Parallel Rendering using OpenGL Multipipe SDK (MPK)

Praveen Bhaniramka
SGI
Talk Outline

- Overview
- Application Structure
- Configuration Interface
- Parallel Rendering
- Related Projects/Approaches
- More Information
Overview

Design Objectives/Features

- Provide an API for development of OpenGL-based Multipipe applications
- ‘Runtime portability’ from desktop to multi-pipe systems
- ‘Runtime scalability’
- Minimize invasiveness

Sort of like a multipipe GLUT
Runtime Portability

- From Single-pipe systems to Multi-pipe systems

Application is independent of system configuration
Runtime Scalability

Application scales as hardware resources are added
Minimal Invasiveness

- Callback driven
- Basic OpenGL Framework
- Relatively simple GLUT-like C API
- Channel & Stereo Independent
Application Structure

- OpenGL Rendering (Application code)
- Multipipe Management (MPK)
- System Configuration (MPK Config file)

Application Callbacks

Load at runtime
System Configuration

- **MPKConfig**
  - Hierarchical Description of System Configuration
  - Specifies the relationships between different components

- **MPKChannel**
  - Basic, display device independent OpenGL rendering unit.
  - These are “framebuffer resources”

- **MPKConfig Hierarchy**
  - A Config has one or more Pipes
  - A Pipe has one or more Windows*
  - A Window has one or more Channels
    - * Each window has a dedicated rendering thread
Application Flow

1. Start application
2. Load Configuration from file
3. Set MPK Callbacks, Initialize shared data
4. Call `mpkConfigInit`()
5. Call `mpkConfigFrame`()
6. Exit Application?
7. If No, Update frame data
8. If Yes, Invoke Exit callbacks, Free shared data
9. Call `mpkConfigExit`()
10. Stop
Render execution flow is controlled by ...

mpkConfigFrame()

... which leads to execution of ...

the draw scene callback (per channel)

• Executes one frame of rendering
• Window threads invoke update callbacks
• Passed framedata is distributed latency-correct
Application

```c
#include <mpk/mpk.h>

// main()
mpkInit();
MPKConfig* cfg = mpkConfigLoad(configFileName);

// read config
mpkConfigSetWindowInitCB(cfg, initWindowCB);

// initialize window callbacks
mpkConfigInit(cfg);

// start: spawn one thread per window

// rendering loop
while ( notDone ) {
    // do application work (spin the cow, make sharks
    // swim, etc)
    updateSharedData( &framedata );
    // sync window loop threads, etc.
    mpkConfigFrame( cfg, framedata );
}

mpkConfigExit(cfg);
```
Application

// updateChannel() user callback: invoked by
// mpkConfigFrame() => glutDisplayFunc
// clear render area
mpkChannelApplyBuffer( c );
mpkChannelApplyViewport( c );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
// apply projection matrix
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
mpkChannelApplyFrustum( c );
// apply modelview matrix
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
mpkChannelApplyTransformation( c );
// cull and/or draw cow, sharks, etc.
drawSharedData( c, framedata );
Rendering Flow

Pipe => Window => Thread => GL Context => Channel

- Application Thread
  - for each frame
    - mpkConfigFrame
  - Window Thread
    - for each window
      - do per window work
    - for each channel
      - do channel rendering
Config File

- Simple ASCII file representation of MPKConfig data structure
- Hierarchical Description for Framebuffer Resources
- Channel Physical Layout
- Channel Decomposition
Simple Config

1 pipe, 1 window

```
config {
    name "1-window"
    pipe {
        window {
            name "MPU:simple"
            viewport [ 0.25 0.25 0.5 0.5]
            channel {
                name "channel"
                wall {
                    bottom_left  [0 0 0]
                    bottom_right [1 0 1]
                    top_left     [1 1 1]
                }
            }
        }
    }
}
```
Simple Config

1 pipe, 2 windows

```plaintext
cfg {  
    name "2-windows"
    pipe {  
        window {  
            name "left"
            viewport [ 0.25 0.25 0.25 0.5]
            channel {  
                name "channel"
                wall {  
                    bottom_left [0 0 0]
                    bottom_right [1 0 1]
                    top_left [1 1 1]
                }
            }
        }
        window {  
            name "right"
            viewport [ 0.5 0.25 0.25 0.5]
            channel {  
            }
        }
    }
}
```
Compounds

- Compounds provide an abstraction for parallel rendering
  - A Config can have one or more compounds
  - Compounds can be hierarchical with a tree-like structure
  - Compounds reference channels as sources and/or destinations
  - SW as well as HW compositing
  - Scaling may require some application awareness
Compounds

Pipe 0
  Draw
  Source channel 0
    Read
    Compound
    Destination channel

Pipe 1
  Draw
  Source channel 1
    Read
    Compound
    Destination channel

Pipe 2
  Draw
  Source channel 2
    Read
    Compound
    Destination channel

Pipe m
  Draw
  Source channel 3
    Read
    Compound
    Destination channel
Compounds

config {

  # one or more pipes with windows and channels
  pipe {
    ... ...
  }
  ...

  # compound for the above config
  compound {
    # specify the compound type, format and output channel
    mode [ 2D/DB/DPLEX, HW/NOCOPY, etc ]
    format [ COLOR, DEPTH, etc]
    channel "channel_1"
    # specify one or more source channels and their params
    region {
      ... ...
    } ...
  }
}

Compounds

- Commonly used Modes
  - 2D (screen tiling)
  - DB (database decomposition)
  - DPLEX (time-slice multiplexing)
  - EYE (stereo decomposition)
  - Others...
2D Compound

Each pipe renders a different viewport
compound {
    mode [ 2D ]
    format [ COLOR ]
    channel "channel_1"
    region {
        viewport [ 0., .0, 1.0, 0.5 ]
        channel "channel_1"
    }
    region {
        viewport [ 0., .5, 1.0, 0.5 ]
        channel "channel_2"
    }
}

2D Compound
2D Compound

- Automatic Load balancing
  - Based on timing values from last frame
  - Good results for low-latency decompositions
DB Compound

Each pipe renders a different part of the data set

N times pixel fill rate
N times texture memory
N times texture download rate
DB Compound

Compositing order changes with view
Adaptive Readback comes in handy
compound {
    mode [ DB ]
    format [ COLOR DEPTH ]
    channel "channel_1"
    region {
        range [ 0., 0.5 ]
        channel "channel_1"
    }
    region {
        range [ 0.5, 1 ]
        channel "channel_2"
    }
}
DPLEX Compound

Each pipe renders a different frame
DPLEX Compound

```
compound {
    mode         [ DPLEX ]
    format       [ COLOR ]
    channel      "channel_1"
    region {
        channel "channel_1"
    }
    region {
        channel "channel_2"
    }
}
```
EYE Compound

Each pipe renders for a different eye position
## Compound Configs

- Choosing the right decomposition mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Geometry Processing</th>
<th>Pixel Fill</th>
<th>Bandwidth to graphics</th>
<th>Graphics Memory</th>
<th>Application Transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>Y/N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y/N</td>
</tr>
<tr>
<td>DPLEX</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DB</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
Hierarchical Compounds

```plaintext
compound {
    mode [ DPLEX ASYNC ]
    channel "channel"
    region {
        compound {
            mode [ DPLEX ASYNC ]
            channel "dplex::1"
            region { channel "dplex:a:1" }
            ...
        }
        ...
    }
    region {
        compound {
            mode [ DPLEX ASYNC ]
            channel "dplex::2"
            region { channel "dplex:b:1" }
            ...
        }
        ...
    }
    region {
        9 => 3 => 1 windows
        ...
    }
}
```
compound {
    mode [ 2D HW NOCOPY ]
    channel "channel0"
    region {
        viewport [ 0., .75, 1., .25 ]
        channel "channel0"
    }
    ... ...
    region {
        viewport [ 0., .25, 1., .25 ]
        channel "channel2"
    }
}
Hybrid Compounds

- Combined DPLEX - 2D

Pipe 0 + Pipe 1 + Pipe 2 + Pipe 3
Related Projects

- Chromium
  http://chromium.sourceforge.net
- OpenGL Multipipe (OMP)
  http://www.sgi.com/software/multipipe
- CAVELib
- VRJuggler
Scalability Approaches

• **Aware Applications**
  – Effort required to port the app to run in MP environment (MPK, PF)
  – Good scalability with app work
  – Immersive environments easier to handle

• **Unaware Applications**
  – Effort goes into the intercept-dispatch library (CR, OMP)
  – Good/Limited scalability depending on the app
  – Immersive environments not so easy
**MPK vs OMP**

- **Multipipe SDK**
  - API for writing MP apps
  - App scales fill, geometry, memory and display

- **OpenGL Multipipe**
  - Transparent app layer
  - Scales display size, fill, well. Limited geometry, texture, scaling.
Recap: Features

- **Ease of Integration**
  - fork, sproc, pthread support
  - Event-driven execution model
  - Adaptive readback interface
  - App-created windows support
  - Non-threaded windows support
  - Xinerama integration
  - Custom compositing interface

- **Runtime Portability**
  - ASCII File Format specification
  - Multi-frustum support
  - Dynamic parallel rendering

- **Runtime Scalability**
  - Compound class specification
  - 2D, DB, EYE, DPLEX and FSAA compounds
  - RGBA, Z and STENCIL image compositing
  - Latency / ASYNC decomposition
  - Automatic load-balancing
  - SGI Scalable Graphics Hardware integration

- **Stereo / Immersion**
  - Off-axis frustum computations
  - Stereo / Head-Tracking support
  - Head Mounted Display [ HMD ]
  - Mirrored Projection Support
MPK: More Information

- MPK 3.0.1 web release available
- Multipipe SDK product web site
  http://www.sgi.com/software/multipipe/sdk
- Engineering mailing list
  mpsdk@els.sgi.com
### Multipipe API Comparison

<table>
<thead>
<tr>
<th></th>
<th>Chromium</th>
<th>OMP</th>
<th>MPK</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application-transparent</strong></td>
<td>Yes/No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Open source</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Supported OSes</strong></td>
<td>Windows, Linux, IRIX, etc.</td>
<td>Linux (IA64), IRIX</td>
<td>Linux (IA64), IRIX</td>
</tr>
<tr>
<td><strong>Programming model</strong></td>
<td>application-transparent OpenGL + optional Chromium extension(s)</td>
<td>application-transparent OpenGL callbacks for frame and data management</td>
<td></td>
</tr>
<tr>
<td><strong>Runtime configuration method</strong></td>
<td>python launch scripts, interchangeable modular Stream Processing Units (SPUs)</td>
<td>command line flags, environment variables</td>
<td>MPK config file</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>multiprocess</td>
<td>multiprocess</td>
<td>multithreaded/fork</td>
</tr>
<tr>
<td><strong>Node structure</strong></td>
<td>arbitrary directed graph of SPUs (many masters, many slaves)</td>
<td>one master (app), one or more cullers, many slaves</td>
<td>one master, many slaves (optional culler per slave)</td>
</tr>
<tr>
<td><strong>Codec</strong></td>
<td>WireGL-like</td>
<td>GLSCodec</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Transport among processes</strong></td>
<td>tcp, miranet, MPI, IB?</td>
<td>Shared memory (queue) - data must be copied into shm</td>
<td>Shared memory (arena) - can pass pointers to data residing in shm</td>
</tr>
<tr>
<td><strong>Decomposition modes</strong></td>
<td>Sort-first (Tilesort)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Sort-last (Depth/Alpha)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Sort-last w/ HW readback</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Timeslice</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Stereo</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Culling (tilesort)</strong></td>
<td>BBox computation</td>
<td>Master node</td>
<td>Master node</td>
</tr>
<tr>
<td></td>
<td>BBox transformation</td>
<td>Master</td>
<td>Culler</td>
</tr>
<tr>
<td></td>
<td>GL state management</td>
<td>Master</td>
<td>Slave</td>
</tr>
<tr>
<td><strong>Number of transport channels</strong></td>
<td>one per edge in the graph</td>
<td>one per rendering application thread</td>
<td>one per MPK application</td>
</tr>
<tr>
<td><strong>Readback/glGet</strong></td>
<td>Not fully supported?</td>
<td>Master queries slaves or local pipe</td>
<td>Native GL</td>
</tr>
</tbody>
</table>
MPK: Configuration