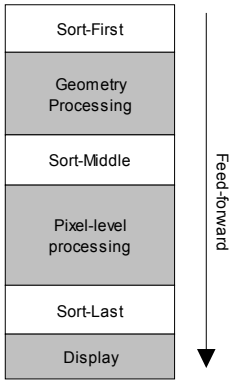

Figure 1: The sorting classification system.

Humphreys et. al. point out, with reference to commodity hardware, that "there are only two points in the graphics pipeline where we can introduce communication: immediately after the application stage, and immediately before the final display stage" [Humphreys et. al. 2001] (pp. 130). Sort-middle systems require access to intermittent data between these stages and are therefore not used in commodity cluster-based systems. The remaining two categories produce only coarse separation between object space and image space sorting methods.

This paper provides finer grained classification than the sorting classification system through development of the *data-stage classification system*. In addition, the categories directly reflect the nature of the data that is transmitted between the nodes of a cluster. This provides an intuitive basis for comparison between categories. To aid the exposition of this classification system, section 2 provides a breakdown of the entire graphics application process into a series of stages, each of which are related to existing models. Section 3 introduces the *data-stage classification system,* a categorization based on the stage at which a virtual reality system enters into parallelism. While section 3 provides a discussion of characteristics per category, it remains academic in its treatment of the system and therefore application of the classification system is deferred until section 4. Section 4 provides a reclassification of major works within the field as confirmation of the practical applicability of the *data-stage classification system*.

2 Data-stage Application Model

In this paper an abstract model of the complete graphics application and rendering process, called the *data-stage application model,* is developed to define the flow of data. This model will be used as a basis for the categorization of cluster based systems.

In the introduction it was established that computer graphics systems follow a general order. This order is represented in the literature using abstract models of the application, such as the OpenGL pipeline model [Eldrige 2001]. Previously these abstractions have been limited to the scope of the particular graphics rendering process. While this is of use to the developers of graphics rendering systems, a higher-level integrated model is needed to provide an understanding of the entire process.

In this paper several prior models are integrated into a unified system, called the *data-stage application model*, illustrated in Figure 2. This model considers the points of communication between interconnected systems libraries. A series of data stages are connected by a succession of operations. Each operation accepts data from the previous stage and produces the data for the next stage. The progression is linear and of a fixed order. The data stages are:

1. Stimulus data is received by the application from external sources such as user input;
2. Stimulus data is processed and produces application specific data;
3. Intermediate data is processed, based on application specific criteria, and produces a series of scene changes;
4. Scene changes are applied to the scene definition in the graphics database through an update operation;
5. Scene definition is processed by a traverse/cull operation that produces graphics rendering commands;
6. Commands are executed, resulting in the final pixel data.

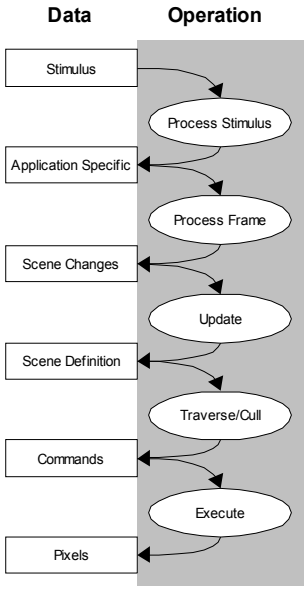

Figure 2: The data-stage application model

Virtual reality applications are a class of interactive graphics applications and therefore conform to the model described.

2.1 The Standard Graphics Pipeline

Traditionally, the graphics application is described using a pipeline model [Eldridge 2001] [Whitman 1992], an example of which is given in Figure 3 (Adapted from [Eldridge 2001, pp. 14]). Abstractly, geometry is transformed into fragments and these fragments are subsequently converted into pixels. In Figure 3, the application is shown to be a source of data, in the form of commands, indicating the starting point in the pipeline. The operations within the shaded area between "application" and "display" are performed by the graphics accelerator libraries.

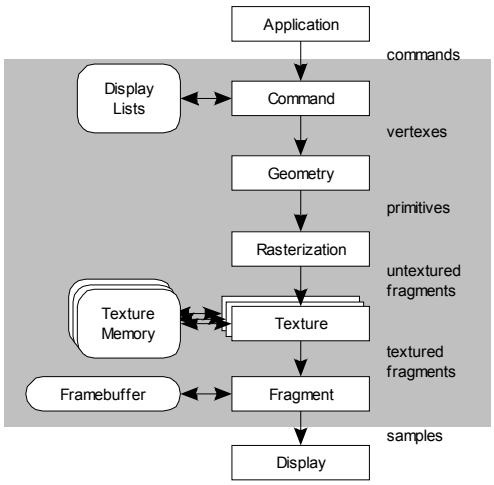

Figure 3: The traditional pipeline model [Eldridge 2001]

A limitation of using commodity hardware architecture is the inability to access intermediate data within the graphics accelerator[Humphreys et. al. 2001]. For this reason the portion of the graphics pipeline that is executed by the graphics accelerator, labeled "execute" in Figure 2, is considered a single unit in the *data-stage application model*. This corresponds with the shaded area in Figure 3, taking series of "commands" as input and producing "Pixels", which correspond to the "samples" data in Figure 3. The pixel data consists of colour, translucency and depth information.

The entire set of commands must be re-transmitted for every frame, since they are discarded once executed. A typical graphics application exhibits a degree of frame-to-frame coherency, that is, the data for a frame changes only incrementally from the previous frame. Optimizations result from exploiting this coherency requiring a graphics database to store the previous frame.

2.2 Graphics Databases

In contrast to the immediate mode interface, a retained mode interface retains the scene definition in a database. This database is processed every frame, resulting in the necessary immediate mode commands. Such a database has knowledge of the entire scene and builds on a variety of techniques including spatial relationships to improve performance. Graphics databases, of which Performer [Rohlf & Helman 1994] is an example, are more commonly referred to as scene-graphs because they describe the scene using a graph data structure.

Performer has been selected as an example of a retained mode interface for the purposes of this paper since it is widely used in the graphics arena. A graphical representation of the Performer system is shown in Figure 4 (reproduced from [SGI 2001]). Performer supports parallelism because it was designed to run on graphics supercomputers that have multiple graphics accelerators. These systems are known as "multi-pipe" systems, referring to each graphics accelerator as a "pipe" or "pipeline". As a consequence, the model in Figure 4 includes parallelism at the pipeline level: the scene is partitioned across multiple pipelines by the scene-graph.

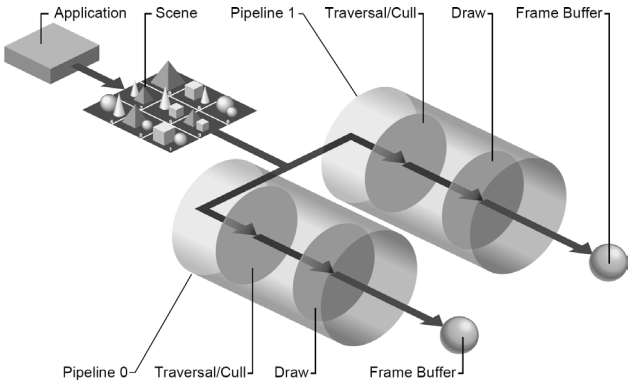

Figure 4: Graphics database model with parallel pipelines [in SGI 2001]

Figure 4 shows the application providing the scene definition, which is processed by each pipeline. Each pipeline applies a series of stages to the data, resulting in frame-buffer data. The purpose of the "Traversal/Cull" stage is to limit the scene data to only the relevant, ordered information enabling efficiency in rendering. The purpose of the "Draw" stage is to produce the actual pixel data.

The *data-stage application model* integrates this system through the "Traversal/Cull" operation of Figure 2. This operation takes a scene definition as input and produces a series of commands, which corresponds to the "Scene" data in Figure 4. Note that the "Draw" operation in Figure 4 represents the same process as the "Execute" operation of the data-stage model in Figure 2.

Although entire applications are developed using an immediate mode interface, such as OpenGL, these applications are not without a graphics database. In this class of applications, the application programmer will maintain their own internal database representation.

2.3 Interactive Graphics and Databases

The "Traversal/Cull" operation in the model considers only the scene definition as supplied by the application, however, interactive graphics applications rely on constantly changing computer imagery. This change is achieved through an "Update" traversal of the scene graph, which is represented by the arrow connecting the "Application" to the "Scene" data in Figure 4. This process applies a set of scene changes to the data, altering the scene data and consequently the resultant images.

Since the update traversal is a data altering event it must be thread safe. Several scene graphs [Reiners 2003, Osfield 2003] are multi-threaded for performance reasons and provide synchronized access to the scene data.

2.4 Application Space

The discussion thus far has considered the underlying libraries used to generate computer graphics, each of which are general in purpose. In practice these are bound to an application written for a

specific purpose, such as a flight-simulator. The part of the program that contains the application specific code is said to exist in *application space*.

The application space code is responsible for the calculation of scene changes, which result from application specific processing and will vary according to the application domain. In the *data-stage application model* these changes are the result of a "process frame" operation, taking "application specific" data as input and producing "scene changes" as output. Application specific data is arbitrarily defined by the application programmer and the modeling of the processing of this data is divided into two subsections:

- scene change calculations, which are represented by the "process frame" operation; and
- stimulus response, which is represented by the "process stimulus" operation.

2.5 Responding to stimulus

The "application specific" data is produced in response to some external stimulus, such as user input events, external data feeds, and timer events. The "stimulus" is not necessarily raw data, but may be the filtered response from an intermediate library such as the CAVELib [VRCO 2003]. The process stimulus operation of the data-stage model includes application specific processing, and produces application specific data.

2.6 Summary

This section presented the *data-stage application model*, which integrates application space, graphics database and traditional graphics application models into a unified process. The model is based on the flow of data, comprising a sequence of data stages and the associated operations that connect them.

3 Data-stage Classification

This section describes a system for classification of clustered graphics and virtual reality systems. This is called the *data-stage classification system* and is based on the *data-stage application model* presented above. Section 4 applies the classification system to several existing works within the field as confirmation of its applicability.

Within a graphics cluster a single node is designated as the master node. The *data-stage classification system* defines the category of the cluster to be determined by the point in the *data-stage application model* at which the master node communicates with the other nodes in the cluster. For example, a cluster where the scene definition is transferred between nodes, is classified as a distributed scene definition cluster. In the as yet undiscovered case where a clustering system communicates at multiple stages, it is classified according to the first point of communication.

Figure 5 illustrates a typical cluster configuration: composed of four nodes, the master node is connected to an input device while the three display nodes are each connected to a separate output device. The *data-stage classification system* is derived by induction. To illustrate this process, each category is presented in the following section with reference to its application in the cluster configuration illustrated in Figure 5.

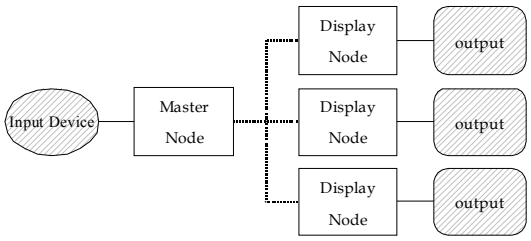

Figure 5: A typical cluster configuration

3.1 Distributed Stimulus

When the cluster in Figure 5 is configured as a *distributed stimulus* system, the master node forwards stimulus data to the display nodes based on readings from the input device. A copy of the application is run at each of the display nodes, typically with differences in the viewing parameters for each. A performance increase can be achieved by pre-processing the stimulus data on the master node, particularly when the number of display nodes increases.

3.2 Distributed Application Specific Data

In the case of *distributed application specific* data, the application developer must construct the application with the intention of its execution on a cluster. Part of the application processing is run on the master node. The remainder of the application execution results from transmission of data to the display nodes. The distribution of this data is based on application specific criteria. The clustering responsibility lies with the application developer and not the underlying systems libraries.

3.3 Distributed Scene Changes

Distributed scene changes uses a "distributed writes" paradigm to provide distributed access to the graphics database. This paradigm is akin to a distributed database system where multiple clients are performing write operations on a database distributed across multiple servers. This configuration is suited to an environment where multiple nodes require write access to the scene definition, which can arise when used in conjunction with *distributed application specific* data.

In the example system given above, the scene definition is stored at each of the render nodes and any changes are sent to each of these nodes. If there are multiple nodes writing to the database, then the render nodes will each perform a conflict resolution process to keep all databases synchronized. The scene definition is not stored on the master node.

3.4 Distributed Scene Definition

Distributed scene definition systems support only a single node with write access to the scene graph. In the example, the master node maintains and updates the scene-graph and runs the application. The scene-graph system is responsible for the render nodes. It ensures that the scene definition is replicated to the render nodes, which apply the read-only operations of culling and drawing.

The distributed scene definition category is similar to a scene-graph system running on a multi-pipe computer with the exception that each pipe is separated from the host processor by a network connection.

3.5 Distributed Commands

Distributed commands involves the transport of the underlying immediate mode commands, such as those of OpenGL. These commands are comprised of:

- state change information, such as lighting and surface properties;
- geometry information, including vertices and polygons;
- synchronization extensions, such as barriers.

In the example system the scene graph executes on the master node, producing a stream of immediate mode commands. These commands are sent to the appropriate render node, where they are executed as they arrive.

3.6 Distributed Pixels

Distributing pixels refers to the transportation of rendered pixels. In the example, the master node executes the entire application and rendering processes. The resultant frame-buffer is then transmitted to each of the render nodes for display. Distributed pixel data systems are used where:

- multiple frame-buffers are combined, either depth composited or tiled; or
- the display device is not attached to the same system that rendered the pixels.

3.7 Category Characteristics

Each data-stage has varying characteristics. Table 1 lists the possible categories and examples of both low and high data requirements. The ideal system for a given project can be imputed by the characteristics of the application.

Category	Example of low data requirement	Example of high data requirement
Distributed Stimulus	Single controller used in a coherent manner	Visualizations of detailed real-time data from an external source
Distributed Application specific	Minimal applications such as walk-throughs	Complex simulations with many distributed processes
Distributed Scene changes	Walk-throughs of static scenes or scenes with few changing elements	Dynamic simulations and visualizations involving continuous large-scale change
Distributed Scene definition	Contained environments with limited detail	Complex, highly detailed and large scenes
Distributed Commands	Objects with planar surfaces, few visible objects	Wide view containing many detailed objects with varying surface properties
Distributed Pixels	Low screen and color resolution	High resolution with auxiliary information (such as depth)

Table 1: Comparison of Categories within the *Data-Stage Classification System*

4 Reclassification of field

This section presents a survey of cluster-based graphics systems classified according to the *data-stage classification system*. The selection of projects is not exhaustive, but it does include sufficient diversity to be representative project types available within the field. Freely available projects such as those supplied under open-source arrangements were given preference, the reasons for which are that they are easily obtained and allow thorough inspection of the code to validate the clustering techniques employed.

4.1 Distributed Stimulus Data

CAVELib

The CAVELib[VRCO 2003] is a set of libraries used as a base for developing virtual reality applications. These libraries manage input devices, inter-process communication, and display parameters. Clustering support for the CAVELib is an extension of a mechanism initially implemented to overcome an earlier limitation of SGI Onyx computers, which had a maximum of three graphics pipelines at the time CAVELib was developed [Pape 1997]. The mechanism, referred to as a "distributed CAVE", was devised to join two such machines via a network for CAVE systems with four to six sides.

A cluster will typically have more than two machines but the underlying mechanics is unchanged. The system uses a distributed stimulus approach where each node of the cluster runs an exact copy of the application.

VR Juggler (ClusterJuggler)

VR Juggler is a suite of APIs that enable platform independent virtual reality application development. Cluster support for VR Juggler is facilitated by the ClusterJuggler API [Allard et. al. 2002]. This component implements a distributed shared memory system with the aim of reducing programming differences between clustered and traditional shared memory systems.

ClusterJuggler implements a remote input device system that allows the raw input data to be processed on the node that is connected to the device. The resulting data is then shared, thereby reducing processing cost by only performing the processing once. ClusterJuggler is classified as a distributed input system.

Syzygy (Master/Slave mode)

Syzygy [Schaeffer & Goudeseune 2003] specifically targets cluster-based virtual reality. The system supports high-performance LANs as well as Internet-based configurations. A significant feature is the ability for nodes to be independently started and reconfigured at run-time, which allows for dynamic cluster configurations. Syzygy supports two modes of operation: master/slave and distributed scene-graph. The application programmer selects the appropriate mode before developing the application. Since this selection is done before the programmer begins to develop the software, Syzygy is best understood as two packages that are distributed together. This section considers the master/slave mode.

In a master/slave setup an exact copy of the application runs at each node. Syzygy is responsible for providing the synchronization mechanism between the nodes. In this mode the system is categorized as distributed stimulus system.

4.2 Distributed Application Specific Data

Domain specific applications written specifically for clusters fit into this category. This paper is concerned with the systems usable by implementers. Consequently, a survey of this class of applications is beyond the scope of this paper.

4.3 Distributed Scene Changes Data

This category is currently empty. Support for *distributed scene changes* is necessary for more flexible use of *distributed application specific* data techniques. Without *distributed scene changes,* the application programmer is required to consolidate any changes at a single node so that they can be written to the scene graph. It is anticipated that systems currently supporting *distributed scene definition* modes of operation will extend to implement distributed scene changes techniques.

4.4 Distributed Scene Definition Data

OpenSceneGraph

The OpenSceneGraph [Osfield 2003] project provides an open-source scene-graph. OpenSceneGraph uses multi-threading to be responsive while providing high rendering performance. As an extension to the thread safety, clustering support is also included natively. In the clustering configuration, remote systems are given synchronized copies of the scene for culling and drawing. The remote systems are treated as pipelines in accordance with Figure 4. OpenSceneGraph provides read-only access to the graphics database and is classified as a distributed scene definition system.

OpenSG

The OpenSG [Reiners 2003] project is an open source scene-graph. Like OpenSceneGraph, OpenSG supports both multi-threading and clustering. OpenSG also allows thread safe write access to the scene graph. This feature, however, does not currently apply to the clustering support. The cluster configuration retains the traditional topology where only the master node has write access and the other nodes act as graphics pipelines. OpenSG is classified as a distributed scene definition data.

Syzygy (Distributed scene-graph mode)

When Syzygy is operating in the distributed scene-graph mode, a single copy of the application is run on the master node. The other nodes in the system act as rendering pipelines, where each rendering a portion of the scene-graph. While a master/slave mode application may render primitives using OpenGL, applications that use the distributed scene-graph must use only the scene-graph API. In its distributed scene-graph mode, Syzygy is classified as a distributed scene definition system.

4.5 Distributed Commands

WireGL

WireGL [Humphreys 2001] intercepts the OpenGL interface and distributes the commands over a network for rendering on other nodes. The node that a command is sent to depends on user-supplied configuration information. Configurations are:

- Sort-first, where each rendering node is responsible for non-overlapping screen regions;
- Sort-last full, where each node renders arbitrary primitives and the entire screen is subsequently depth composited; and
- Sort-last half, which is a hybrid of the other two.

WireGL distributes the immediate mode command stream and is classified as a distributed command system.

The cluster may be configured to have fewer display nodes than rendering nodes, which requires that the resulting pixel tiles are combined for display. This can be facilitated using a hardware based pixel distribution system. WireGL also supports a software approach that reads the frame-buffer and transmits the contents to another node for display. Although this component of WireGL is a member of the pixel distribution category, the user of WireGL does not use this component exclusively of the distributed command system.

AnyGL

AnyGL [Yang et. al. 2002] implements extensions to WireGL for the visualization of large-scale scenes. These extensions include higher levels of data distribution and data compression and have been shown to improve the performance for high-demand systems [Yang et. al. 2002]. As with WireGL, AnyGL distributes the immediate mode command stream and is classified as a distributed command system.

Chromium

Chromium [Humphreys et. al. 2002] presents a stream processing framework for OpenGL command streams. The OpenGL commands are submitted to a series of stream processing units, which each successively transform the command stream arbitrarily. The default implementation of Chromium provides stream processing units that replicate the behavior of WireGL.

Chromium defines the interface between stream processing units to be the OpenGL command interface. When replicating the behavior of the WireGL software-based pixel distribution system, the pixel data is translated into a OpenGL DrawPixels command.

Chromium is categorized as a distributed commands system.

4.6 Distributed Pixel Data

Lighting-2

Lighting-2 is a custom built hardware device that uses the Digital Video Interface (DVI) to scan-out the frame-buffer on a host system [Stoll et. al. 2001]. It supports a tiled configuration that combines several frame-buffers as subsections of a larger display. The advantage of this system is its independence from the host system, meaning it does not consume bandwidth or processing power from the host system.

Lighting-2 is a pixel distribution system.

Metabuffer

The Metabuffer [Blanke 2000] is custom built hardware that similarly uses the DVI to scan out the frame-buffer. The frame-buffers are depth composited in the same manner as WireGL's sort-last configurations. The Metabuffer distributes pixel data, including depth information, and is categorized as a pixel distribution system.

Sepia-2

The first generation of Sepia provided a hardware image

compositing system that required a software process to read the frame-buffer from the card. This process placed a significant requirement on the host processor, which provided the motivation for Sepia-2 to adopt the DVI strategy that both Metabuffer and Lighting-2 implement. Sepia-2 provides higher scalability than Lighting-2 and Metabuffer[Heirich et. al. 2003].

Sepia-2 distributes pixel data including depth information and is classified as a distributed pixel data system.

4.7 Summary

This section reinforces the applicability of the *data-stage classification system* by categorizing several existing works within the field. *Distributed application specific data* was skipped due to its domain specific nature. The *distributed scene changes data* category is notably empty, although it is anticipated that this may change.

Table 2 summarizes the categorization as applied in this section.

Category	Projects in this category
Distributed Stimulus	CAVELib, VR Juggler, Syzygy (Master/Slave mode)
Distributed Application specific	N/A
Distributed Scene changes	
Distributed Scene definition	OpenSceneGraph, OpenSG, Syzygy (Distributed scene-graph mode)
Distributed Command	WireGL, AnyGL, Chromium
Distributed Pixel	Lightning-2, Metabuffer, Sepia-2

Table 2: Categorization of commodity cluster-based graphics systems

5 Conclusion

There is a general design for a graphics application, of which virtual reality systems are specialized class. This design is given by the *data-stage application model* shown in Figure 2, which models the flow of data through a sequence of operations.

The *data-stage classification system* has been developed to categorize graphics clustering systems according to the stage of the *data-stage application model* at which the first communication between nodes occurs. The category directly corresponds to the type of data that is transmitted and the stage the program is in, which provides an intuitive means of understanding and comparing systems.

A survey of the field of commodity cluster-based graphics systems was presented, categorizing projects according to the *data-stage classification system*. This survey demonstrated that while the classification system covers the works within the field, there is currently a vacancy in the *distributed scene changes* category.

6 Future Work

The results listed in Table 1 are based on induction and have not been empirically verified. An empirical study is beyond the scope of this paper since its construction is non-trivial. Results for each clustering implementation and application pair may vary and should be chosen with care. Empirical study of several projects would, however, provide greater insight into the limitations of each of the implementations. A list of suggested measurements is given in Table 3.

Category	Examples of Measurements
Distributed Stimulus	MB/s, FPS, bytes/controller
Distributed Application specific	MB/s, FPS
Distributed Scene changes	MB/s, FPS, changes/MB, MB/master node, frames/MB
Distributed Scene definition	MB/s, FPS, objects/MB, frames/MB
Distributed Command	MB/s, FPS, Commands/s, Commands/MB, Commands/frame, frames/MB
Distributed Pixel	MB/s, FPS, frame/MB

Table 3 Example Measurements for each Category

The categories of the *data-stage classification system* are mutually independent, suggesting that systems from different categories may be run in parallel to one another, forming a *hybrid cluster*. *Hybrid clusters* run two or more categories of software simultaneously. Preliminary work on this topic has been carried out and shows that such a cluster benefits not only from increased performance and scalability, but also in flexibility and functionally. The a priori cluster applied various visual effects through combinations of the distributed command and distributed scene changes systems.

References

ALLARD, J., GOURANTON, V., LECOINTRE, L., MELIN, E., AND RAFFIN, B. 2002. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *Proceedings of Virtual Reality 2002,* IEEE, 273-274.

BLANKE, W. 2000. *The Metabuffer: A Scalable Multiresolution Multidisplay 3-D Graphics System*. Technical Report, University of Texas at Austin.

ELDRIDGE, M. 2001. *Designing Graphics Architectures Around Scalability and Communication*. Ph.D. Dissertation, Stanford University.

HEIRICH, A. 2003. *A Scalable Image Compositing Architecture With Generalized Arithmetic Built From Commodity Components*. Research Report, Compaq Computer Corporation.

HOUSTON, M., HUMPHREYS, G., FRANK, P., AND HANRAHAN, P. 2002. *Life with the Stanford/DOE Graphics Cluster*. Research Report, Stanford University.

HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. 2001. WireGL: a Scalable Graphics System for Clusters. In *Proceedings of SIGGRAPH 2001*, ACM, 129-140.

HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHER, P., AND KLOSOWS, J. 2002. Chromium: a Stream-Processing Framework. In *Proceedings of SIGGRAPH 2002*. ACM, 693-702.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A Sorting Classification of Parallel Rendering. *Computer Graphics and Applications*, 14 (4), 23-32.

OSFIELD, R. 2003. *OpenSceneGraph Documentation*. Technical Reference.

PAPE, D. 1997. *CAVE Library Features*. Invited Talk, Visual Supercomputing Institute.

REINERS, D. 2003. *OpenSG Developer's Guide*. Technical Reference.

ROHLF, J., AND HELMAN, J. 1994. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proceedings of SIGGRAPH 1994*, ACM, 381-394.

SAMANTA, R., FUNKHOUSER, T., LI, K., AND SINGH, J. 2000. *Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs*. Research Report, Princeton University.

SCHAEFFER, B., AND GOUDESEUNE, C. 2003. Syzygy: Native PC Cluster VR. In *Proceedings of Virtual Reality 2003*, IEEE, 15-22.

SGI, INC. 2001. *OpenGL Performer Programmer's Guide.* Technical Reference

STOLL, G., ELDRIDGE, M., PATTERSON, D., WEBB, A., BERMAN, S., LEVEY, R., AND CAYWOO, C. 2001. Lighting-2: a High-Performance Display Subsystem for PC Clusters. In *Proceedings of SIGGRAPH 2001*. ACM, 141-148.

SUTHERLAND, I. 1965. The Ultimate Display. In *Proceedings of the Int. Fed. of Information Processing Congress*, vol 2, 506–508.

VRCO, INC. 2003. *CAVELib User's Manual*. Technical Reference.

WHITMAN, S. 1992. *Multiprocessor Methods for Computer Graphics Rendering.* Jones and Bartlett Publishers.

YANG, J., SHI, J., JIN, Z., AND ZHANG, H. 2002. Design and Implementation of a Large-Scale Hybrid Distributed Graphics System. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*. ACM, 39-49.