Extending the Processing Programming Environment to Tiled Displays

Brandt Michael Westing*, Robert Turknett**

Texas Advanced Computing Center

The University of Texas at Austin

Abstract

MostPixelsEver – Cluster Edition is an extension of the Processing programming environment that enables visualization in cluster-driven display environments without extensive knowledge of programming languages, graphics interfaces, or distributed computing. The work described here enables visual artists, humanities scholars and students, and even traditional programmers to create interactive visualizations in high-resolution distributed environments with simplicity. *MostPixelsEver* hides the inherent complexity of distributed environments by abstraction, and makes it possible to rapidly create visualizations on large displays.

Keywords: tiled displays, programming languages, visual arts.

Index Terms: D.1.3 [Programming Techniques]: Distributed Programming; I.3.8 [Computer Graphics]: Applications

1 Introduction

Processing is a programming environment that was developed in 2001 to promote software literacy in the visual arts[1]. It is a free and open-source programming language and development environment that was originally developed to teach fundamentals of computer programming, but quickly developed into a tool for creating professional work. Processing abstracts complicated programming concepts and simplifies the workflow in creating software by: hiding compilation, linking, and running the program executable; by implementing a scripting layer on Java; and by providing a simplified development environment. Furthermore, Processing provides a programming construct broken into two functional components that abstract the control loop found in most applications.

While Processing works well on its own, it does not natively support *distributed instances*, or multiple dependent processes running on multiple hosts. This presents a barrier for usage of this easy to use language and environment on tiled displays or multi-projector systems. Others have developed frameworks that allow the development of distributed graphics software [1][2], but the work described here provides a much simpler interface to programming with distributed graphics contexts by harnessing the usability of the Processing language. [4] provides similar functionality, but does not provide for easy configuration, relying on the user to create individual configuration files per host, while our work needs only one file. Additionally, [4] does not support broadcast of arbitrary message types between processes, and does not implement non-busy frame locking. This work, by building on [4], simplifies the use of Processing in distributed graphics contexts.

----

*bwesting@tacc.utexas.edu
**turknett@tacc.utexas.edu

2 Implementation

In an effort make the distributed environment transparent to the user, this work requires the user only make a small number of method calls when setting up the scene. Once the calls are made, the user can write the Processing *sketch* how they would in a single process environment. The notion of a messaging *Process* is introduced here that implements the abstraction of the distributed environment for the user. Once a *Process* is instantiated and started, communication with other Processing *sketches* on the network are handled transparently by the messaging *Process* running locally. In the case of a tiled display, there would be a Processing *sketch* instance running on each node, with a complementary messaging *Process* instance running on each node. The messaging *Process* instance handles communication and synchronization via frame locking.

2.1 Implementation: Configuration

To properly set the view frustum of each renderer within the distributed context, a configuration file is created that contains the screen dimensions and location of the process within the larger display. This file serves to also indicate a leader messaging *Process*: the process that will serve frame event messages to follower messaging *Processes*.

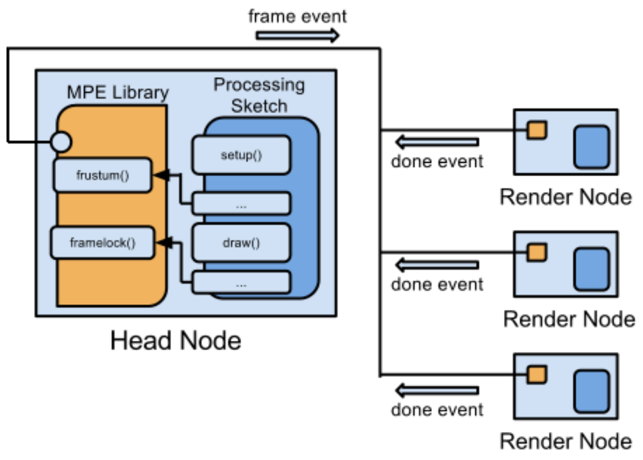

Figure 1: The messaging *Process* (MPE Library) instance is run on each host and transparently handles communication between the leader process and follower processes.

2.2 Implementation: Interaction with Processing

Once a messaging *Process* instance is started on a host by the main Processing instance, it registers itself with the main instance by requiring that the main process call designated method call-back in the messaging *Process* instance before and after the scene is drawn. This call-back registration serves to enable frame locking and frustum correction (Figure 2). Before the scene is drawn, the main instance transparently calls the messaging *Process* object to adjust the view frustum such that the scene appears correct within the larger display surface. This call ensures that the entire scene is not drawn to the local window, but that only the correct portion of the scene is drawn with respect to the larger display area (Figure 2).

After the scene has been drawn, but before the scene is pushed to the graphics pipeline for rasterization, a post-draw method is called on the *Process* and causes the Processing instance to halt until the frame lock has been unlocked. The frame lock is controlled by the Process instance and unlocks the frame lock when a frame event message is sent from the master messaging *Process* (or head node). This sequence of events effectively synchronizes the display surface such that each process must be ready to render the scene before the scene is rendered. While this has the negative side effect that the slowest process controls the speed of rendering, it guarantees that the display is in sync.

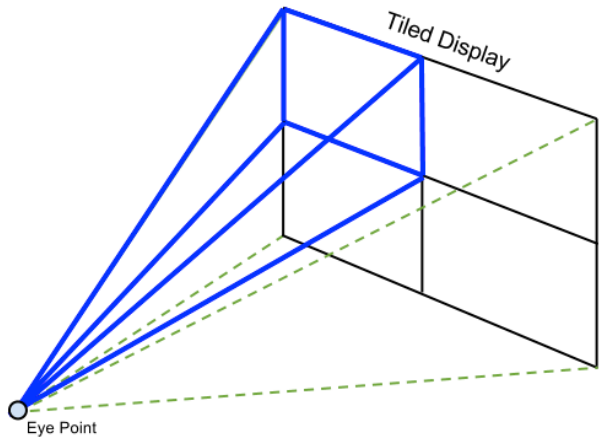

Figure 2: Frustum per process (blue) must be corrected from the aggregated display frustum (green).

2.3 Implementation: Inter-process Communication

The configuration file designates a leader messaging *Process* that will coordinate synchronization between follower *Processes*. When a follower *Process* is instantiated, it connects to the leader through socket communication and establishes itself as a follower. When all followers have registered with the leader *Process*, the leader will broadcast a 'frame event' message that signals that the followers may have their frame lock unlocked. At this time, the leader will also unlock its own frame lock and render the scene. After a follower has rendered the scene, it sends an 'end frame' message to the leader. Once the leader has received all the 'end frame' messages, it again broadcasts a 'frame event' message signalling that the followers may now render. This sequence iterates until interruption by application exit.

The leader *Process* may also send attribute information along with 'frame event' messages. This attribute information may contain information on mouse input at the head node, or any other input sequence that would support interaction. Lastly, an asynchronous messaging *Process* may be started that can contact the leader to provide input. This asynchronous *Process* need not render anything or abide by frame locking, as its sole purpose is to provide input to the *Processes* that make up the distributed graphics context.

3 Conclusions and Future work

The work described provides a transparent interface for writing graphics applications that span multi-node distributed display environments. Extending the Processing Programming Environment to distributed environments creates an effective and easy-to-use graphics environment for tiled displays supporting interaction and facilitating application creation.

This work is still in progress and is an early indication of the reality in easy-to-use interfaces for an otherwise difficult system. The limitations of the system described are that it runs only as fast as the slowest process in the distributed system. This can be alleviated by removing frame locking, but at the cost of synchronization. A more complex protocol, perhaps one that uses buffering, could be explored that might alleviate this problem. Furthermore, this system works because of the inherent determinism in an application. Given an input set, the computation will result in the same output on each run. However, when user input is present at one process and not the other, the input must be broadcasted to all other processes over the network. This is possible with simple input such as mouse and keyboard, but quickly becomes unwieldy with more complex input such as video or sensors.

4 Sample Code

The following snippet of code renders a cube to a distributed display. Commented lines show the *MostPixelsEver* additions:

```
Process process; // MPE Process thread
Configuration tileConfig; // MPE Configuration object

void setup() {
  // create a new configuration object
  tileConfig = new Configuration("configuration.xml", this);

  // set the size of the sketch based on the configuration file
  size(tileConfig.getLWidth(), tileConfig.getLHeight(), OPENGL);

  // create a new process
  process = new Process(tileConfig);

  // start the MPE process
  process.start();
}

void draw() {
  rotateY(-.5);
  background(0);
  fill(255,0,0);
  box(200);
}
```

5 Acknowledgements

This work was made possible by funding from the National Endowment for the Humanities (NEH) Grant: HD-51475-11, A Thousand Words: Advanced Visualization for the Humanities.

References

[1] K. Doerr, F. Kuester, "CGLX: A Scalable, High-performance Visualization Framework for Networked Display Environments," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, 2010.

[2] S. Eilemann, M. Makhinya, R. Pajarola, "Equalizer: A Scalable Parallel Rendering Framework," IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 3, 2009.

[3] *Processing*, Cover, B. Fry, C. Reas, date last referenced: 06/25/2012, http:// http://processing.org/.

[4] *Daniel Shiffman*, Most Pixels Ever, D. Shiffman, date last referenced: 06/25/2012, http://www.shiffman.net/2007/03/02/most-pixels-ever/.