Visualizing the Mandelbrot set in real-time using compute shaders and a multi-display system

Dr. Michael S. A. Robb

July 2015

Table of Contents

Table of Contents

Index of Tables

Illustration Index

Introduction

This application is a command line Mandelbrot fractal viewer that allows the user to view the two dimensional space interactively in real-time. Going beyond the traditional Mandelbrot viewer, it also allows the user to use the TCP/IP and sockets to connect together different systems so that the display capabilities of each system can be combined together to form a larger “virtual” screen.

Inspired by the early fractal viewers (fractx86 and fractint) and more recent versions such as the CUDA GPU example, and CPU based versions such as XAOS, I wanted to write a fractal viewer that could take advantage of both networks, multiple screens (either desktop monitors or mobile devices) to provide high-resolution visualizations of the Mandelbrot set with a variety of rendering methods including color cycling and smooth rendering.

Command line options

Starting the application

The application can be started as a window simply by simply typing:

```
> compute
```

Alternatively to start as fullscreen without a border:

```
> compute -b -u
```

This will put the application into full-screen mode. You should then see a screen as follows:

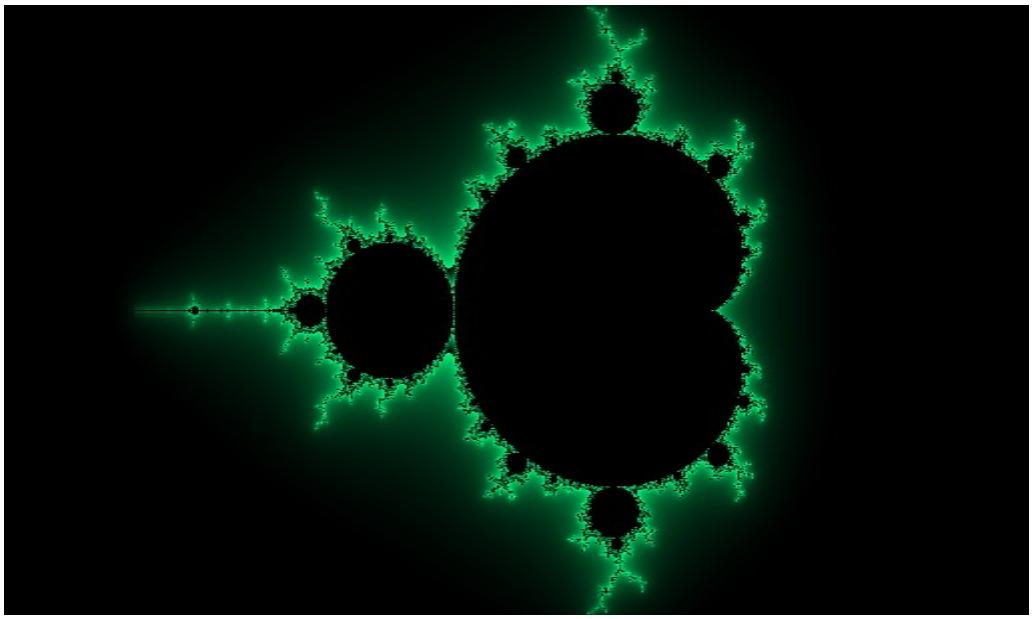


Illustration 1: Startup view

At any time you can press [F12] to exit, or right [CTRL] and [Space] together to reset back to this view. Various actions are also possible using the mouse. These include zooming in, zooming out, and rotating clockwise or anticlockwise.

Press [s] to enable smoothing mode. You should see a frame like:

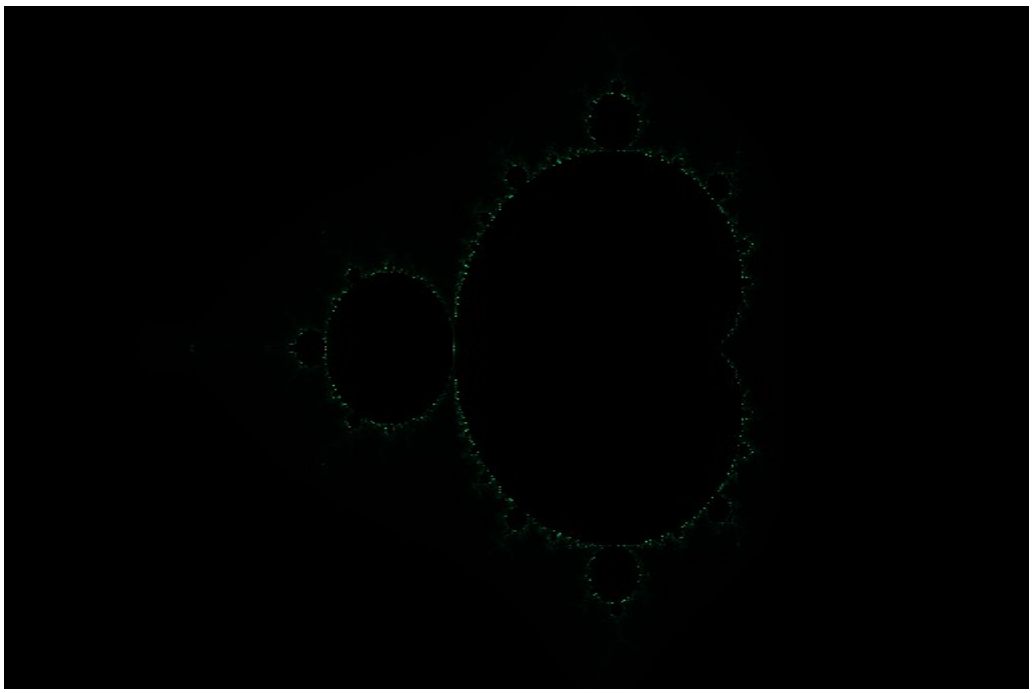


Illustration 2: Smoothing mode enabled

This is the basic image on startup. Start by zooming and rotating with the [c] [v], [z], [x] and the mouse (using the jogwheel as well).

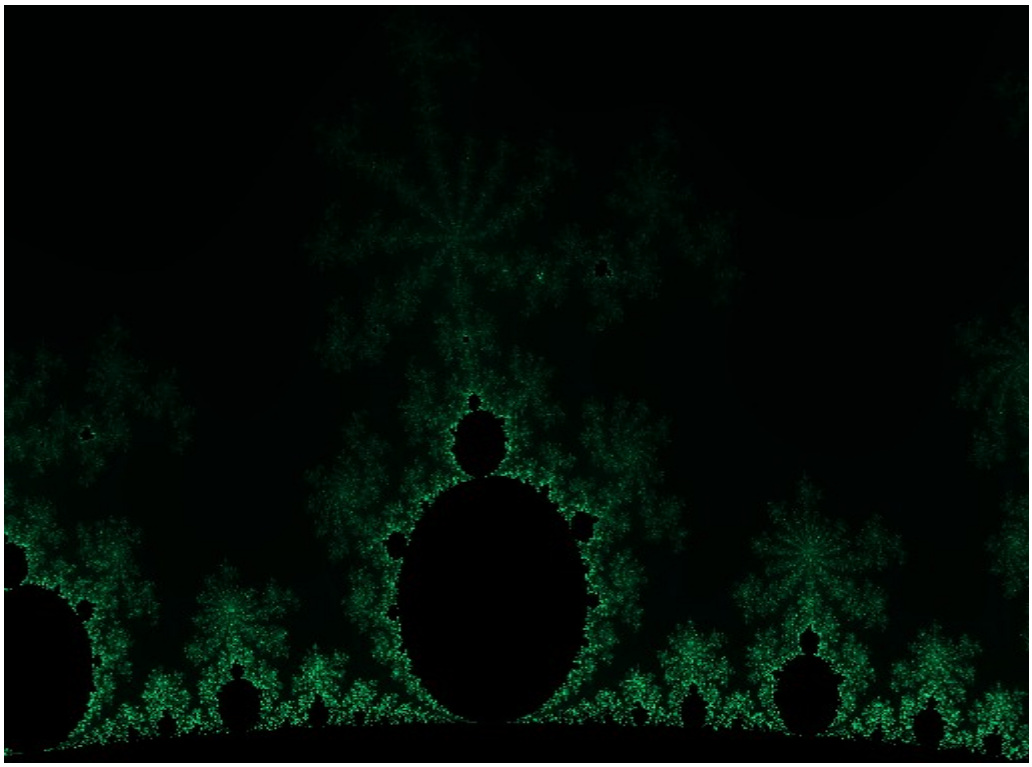


Illustration 3: Zoomed in view

You can press the [I] key to get a text version of the parameters:

```
{fractal
        {escaperadius 4 }
        {xcen -0.690092 }
        {ycen -0.35243 }
        {xwidth 0.140292 }
        {ywidth 0.0789141 }
        {maxiter 1024 }
        {angle -1.04587 }
        {power 2 }
        {banding 1 }
        {modesmooth 0 }
        {modepower 2 }
        {texpalette 0 }
        {makemipmaps 0 }
        {minclamp 0 }
        {texwidth 512 }
        {texheight 512 }
        {cycleoffset 0 }
        {modeinvert 0 }
        {autocycle 0 }
}
```

Text 1: Parameters for "Mint Fractal"

You can save this text to a file and load it from the command line using the [-f] option

Use the [<] and [>] keys to decrease and increase the texture resolution. Press the [f] key to enable mip-mapping. This will soften the image:

Illustration 4: Mint Fractal

The color palette in use is selected through the left [CTRL] key and the number keys [0] to [9]. These are:

Key combination	Palette
[CTRL] [1]	Mint Green
[CTRL] [2]	Electric blue
[CTRL] [3]	Orange
[CTRL] [4]	Lime Green
[CTRL] [5]	Electric Pink
[CTRL] [6]	UV Purple
[CTRL] [7]	Yellow-Purple
[CTRL] [8]	Basic rainbow
[CTRL] [9]	Black/White
[CTRL] [0]	Stripes, use the left bracket key [to remove clamping

Table 1: Default color palettes

Stripy fractals can be generated by pressing [CTRL][0], ensuring that mip-mapping is in use with [F] key and then increasing the texture size using the [>] key. There's a trade-off between rendering speed, texture mapping and the use of mip-mapping.

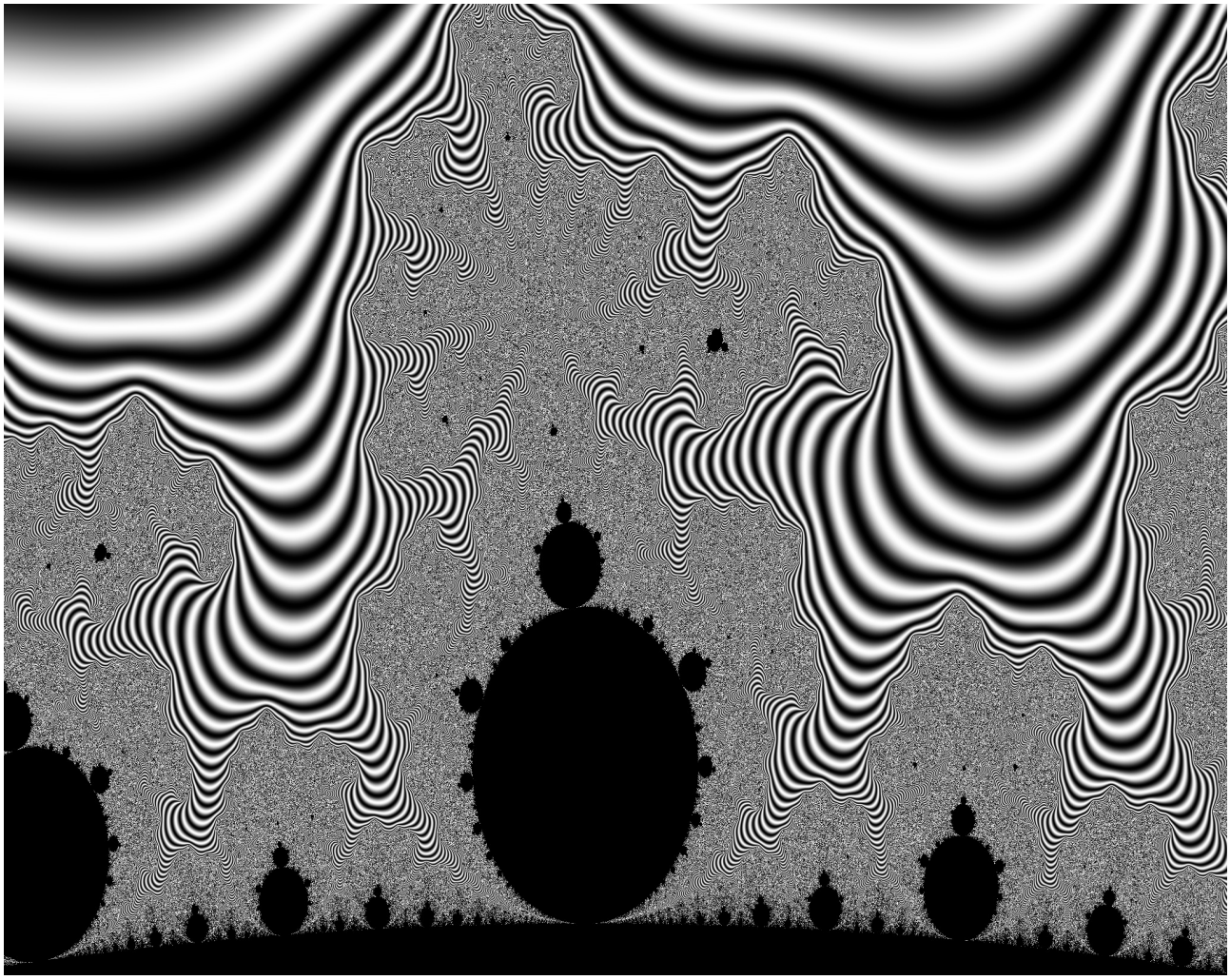


Illustration 5: "Stripes"

```
{fractal
     {escaperadius 4 }
     {xcen -0.690092 }
     {ycen -0.35243 }
     {xwidth 0.140292 }
     {ywidth 0.0789143 }
     {maxiter 1024 }
     {angle -1.04587 }
     {power 2 }
     {banding 1 }
     {modesmooth 1 }
     {modepower 2 }
     {texpalette 9 }
     {makemipmaps 1 }
     {minclamp 0 }
     {texwidth 4096 }
     {texheight 4096 }
     {cycleoffset 0 }
     {modeinvert 0 }
     {autocycle 0 }
}
```

Text 2: Parameters for "Stripes"

It is possible to change the palette simply by editing this file or from the application:

```
{fractal
     {escaperadius 4 }
     {xcen -0.690092 }
     {ycen -0.35243 }
     {xwidth 0.140292 }
     {ywidth 0.0789143 }
     {maxiter 1024 }
     {angle -1.04587 }
     {power 2 }
     {banding 1 }
     {modesmooth 0 }
     {modepower 2 }
     {texpalette 1 }
     {makemipmaps 1 }
     {minclamp 0 }
     {texwidth 4096 }
     {texheight 4096 }
     {cycleoffset 0 }
     {modeinvert 0 }
     {autocycle 0 }
}
```

Text 3: Parameters for "Electric Blues"

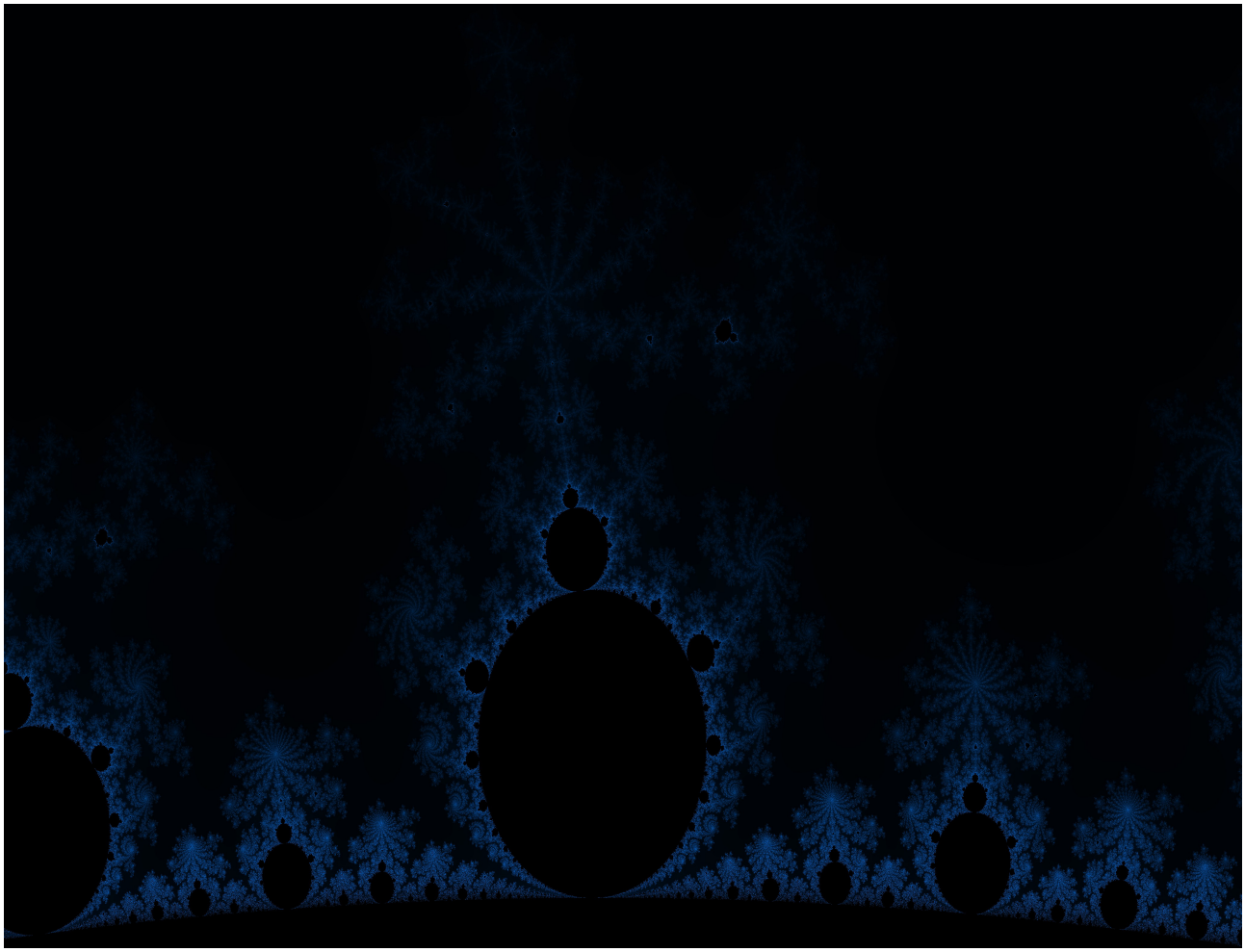


Illustration 6: "Electric Blue" fractal

Color cycling

To enable color cycling press either the [-] or [+] keys on the keypad. This will affect all palettes. Press the keys again to speed up or slow down the color cycle. Press the keypad [enter] key to stop the color cycle. The direction of color cycling can be reversed through the use of the invert palette buttons (open and close square brackets).

Creating screen shots

To take screenshots or “texture shots”, press either the [F1], [F2], [F3] or [F4] keys. These will generate one or more subframes of the current image which can be joined together to form a larger image.

Activating server mode

To run the application as a server, use the [-s] option to enable the server, and [-p] to select a TCP/IP port to receive commands remotely. You will need to set up a configuration file on the client to define the relationship of the remote screen to the local screen. These can either be identical or offset to each other. It is also possible to have two applications run both as server and client simultaneously, so that they can send updates to each other.

To allow the testing of such configurations on a single Linux system, different configuration files and ports can be used on the same system:

```
./compute –s –c –nconfigs/nodesimple2.txt –p8080 $1 &
./compute –s –c –nconfigs/nodesimple.txt –p8081 $2 &
```

Text 4: Twinned windows

This allows two windows to operate side by side and share user input events such as movement, rotation, palette selection and animation.

A window can choose whether or not to accept new parameters from across the network by pressing the [F5] key. An update can still be performed by pressing the [F9] key.

For more details on having multi-windowed displays, see the chapter titled “Display configuration files“.

Summary of keyboard commands

Keyboard movement commands

When client mode is active, the following options are available to control movement of the fractal view:

'C'	Zoom in (slowly)
[Shift] + 'C'	Zoom in (fast)
'V'	Zoom out (slowly)
[Shift] + 'V'	Zoom in (fast)
'Z'	Rotate anticlockwise (slowly)
[Shift] + 'Z'	Rotate anticlockwise (fast)
'X'	Rotate clockwise (slowly)
[Shift] + 'X'	Rotate clockwise (fast)
[CtrlRight] + [Space]	Panic button to reset parameters
[F8]	Request Parameters from remote hosts
[F12]	Panic button to exit client
[Left-Arrow]	Move center-point left (slow)
[Shift-Left] + [Left-Arrow]	Move center-point left (fast)
[Right-Arrow]	Move center-point right (slow)
[Shift-Left] + [Right-Arrow]	Move center-point right (fast)
[Up-Arrow]	Move center-point up (slow)
[Shift-Left] + [Up-Arrow]	Move center-point up (fast)
[Down-Arrow]	Move center-point down (slow)
[Shift-Left] + [Down-Arrow]	Move center-point down (fast)

Table 2: Keyboard commands for movement

Keyboard visual appearance commands

When client mode is in use, the following keyboard commands are available to control the visual appearance of the fractal:

Key	Purpose
'F'	Toggle mip-mapping
'S'	Toggle smoothing mode
'B'	Reduce banding
'N'	Increase banding
'<'	Reduce texture size by a half
'>'	Increase texture size by a half
'0' .. '9'	Set number of iterations as power of 2 16, 32, 64, 128, 256, 512, 1024, 2048 ...
[Ctrl][Left] + '0 .. '9'	Select transfer texture 1 = rainbow 2 = monochrome 3 = purple/gold 4 = red/yellow 5 = black/blue 6 = stripy
'['	Disable clamping for smooth mode
']'	Enable clamping for smooth mode
'J'	Set power mode to 2
'K'	Set power mode to 3
'L'	Set power mode to N
[F1]	Take snapshot from texture (same size as texture)
[F2]	Take snapshot from texture (4 subframes) (total size = 2x texture size in each axis)
[F3]	Take snapshot from texture (16 subframes) (total size = 4x texture size in each axis)
[F4]	Take snapshot from texture (64 subframes) (total size = 16 x texture size in each axis)

Table 3: Keyboard commands to affect visual appearance

Keyboard commands to load and save files

When client mode is in use, the following keyboard options are available to load, save and display parameters:

'W'	Write current parameters to “default.frac”
'R'	Read current parameters from “default.frac”
'I'	Display information on current settings (Text can be saved and loaded in using the -f command line option)

Table 4: Keyboard commands to load, save and display parameters

Keyboard commands to edit keyframes

When client mode is active, the following keyframe editing options are available:

'D' + [CtrlLeft] + [CtrlRight]	Delete keyframe list
'M'	Save keyframe list
'A'	Add entry to keyframe list
'E'	Delete last saved keyframe location
'P'	Replay animation from the keyframe list
[ESC]	End replay of saved keyframe animation

Table 5: Keyboard commands to edit animation keyframes

Keyboard commands to manipulate color cycling

When client mode is active, the following color cycling controls are active:

Keypad [-]	Decrease rate of color cycling
Keypad [+]	Increase rate of color cycling
Keypad [ENTER]	Stop color cycling (stop color cycling)
[\]	Reverse texture lookup

Table 6: Keyboard commands to control color cycling

Keyboard commands to control network settings

When client mode is active, the following function keys control transfer of parameters:

[F5]	Toggle updates from remote client
[F8]	Send request for parameters from peers
[F9]	Send a ping message
[F10]	Send a request window size message

Table 7: Keyboard commands to control network transfer of parameters

Summary of mouse commands

Client mode mouse commands

When client mode is active, the following mouse commands are available:

Mouse command	Purpose
Jogwheel up + no buttons down	Zoom in
Jogwheel down + no buttons down	Zoom out
Mouse movement + Left button down	Move center-point
Jogwheel up + Right button down	Rotate clockwise
Jogwheel down + Left button down	Rotate anticlockwise

Table 8: Mouse control commands

Summary of command line options

As the application is intended to be run from the Linux command line, various command line options have been made available to allow configuration from just a few commands:

Command line option	Purpose
-v	Enable verbose mode
-f<filename>	Load in a single set of fractal parameters from an ASCII file
-r<replayfile>	Load in a replay file
-s	Enable server mode
-c	Enable client mode
-n<nodefile>	Specify the set of additional display nodes
-p<port number>	Set the port number
-t<dimensions>	Set texture dimensions in 2D (eg -t1024x1024)
-w<dimensions>	Set window dimensions in 2D (eg. -w1024x1024)
-z<imagefile>	Set transfer texture for colouring
-b	Set borderless window
-u	Set fullscreen window (activate borderless window)
-x<coordinates>	Sets top left corner of window
-m	Dump message packets

Table 9: Summary of command line options

With this application, a “server node” is a version of the application that only receives commands across the network, but does not accept any local user input. A “client node” is a node which accepts user input local and sends it across to additional “server nodes”. A “node” is defined as a combination of IP address or hostname and port address. eg. A combination of 127.0.0.1 and port of 8080 would be an address on the local system.

Server mode and client mode are mutually exclusive. Only one or the other can be set. However, both client and server nodes can have their own set of peer nodes that it propagates display parameters to. The set of peer nodes is specified through the “-n” command line option.

Display configuration files

The specification of the display configuration that a client or server node sends updated display parameters to is specified using a tagged ASCII file format. This is described as follows:

The file is prefixed with the header block:

```
{nodelist
```

Each individual node is specified with the node header block:

```
    {nodedata
```

The individual parameters of that particular host are specified as follows:

```
      {hostname "bigscreen"}
      {port 8080}
      {exid 0}
      {offx -1.0 }
      {offy  0.0 }
      {width 1.0}
      {height 1.0}
      {offheading 0.0}
```

Each individual node is also terminated with a closing character:

```
    }
```

There is no limit to the number of nodes that may be specified in this file.

The file is terminated with a closing character:

```
}
```

Each of the parameters has the following purpose:

{hostname “<ip address>”} {hostname “<hostname”}	The hostname of the node eg. {hostname “127.0.0.1”} or {hostname “deepthought”}
{port <integer> }	Network port number eg. {port 8080}
{exid <integer> }	Window number (reserved for future use, so always zero) {exid 0}
{offx <float> }	Offset in X-axis in units of current node width
{offy <float> }	Offset in Y-axis in units of current node height
{width <float> }	Width of node display in units of current node
{height <float>}	Height of node display in units of current node
{offheading <float>}	Offset to angle of rotation. This allows for the use of monitors that have been rotated at different angles.

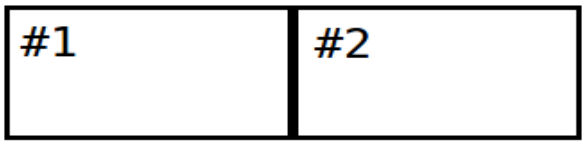
Table 10: Virtual screen node list specification format

Through the use of this specification file, it's possible to allow one system to remotely control a multitude of displays to form one “virtual screen”.

To enable testing and configuration on a single desktop PC, it's possible to have one or more versions of the application listening on different ports, while a single interactive client calculates all the relevant parameters. The next page describes how a quad display can be set up on a single PC:

Example 1: Configuring a simple two unit display

For this example, the goal is to set up a 2x1 display in the following shape:



#1 is the server, while #2 is the active interactive client

To connect up the display servers and client, the node specification file is as follows:

```
{nodelist
        {nodedata
                {hostname "127.0.0.1"}
                {port 8080}
                {exid 0}
                {offx -1.0 }
                {offy  0.0 }
                {width 1.0}
                {height 1.0}
                {offheading 0.0}
                {aspect 1.0 }
        }
}
```

Text 5: Simple paired window

For this example, the file is called “nodesimple.txt”.

This allows each node to have an area of the client screen that is half the width but full height of the client node. Because these displays are arranged 2x1, the offset to the centre-point of each server node in X is a quarter of the width of the client node, but the height remains the same The assignments of node parameters are then as follows:

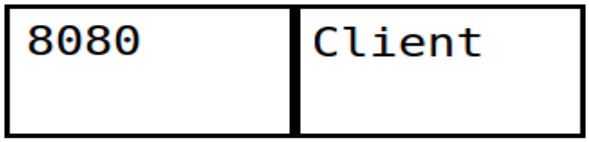


Illustration 7: Paired windows

To get the server to run on the local PC, we run the following:

```
#!/bin/bash
./compute -s -p8080 &
sleep 1
./compute -c -nnodesimple.txt
```

Text 6: Script to set up paired windows

The first two commands set up four servers each listening on a separate port, then the sleep command forces the script to wait while the windows are set up by the system. The next step is to get the interactive client running using the specification file.

Example 2: Configuring a virtual quad display

For this example, the goal is to set up a 2x2 display in the following shape:

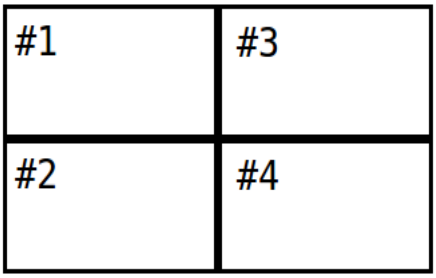


Illustration 8: Layout for a virtual quad display

To connect up the display servers and client, the node specification file is as follows:

```
{nodelist
        {nodedata
                {hostname "127.0.0.1"}
                {port 8082}
                {exid 0}
                {offx -0.25 }
                {offy  0.25 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8080}
                {exid 0}
                {offx -0.25 }
                {offy -0.25 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8083}
                {exid 0}
                {offx  0.25 }
                {offy  0.25 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8081}
                {exid 0}
                {offx  0.25 }
                {offy -0.25 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
        }
}
```

Text 7: Example node specification file for virtual quad display

For this example, the file is called “nodequads.txt”.

This allows each node to have an area of the client screen that is half the width and height of the client node. Because these displays are arranged 2x2, the offset to the centre-point of each server node in X and Y is a quarter of the width or height of the client node. The assignments of node ports are then as follows:

8080	8082
8081	8083

Illustration 9: Port numbers for quad display

If this display were to be modified to support a 3x3 configuration of displays, the width and height would be reduced to 0.3333333, while the offsets in each dimension would be multiples of 0.16666666

To get each server to run on the test platform, we run the following:

```
#!/bin/bash

# Set up a local virtual display of 2x2 quads.
# These have the following layout in terms of port numbers:
#
# 8080 8081
# 8082 8083

./compute -s -p8080 -b -x800x0 &
./compute -s -p8081 -b -x1312x0 &
./compute -s -p8082 -b -x800x512 &
./compute -s -p8083 -b -x1312x512 &


sleep 5

# Set up the client window. The file nodequads.txt defines the
# hostnames and port numbers for each remote display

./compute -c -nnodequads.txt
# Set up the client window. The file nodequads.txt defines the
# hostnames and port numbers for each remote display
```

Text 8: Script to set up quad view windows

The first four commands set up four servers each listening on a separate port, then the sleep command forces the script to wait while the windows are set up by the system. The next step is to get the interactive client running using the specification file.

Example 3: Configuring a seven unit hexagonal display

For this example, the goal is to set up a 7 segment display in a hexagonal shape:

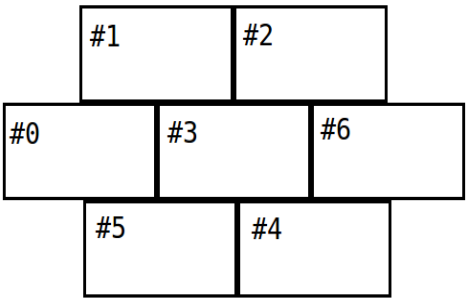


Illustration 10: Layout of a hexagonal display

To connect up the display servers and client, the node specification file is as follows:

```
{nodelist
        {nodedata
                {hostname "127.0.0.1"}
                {port 8080}
                {exid 0}
                {offx 0.0 }
                {offy  0.0 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777 }
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8081}
                {exid 0}
                {offx  -0.25 }
                {offy -0.5 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777 }
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8082}
                {exid 0}
                {offx 0.25 }
                {offy -0.5 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8083}
                {exid 0}
                {offx  0.5 }
                {offy  0.0 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
        }
```

```
   {nodedata
                {hostname "127.0.0.1"}
                {port 8084}
                {exid 0}
                {offx  0.25 }
                {offy  0.5 }
                {width 0.5}
                {height 0.5}
                {offheading 0.0}
                {aspect 1.777}
          }
          {nodedata
                  {hostname "127.0.0.1"}
                  {port 8085}
                  {exid 0}
                  {offx  -0.25 }
                  {offy 0.5 }
                  {width 0.5}
                  {height 0.5}
                  {offheading 0.0}
                  {aspect 1.777}
          }
          {nodedata
                  {hostname "127.0.0.1"}
                  {port 8086}
                  {exid 0}
                  {offx  -0.5 }
                  {offy 0.0 }
                  {width 0.5}
                  {height 0.5}
                  {offheading 0.0}
                  {aspect 1.777}
          }
}
```

For this example, the file is called “nodehex.txt”.

This allows each node to have an area of the client screen arranged in a staggered brick pattern. The port number assignments are as follows:

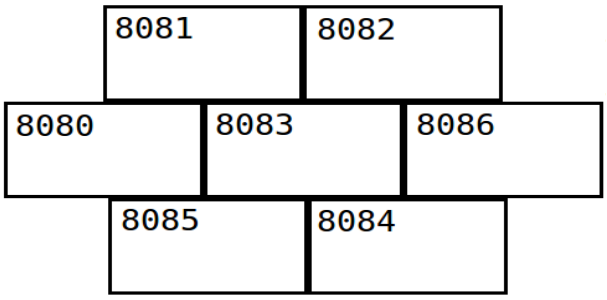


Illustration 11: Port assignments for a hexagonal display

The script file used to set up the servers and client is as follows:

```
#!/bin/bash

# Set up a local virtual display of 2x2 quads.
# These have the following layout in terms of port numbers:
#
#      #1    #2
#  #0    #3    #6
#      #5    #4

DIMENSIONS=-w160x90

./compute -s -p8080 -b -x160x90 $DIMENSIONS &
./compute -s -p8081 -b -x80x0 $DIMENSIONS &
./compute -s -p8082 -b -x240x0 $DIMENSIONS &
./compute -s -p8083 -b -x320x90 $DIMENSIONS &
./compute -s -p8084 -b -x240x180 $DIMENSIONS &
./compute -s -p8085 -b -x80x180 $DIMENSIONS &
./compute -s -p8086 -b -x0x90 $DIMENSIONS &

# Set up the client window. The file nodequads.txt defines the
# hostnames and port numbers for each remote display

./compute -c -nnodehex.txt
```

Text 9: Script to set up a hexagonal display

Example 4: Configuring a four unit windmill shape

For this example, the goal is to set up a 4 unit display in a shape of a windmill:

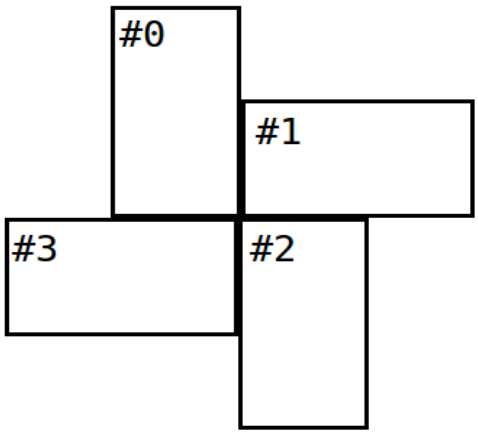


Illustration 12: Layout for a windmill shaped display

This requires calculation of the center points and dimensions of each screen. We start by calculating the total number of pixels in width and height based upon the pixel resolution of each screen. In this case, because our screens are 160x90 in dimensions (16:9 aspect ratio), and there are two end-to-end screens, the total dimensions are 320x320.

We can then calculate the fractional center-points from the pixel coordinates of each center point divided by the total width and height. In each case, the virtual width and height is double the value of the center point.

Unit	X-center (pixels)	Y-center (pixels)	X-center (fraction)	Y-center (fraction)	Width (fraction)	Height (fraction)	Angle
0	115	70	-0.140625	-0.25	0.28125	0.5	-90
1	250	115	0.25	-0.140625	0.5	0.28125	0
2	205	250	0.140625	0.25	0.28125	0.5	90
3	70	205	-0.25	0.140625	0.140625	0.28125	180

Table 11: Offsets and dimensions for a windmill display

For our virtual display on the local screen, all rotation angles must be 0, since X-windows doesn't have any concept of rotation. However, for each remote display, the rotation angle must be a multiple of 90 degrees.

To connect up the display servers and client, the node specification file is as follows:

```
{nodelist
        {nodedata
                {hostname "127.0.0.1"}
                {port 8080}
                {exid 0}
                {offx -0.140625 }
                {offy -0.25 }
                {width 0.28125}
                {height 0.5}
                {offheading 0}
                {aspect 1.0}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8081}
                {exid 0}
                {offx  0.25 }
                {offy -0.140625 }
                {width 0.5}
                {height 0.28125}
                {offheading 0.0}
                {aspect 1.0}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8082}
                {exid 0}
                {offx  0.140625 }
                {offy  0.25 }
                {width 0.28125}
                {height 0.5}
                {offheading 0}
                {aspect 1.0}
        }
        {nodedata
                {hostname "127.0.0.1"}
                {port 8083}
                {exid 0}
                {offx -0.25 }
                {offy  0.140625 }
                {width 0.5}
                {height 0.28125}
                {offheading 0}
                {aspect 1.0}
        }
}
```

Text 10: Script to setup a windmill display

For this example, the file is called “nodewindmill.txt”.

The port number assignments are then as follows:

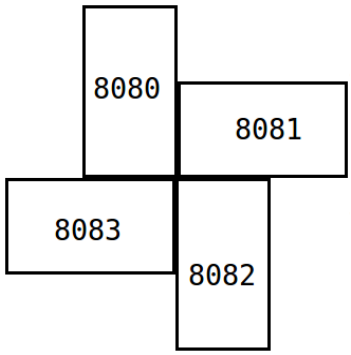


Illustration 13: Port assignments for a windmill display

The script used to set up the servers and client is as follows:

```
#!/bin/bash

# Create a windmill shape using four servers

./compute -s -p8080 -b -w90x160 -x70x0 &
./compute -s -p8081 -b -w160x90 -x160x70 &
./compute -s -p8082 -b -w90x160 -x160x160 &
./compute -s -p8083 -b -w160x90 -x0x160 &

sleep 5
# Now run the client

./compute -c -nnodewindmill.txt
```

Text 11: Script to setup a windmill display

Data flow within this application

The general architecture of this system is to have every module of software arranged in layers, as matched by the order of the files includes by the main header file “fractals.h”

The lowest layer is the network layer which manages connections between servers and clients. Above that is the definition for fractal parameters and lists of fractal parameters. This class object originally used to store fractal parameters to send to the compute shader is reused as a container to create and retrieve parameters send through data messages sent between clients and servers as well as between windows and compute shaders.

Between the network and GUI layers is an interface class which manages the network system as a whole and allows fractal parameters to be packed and unpacked into network messages.

At the lower levels of the GUI layer are classes to manage the compute shader used to evaluate selected regions of the Mandelbrot set. The middle levels provide the definitions used to maintain the GUI system. These include all the data required to store the state of a single window which includes keyboard, mouse and OpenGL state. Other classes are used to represent lists of windows and the state of the compute shader.

Operation of the networking system

Introduction

This section aims to explain how the networking system works in this application. There were several goals that formed the basis for this module. These were that:

- It should be modular – any number of displays can be set up
- It should be able to be tested on one or more systems
- Displays can be orientated in any angle

To achieve modularity across systems, the TCP/IP “sockets” library is used. This is a cross-platform library that allows a wide variety of networking methods from Ethernet (twisted pair), Wi-Fi, Bluetooth and other alternative methods of communication (eg. Dial-up PPP/SLIP) to be used transparently. Using TCP/IP allows the “sockets” library API to be used to set up network connections. As another benefit, this also allows the application to be tested on multiple systems. The use of TCP/IP also allows the application to make use of multi-threading to implement the client/server design pattern for receiving and sending data.

The use of sockets can be considered the data equivalent of a “Star Gate” in a Sci-Fi series. In these stories, “Star Gates” can be anywhere in the universe from a planet, to a remote orbiting satellite to a starship, and allow material objects to be sent across instantaneously without the need for physical transport across the universe.

In the same way, the Internet (built over TCP/IP and the sockets API) allows data to be sent across the Internet instantaneously without the need for the physical transportation of information in the form of books, magnetic tape or disk drives. The Internet itself consists of a variety of different types of dedicated system (client, servers, routers, bridges, firewalls, gateways) which each either send data requests, respond to data requests, route traffic, connect two similar but separate networks together, filter internet data, or connect different types of networks together. These are in turn connected together through all sorts of different communications methods: satellite links, microwave towers, FDDI fiber-optic cables, local area networks, wide area networks, wi-fi, twisted pair Ethernet cable with RJ45 connectors, SLIP/PPP running over dial-up lines. All of these operates transparently to the software running on both the client and server systems.

However, data can only be sent between two systems if they are both running and able to communicate across the network. For data exchange to occur, it is essential that one system is running as a “server” so that it can accept connections, receive data requests and send data responses. A data request can be anything from a request for a search on some text, a

web page to be sent back, or a file to be uploaded as well as downloaded, or even a comment to be added to a blog. A data response can either be the contents of a downloaded file, some data or even confirmation that a request was accepted or denied. This then requires that other systems run as “clients” which can send data and receive data responses. Thus any application built from TCP/IP and sockets must implement the client/server model where one system sends out a request, the other receives a request, creates and sends a reply, and the original system receives that reply. In some cases, those systems might even require an acknowledgement that the reply was received. One of the most simplest client-server functions is “ping” which simply sends an empty message from the client to the server and waits for the server to “echo” that reply. For the purpose of this application, a data request is a set of parameters that specifies the rendering of a fractal. When used locally they would be sent to the graphics engine to set up texture dimensions, color palettes, rendering options and the sample area of the fractal.

Overview of TCP/IP

The fundamental operation of sending data via TCP/IP is based on several functions:

- gethostbyname – find the Internet details of a host known by a name eg. “AlphaBase”
- gethostbyaddr – find the Internet details of host known by an address eg. 127.0.0.1
- inet_ntoa – convert a binary adddress (struct in_addr) into ASCII
- socket – open a socket (server)
- bind – bind a socket to a particular address (server)
- listen – make a socket a passive socket that arranges data connections (server)
- accept – create a new socket for a new data connection (server) (*3)
- connect – request a new socket (client)
- close – close a socket (both server/client)
- send – send a block of data sent through a socket (*1)
- read – read a block of data sent through a socket (both server/client)
- write – write a block of data sent through a socket (both server/client)
- recvfrom – read a block of data sent through a socket (both server/client) (*2)

(*1) options include not generating signals (MSG_NOSIGNAL) that would halt execution of the application.

(*2) options include not waiting for data to arrive (MSG_DONTWAIT) that would pause execution of the application

(*3) options include making the socket non-blocking

Setting up a server

Setting up a server using TCP/IP involves the following function calls:

- socket – to create a socket
- bind - to bind that socket to listen to a particular address and port number
- accept – to accept a new connection (this creates a new active socket)
- close – to close the connections

Setting up a client

Setting up a client is done in a similar way

- socket – to create a socket
- gethostbyname or gethostbyaddr – to find a server to connect to
- connect – to connect to the server

Data communication

Data can then be sent between the two using the various commands like “read”, “recvfrom”, “send”, “sendto” and their variants.

Shutting down a server or client

Shutting down both server or client is achieved simply by calling “close” on the currently open sockets. The remote end will pick this event up as a “broken pipe” and may generate an EPIPE signal when reading or writing unless caught or blocked.

Implementation

For this application, all the data and functions used to interact with TCP/IP and sockets is bundled into the class “CNetworkNode”. This includes the host name, the port, the listening and active sockets, make a socket non-blocking as well as the functions to read and write data.

Built on top of this is the “CNetworkSettings” class with defines whether a client or server should run, the external file used to specify the list of remote servers to send parameters to, and the server component of the application. This class also contains functions to create, send and receive the various messages that form a basic protocol.

The class used to define a set of parameters used to render a single fractal image “CFractalParameters” is extended so that it can contain the contents of every possible message, as well as support the packing and unpacking of data into a binary format.

From these classes, it is possible to implement the two thread functions that perform the role of client and server. The server handles network events, while the client handles user events. Functions are called from the client side to send parameters, pings and requests for parameters, while the server side receives parameters, responds to pings and send parameters in response to parameter requests. The implementation of these two functions is compact and is very close to the pseudo-code specification.

The actual implementation within this application is through the following class hierarchy:

The parameters used to define a single host address/port/window are stored in CNetNodeEntryData. This is inherited by CNodeDataIO to add file IO routines.

```
CNetNodeEntryData
     hostname
     port
     windowid
     offsetx
     offsety
     width
     height
     offsetheading
     aspect ratio

CNodeDataIO
     CNetNodeEntryDataAsciiIO
     CNetNodeEntryData
```

The fundamental unit of communication is the network node which defines the server and client state. This is the class CNetworkNode. This class is built on top of the network node data file IO routines (CNodeDataIO). The network node has the following functions:

- Open client
- Close client
- Open server
- Close server
- Write message
- Read message
- Set non-blocking mode
- Match hostnames

Network Nodes can be placed in a STL vector array to define lists of servers stored in the class “CNetNodeEntryListData”. This class is inherited by CnetNodeEntryListIO

```
CnetNodeEntryListData
     vector <CNetworkNode *>

CNetNodeEntryListIO
     CNetNodeEntryListDataAsciiIO
          CVirtualFileSystemGlobalAscii
          CNetNodeEntryListData

CNetworkNodeList
     CNetNodeEntryListIO
```

All the parameters related to this client and all the servers are bundled into the class CNetworkSettings.

```
CNetworkSettings
     CNetworkNodeList (list of servers)
     CNetworkNode     (this server)
```

The class CNetworkSettings has the following tasks:

- Open all connections to other servers from client (Client → Servers)
- Close all connection to other servers from client (Client → Servers)
- SendBroadcast (Client → Server)
- SendPing (Client → Server)
- SendRequest (Client → Server)
- SendQueryWindowSize (Client → Server)
- SendTerminate (Client → Server)
- SendParameters (Client → Server)
- ReceiveParameters (Server ← Client)
- SendPingReply (Server → Client)
- SendRequestReply (Server → Client)
- SendQueryWindowSizeReply (Server → Client)

Those functions labeled (Client → Server) send data from the client to the server. Those functions labeled (Server → Client) send data from the server back to the client. The function ReceiveParameters is the only exception as it used by the server to received data from clients.

While the network layer classes deal with the maintenance of connections between sent the client and server half of applications, the actual data itself is packed and unpacked to and from fixed size messages through the class CFractalParameters. In this way, the processing of the contents of messages is kept separate from the network layer.

Operation of the compute shader

The computer shader is the core of this application. Rather than using one CPU core or even sixteen or more (using latest Intel or AMD CPU's), all calculations are performed on the GPU via an OpenGL compute shader, taking advantage of the vast number of cores available on the GPU. This can range anywhere from a handful to over 6000 on high-end GPU's.

Running a compute shader simply involves providing suitable source code to the OpenGL driver, compiling that source code into a shader program, passing references to a source texture (the color palette), and a destination texture (the rendered image). The fractal parameters are passed to the compute shader via uniform variables.

The source code to the compute shader itself is about 150 lines long, and is effectively all the calculations required to process a single pixel for a fractal image.

The compute shader itself has several tasks:

- Convert a parametric value into a color value from the color palette (or transfer texture)
- Provide routines to convert between complex and polar coordinates
- Provide several different Mandelbrot functions (power of 2, power of 3, power of N)
- Transform a parametric coordinate in the range (0.0, 0.0) to (1.0, 1.0) to coordinates in the compex plane
- Convert compute shader thread IDs into a parametric coordinate, transform this into a complex coordinate, perform the Mandelbrot function iteration, calculate a parametric value to use to generate a color, and write this color into the destination image texture.

Depending on the resolution of the texture image, a whole frame can be rendered between real-time rates and five frames/second, considerably faster than an equivalent CPU function.

Operation of the windows system

The windows systems used by this application is designed to allow multiple windows to provide different views of the fractal using different parameters and color palettes. There are several classes used to implement this module:

- CWindowState – Stores the current state of all keyboard key presses and releases
- CShaderUtils – Helper class used to compile shaders
- CWindowCompute – Class used to implement the compute shader
- CWindowParameters – Complete definition for a single window
- CWindowParametersList – List of window parameters
- CWindowSettings – Defines a set of default parameters for all windows

Of all the classes defined here, CWindowParameters is the most important as it bundles together the GLX context, the render shader, the window ID, a mutex used to provide exclusive updates of the window, a CPU timer and a flag allowing or denying the server to provide parameter updates.

Provided functions allow for the creation of the GL context, basic geometry and the various textures used. Other functions are used to handle input events and to process actions such as the automatic update of rotation, zoom and color cycling.

Two functions, “setparameters” and “getparameters” are used to get and set the fractal parameters using the safety of a mutex lock.

Prints of the current texture can be saved to disk through the use of the “savescreen” and “savesuperscreen” functions.

Future work

There are many areas that this application could be improved. These include the areas of user interface for setting rendering options, more advanced rendering methods and more complex mathematical functions. Probably the most important would be a simpler way of setting up a distributed rendering system since this presently requires the manual editing of screen layouts.

User Interface for setting rendering options

The present user interface simply involves using keyboard commands to enable and disable various options or to increase or decrease them. This could be adapted for mobile use through a touch screen.

More advanced rendering methods

The current system uses a single compute shader to evaluate a single point in the Mandelbrot set, implement a transfer texture (or color palette) with color cycling, and to perform smooth shading as an alternative to the traditional iteration count method. There are many other rendering methods available today which are based on various transformations on the set of iteration points. These include calculating the “average curvature, the “triangle inequality” method, and “stripe average” to name a few.

More complex mathematical functions

Applications such as XAOS provide a rich variety of mathematical function including complex polynomials which are a combination of power terms (p^a + p^b + p^c) rather than the single (p^2) than most viewers use. This provides an unlimited range of functions.

Easier display configuration setup

At present, setting up a multi-display system involves specifying configuration files for each server/client, along with having to define the aspect ratio of the remote system locally, rather than being able to query it. Defining a configuration file involves performing manual calculations in order to determine the layout of screens and windows in terms of relative offsets, center-points and aspect ratios. This could be improved by having a central display system manager operating in the same way as a telephone exchange.

Conclusions

This application demonstrates how it is possible to combine multi-threading, networking and GPU programming techniques in order to provide a simple but fun interactive application. Adding new features is an open-ended project since there are so many different directions that can be taken, especially with the constantly evolving hardware and fractal rendering techniques.

Appendix A - ASCII Data Exchange Format

To allow exchange of data through E-mail as well as the ability to load parameter files and replay files, a tagged ASCII file format has been designed. This has the following specification:

Individual data blocks

Each text begins with a header block of:

```
{fractal
```

The contents of the fractal parameters are specified with:

```
{escaperadius 4 }
{xcen -1.31822 }
{ycen -0.0788025 }
{xwidth 9.67938e-05 }
{ywidth 9.67938e-05 }
{maxiter 256 }
{angle 0 }
{power 2 }
{banding 24 }
{modesmooth 1 }
{insidecolor 0 0 0 0 }
{modepower 0 }
{texpalette 0 }
{makemipmaps 0 }
{minclamp -10 }
{texwidth 8192 }
{texheight 8192 }
{cycleoffset 0 }
{modeinvert 0 }
{autocycle 0 }
```

This is terminated with a closing character

```
}
```

Appendix B - Binary Data Exchange Format

The binary data exchange format is a more compact and precise representation that the ASCII data format with the intention for use with network communication. Each message is tagged as before, but in a binary format. The use of field tags allows new types messages to be created while still using the existing routines.

Each field is identified with a two letter code stored as two bytes. The field codes are as follows:

Field code	Purpose	Data type	Details
ss	Start	None	Start of message (no other data)
er	Escape Radius	Float	Fundamental escape radius
mx	X-axis center	Float	Point at center of frame (X coordinate)
my	Y-axis center	Float	Point at center of frame (Y coordinate)
wi	X-axis Width	Float	Width of sampling region visible on screen
he	Y-axis Height	Float	Height of sampling region visible on screen
mi	Max Iterations	Integer	Maximum number of iterations
an	Angle	Float	Rotation angle of sampling region
pw	Power	Float	Power factor for calculations
ba	Banding	Float	Banding scale for stepped rendering
ms	Smoothing mode	Boolean	Whether or not to use smooth rendering
mp	Power mode	Integer	Which power mode shader should use
te	Texture palette index	Integer	Which texture to use for rendering
mm	Make mip-maps	Boolean	Whether or not to create mipmaps after compute
mc	Min. clamp value	Float	Value used when smoothing mode is enabled
tw	Texture width	Integer	Width of texture used for calculations
th	Texture height	Integer	Height of texture used for calculations
co	Color cycle offset	Integer	Color cycle offset
ac	Auto cycle	Integer	Automatic color cycling rate
mv	Invert lookup mode	Integer	Whether or not to invert banding
ee	Finish	None	End of message (no further data)

Neither the start and finish fields contain any data. No further information follows the end field tag. To allow for communications between windows of different servers and clients, the following fields are also available:

wf	Window from	Integer	ID of the window sending message
wt	Window to	Integer	ID of the window to receive message
po	Internet port	Integer	Internet port to send reply to
ww	Window width	Integer	Width of the selected window in pixels
wh	Window height	Integer	Height of the selected window in pixels
ad	Internet address(IPv4)	Address	Internet address to send reply to

Appendix C - Replay file format

Replay files used the same block format as single data blocks, but are placed within a container. The file has a prefix tag as well as a hint to the number of blocks contained inside:

```
{fractallist
     {listnum 3}
```

Text 12: Example animation file

For an animation consisting of three frames, they are as follows:

```
        {fractal
                {escaperadius 4 }
                {xcen -0.5 }
                {ycen 0 }
                {xwidth 4 }
                {ywidth 4 }
                {maxiter 64 }
                {angle 0 }
                {power 2 }
                {banding 24 }
                {modesmooth 1 }
                {insidecolor 0 0 0 0 }
                {modepower 0 }
                {texpalette 0 }
                {makemipmaps 0 }
                {minclamp -10 }
                {texwidth 8192 }
                {texheight 8192 }
                {cycleoffset 0 }
                {modeinvert 0 }
                {autocycle 0 }
        }
```

Text 13: Example animation file (continued)

```
        {fractal
                {escaperadius 4 }
                {xcen -0.712092 }
                {ycen 0.272935 }
                {xwidth 0.220372 }
                {ywidth 0.220372 }
                {maxiter 64 }
                {angle 0 }
                {power 2 }
                {banding 24 }
                {modesmooth 1 }
                {insidecolor 0 0 0 0 }
                {modepower 0 }
                {texpalette 0 }
                {makemipmaps 0 }
                {minclamp -10 }
                {texwidth 8192 }
                {texheight 8192 }
                {cycleoffset 0 }
                {modeinvert 0 }
                {autocycle 0 }
        }
        {fractal
                {escaperadius 4 }
                {xcen -0.710428 }
                {ycen 0.269771 }
                {xwidth 2.43277e-05 }
                {ywidth 2.43277e-05 }
                {maxiter 512 }
                {angle 0 }
                {power 2 }
                {banding 24 }
                {modesmooth 1 }
                {insidecolor 0 0 0 0 }
                {modepower 0 }
                {texpalette 0 }
                {makemipmaps 0 }
                {minclamp -10 }
                {texwidth 8192 }
                {texheight 8192 }
                {cycleoffset 0 }
                {modeinvert 0 }
                {autocycle 0 }
        }
}
```

Text 14: Example animation file (continued)

The file is terminated by a closing character. There is no limit to the size of this file.

Appendix D - Source code files and headers

The file “fractals.h” defines a general hierarchy of how the classes are used:

The following kernel and system files are used:

```
// ----- Kernel/systems programming -----

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <assert.h>
```

Text 15: Main header file "fractals.h"

The following C++ STL classes are used:

```
// ----- C++ library -----

#include <string>
#include <vector>
#include <iostream>
#include <sstream>
#include <iomanip>
```

Text 16: Main header file "fractals.h" (continued)

For ASCII file parsing and texture reading, the following files are used:

```
// ----- File reading/writing -----

#include "filesysascii.h"
#include "texlib.h"
```

Text 17: Main header file "fractals.h" (continued)

High precision CPU timers are used for timing functions:

```
// ----- File reading/writing -----

#include "cputimer.h"
```

Text 18: Main header file "fractals.h" (continued)

Network node management is defined by those files prefixed with net*

```
// ----- Network configuration -----

#include "netnodeentryio.h"
#include "netnode.h"
#include "netnodeentrylistio.h"
```

Text 19: Main header file "fractals.h" (continued)

Fractal parameters and lists of fractal parameters are defined along with the file IO management:

```
// ----- Fractal parameters -----

#include "fractalparamsio.h"
#include "fractalparams.h"
#include "fractalparamslistio.h"
#include "fractalparamslist.h"
```

Text 20: Main header file "fractals.h" (continued)

Compute shader formats are also defined:

```
// ----- Compute shaders -----

#include "computeformats.h"
```

Text 21: Main header file "fractals.h" (continued)

GUI level management of animation playback and window events is also performed:

```
// ----- GUI level -----

#include "glx.h"
#include "fractplayer.h"
#include "fractwindow.h"

#include "fractnet.h"
#include "fractsystem.h"
```

Text 22: Main header file "fractals.h" (continued)

The class hierarchy in more detail is as follows:

Low level file IO

texlib.h

- CTexture - class used to read/write image data

filesysascii.h

- file parsing lexers

CPU timer

- CTimerCPU
- High precision timer used for color cycling

Network System

netnodeentryio.h

netnodeentryio.cpp

- CNodeDataIO
- class encapsulating all file input/output for a network node

netnode.h

netnode.cpp

- CNetworkNode
- class encapsulating all network communications

netnodeentrylistio.h

netnodeentrylistio.cpp

- CNetNodeEntryListIO

Fractal Parameters

fractalparamsio.h

fractalparamsio.cpp

- CFractalParamsIO

fractalparams.h

fractalparams.cpp

- CFractalParameters

– class storing all parameters for a single fractal image

fractalparamslistio.h

fractalparamslistio.cpp

- CFractalParamsListIO

– class encapsulating all file input/out for a list of parameters

fractalparamslist.cpp

- CFractalParametersList

- class encapsulating an entire list of fractal parameters

Compute Shaders

computeformats.h

- Defines sets of macro names that are compatible with compute shaders

X-windows/OpenGL

glx.h

glx.cpp

- CGLXWindow
- Encapsulates all X-window data elements (Display, Window, GLXContext) into a single class object
- CGLXWindowList
- Defines a vector list of CGLXWindow's

GUI System

fractplayer.h

- CframePlayer
- Simple animation frame player

fractnet.h

fractnet.cpp

- CNetworkNode List
- CNetworkSettings
  - CNetworkNodeList
  - CNetworkNode
- Handles network communications

fractwindow.h

fractwindow.cpp

- CWindowState
- Stores all the keyboard/pointer state in a single class
- CwindowCompute
- Stores all the parameters related to the use of a compute shader
- CWindowParameters
- Stores all the state related to a single GLX window
- CWindowParametersList
- Stores a vector list of CwindowParameters

fractsystem.h

fractsystem.cpp

- CSystemParameters
- All parameters related to setting up an interactive client
- Contains:
    - CFractalParametersList
    - CNetworkNodeList
        - list of additional displays accessed via network nodes
    - CNetworkNode
        - current network node
    - CWindowList
        - List of local windows

fractserver.cpp

- Server thread to handle network events

fractclient.cpp

- Client thread to handle user events

main.cpp

Client and Server thread interaction

Interaction between the client and server can be described using the follow table

Server thread	**Client thread**
Startup server.open_server server.open_serverconnection	Startup For each window mutex_lock glxMakeCurrent Calculate viewport Initialize context mutex_unlock

Main loop
Receive message
 	
Process one of:
 	
SendPingReply
 	
ReceiveParameters
 	
SendRequestReply
 	
for each window
mutex_lock
window.setparameters
mutex_unlock
	For each window
	mutex_lock
	process_action
	mutex_unlock
	
	For each window
	mutex_lock
	process_action
	glxMakeCurrent
	resize_context
	compute_gpu
	calculate_viewport
	mutex_unlock
	
	if (client_mode)
	{
	for each node
	open_client
	
	for each window
	mutex_lock
	sysparams.SendParameters
	mutex_unlock
	
	for each window
	swapbuffers

Shutdown
server.close_server
	Shutdown
	if (client_mode)
	{
	for each node
	close_client
	}

References

1. “The Fractal Geometry of Nature” - Benoit Mandelbrot

2. “On Smooth Fractal Coloring Techniques - “Jussi Härkönen”