The CompCore
Immersive Display

William Thibault
Professor
Math/CS, CSUEB

Multimedia Forum
May 26, 2010
Overview

• Why immersion?
• What makes it easier now?
• How can you do it?
• What next?
Why immersion?

- The world surrounds us: we're built for it.
- Body is key
  - emotion
  - memory
- Flat media are boring
What's makes it easier now?

- projector prices plummeting
- commodity PC and graphics hardware
- PC clustering software (rocksclusters.org)
- projector-camera systems research (procams.org)
- soon: cameraphones with projectors
How can you do it?

if $, buy it

• planetarium vendors

• Scalable Displays (scaleabledisplay.com)

• Mersive (mersive.com)

else build it
Build it!

- calibrate a camera
- find projector coverage
- 2-pass rendering
- synchronized rendering
- content
Calibrating a lens

- math for lens model: \( r = f(\theta) \)
- calibration object: known positions
- feature detection (corners)
- numerical minimization: find the parameters that explain the observed feature positions
- result: equation to convert to and from:
  - camera pixel locations
  - directions into the scene
## Projector-Camera Correspondences

- **temporal coding**
- **draw circle** $i$ in image $2^j$ if $j$-th bit of $i$ is one
- **draw circle** $i$ in image $2^{j+1}$ if $j$-th bit of $i$ is zero
- **result** = grid of points $(\text{camx}, \text{camy}) \rightarrow (\text{projx}, \text{projy})

<table>
<thead>
<tr>
<th>bit 3</th>
<th>bit 4</th>
<th>bit 5</th>
<th>bit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>![image 0]</td>
<td>![image 1]</td>
<td>![image 2]</td>
<td>![image 3]</td>
</tr>
<tr>
<td>![image 8]</td>
<td>![image 9]</td>
<td>![image 10]</td>
<td>![image 11]</td>
</tr>
</tbody>
</table>
Warping Images for Raskar's 2-pass algorithm

- build "distortion map" by interpolating correspondences
- \( R, G \) = pixel location in unwarped image
- \( B \) = invalid pixel

(pass 1) Render desired image from point-of-view at sweet spot

(pass 2) Warp and project
Synchronize Rendering

- single program, same scene, different camera
- Equalizer framework (equalizergraphics.com)

**Figure 1:** Parallel Rendering

A parallel rendering application uses the same basic execution model and extends it by separating the rendering code from the main event loop. The rendering code is then executed in parallel on different resources, depending on the configuration chosen at runtime. This model is naturally followed by Equalizer, thus making application development as easy as possible.

---

Equalizer is the most advanced middleware for scalable 3D visualization, providing the broadest set of parallel rendering features available in an open source library to any OpenGL application. Many commercial and open source applications in a variety of different markets rely on Equalizer for flexibility and scalability.

Equalizer provides the domain-specific parallel rendering knowhow and abstracts configuration, threading, synchronization, windowing, and event handling. It is a "GLUT on steroids," providing parallel and distributed execution, scalable rendering features, network data distribution, and fully customizable event handling.

If you have any questions regarding Equalizer programming, this guide, or other specific problems you encountered, please direct them to the eq-dev mailing list.

---

1. Introduction

Equalizer is the standard middleware for the development and deployment of parallel OpenGL applications. It enables applications to benefit from multiple graphics cards, processors, and computers to scale rendering performance, visual quality, and display size. An Equalizer-based application runs unmodified on any visualization system, from a simple workstation to large scale graphics clusters, multiGPU workstations, and Virtual Reality installations.

This User and Programming Guide introduces parallel rendering concepts, the configuration of Equalizer-based applications, and programming using the Equalizer parallel rendering framework.
CompCore Immersive Display Architecture
Gigapan Epic (gigapan.org)
surround video

Point Gray Ladybug2
(ptgray.com)

dodecahedral camera
(immersivemedia.com)
surround CGI

- cubemap rendering
- 6 cameras, 90-deg FOV
- standard cameras
- non-linear projections
- linear fisheye
- latitude-longitude
- special "cameras"
- typical distribution format: fisheye
- real-time rendering
- 2-pass algorithm
- assume viewer at "sweet spot"
- find portions of field-of-view lit by each projector
- for rendering, use a different camera for each projector
  - place each camera at the sweet spot
  - give each camera a different direction and fov
  - render scene to offscreen buffer
  - warp rendering to account for distortion

(pass 1) Render desired image from point-of-view at sweet spot

(pass 2) Warp and project

our approach