Multi-threaded GUI Design

An Object Oriented Approach

By Christopher J. Slominski

GUI Application Requirements

- Responsive GUI
  - GUI does not freeze when handling a lengthy request
- Simple design
  - Reliable
  - Maintainable

Typical GUI Design

- Program execution is turned over to an event dispatcher, which calls program event handlers.
  - Keyboard /Mouse
  - Window manager
  - System clock
  - I/O completion

Serial Execution Score Card

- GUI works well when …
  - Event handlers complete rapidly
- GUI works poorly when …
  - Event handlers perform lengthy procedures
  - Other procedures want the "driver's seat"

Unresponsive GUI Solutions

- Timer events
- Idle time processing
- Forced event loop iterations
- Multi-tasking
- Multi-threading

Unresponsive GUI Solutions

- **Timer events** 
- Idle time processing
- Forced event loop iterations
- Multi-tasking
- Multi-threading

Unresponsive GUI Solutions

- Timer events 
- **Idle time processing** 
- Forced event loop iterations
- Multi-tasking
- Multi-threading

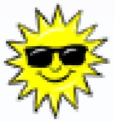
Unresponsive GUI Solutions

- Timer events
- Idle time processing
- **Forced event loop iterations**
- Multi-tasking
- Multi-threading

Unresponsive GUI Solutions

- Timer events 
- Idle time processing 
- Forced event loop iterations 
- **Multi-tasking** 
- Multi-threading

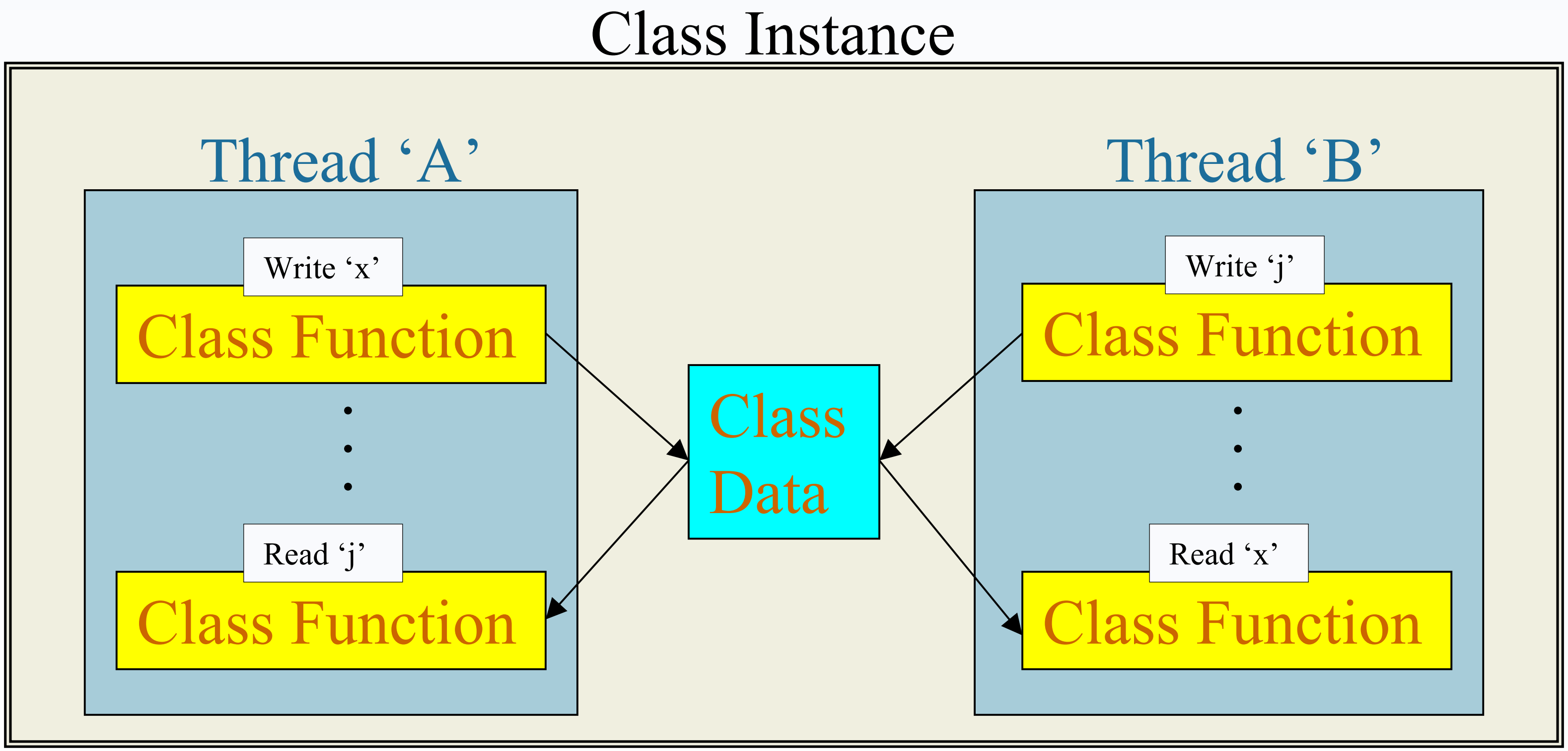
Unresponsive GUI Solutions

- Timer events 
- Idle time processing 
- Forced event loop iterations 
- Multi-tasking 
- **Multi-threading** 

Multi-threaded Objects

- Leaves the job of breaking execution into discrete chunks up to the OS.
- Exposes public interfaces of objects to other instances of execution.
- Complexity of threading mechanism hidden from developer through inheritance.

Thread/class Interaction



Sample C++ Code (1 of 2)

```
// This class creates a GUI that handles event
// processing in its own thread of execution
class GUI : public Thread
{
  public:
    void NewData(double *value)
    { // Runs in context of another thread.
      // Perform mutual exclusion and initiates
      // display update with the new value.
    }

  protected:
    int ThreadMain(void)
    { // create GUI and give this thread's
      // control to the event dispatcher.
    }
};
```

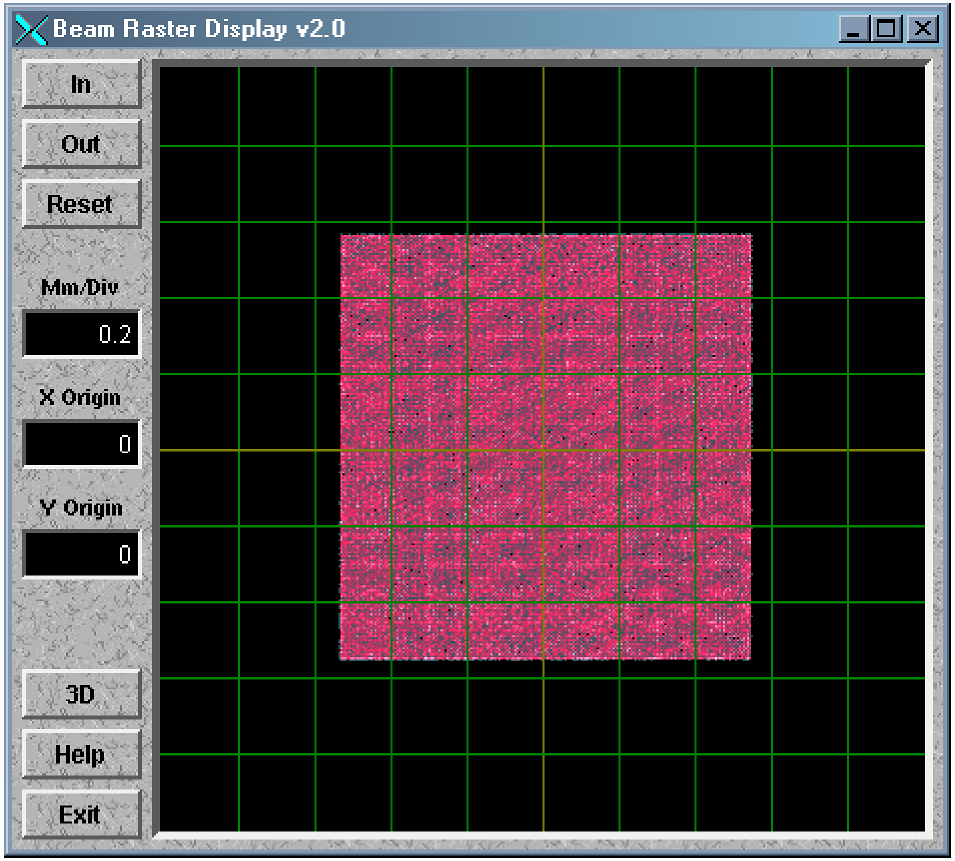
Sample C++ Code (2 of 2)

```
// This is the channel access monitor callback
// routine.
void Handler(event_handler_args args)
{
  GUI *gui = static_cast<GUI *>(args.usr);
  gui->NewData(static_cast<double *>(args.dbr));
}


// The main thread instantiates a GUI object
// and turns execution over to EPICS to
// monitor for a PV update.
int main(void)
{
  GUI myGui;

  // Set up a channel access monitor and turn
  // execution over to EPICS.
  ca_add_masked_array_event(Handler, &myGui)
  ca_pend_event(0);
}
```

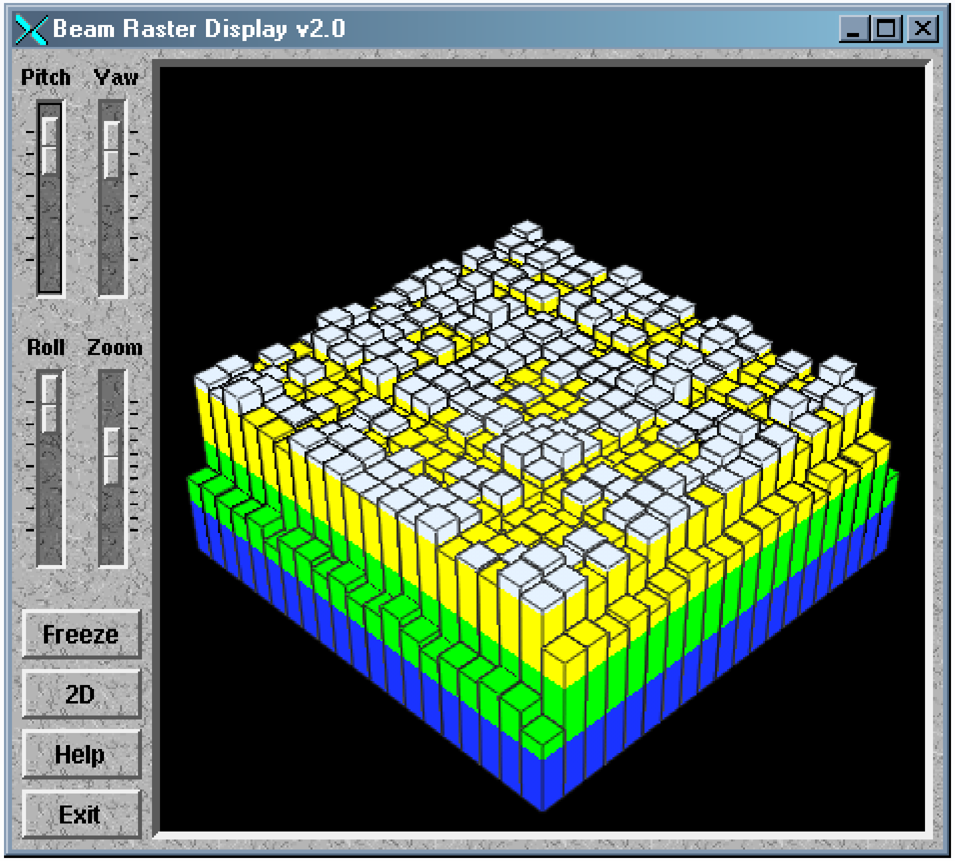
Example Application

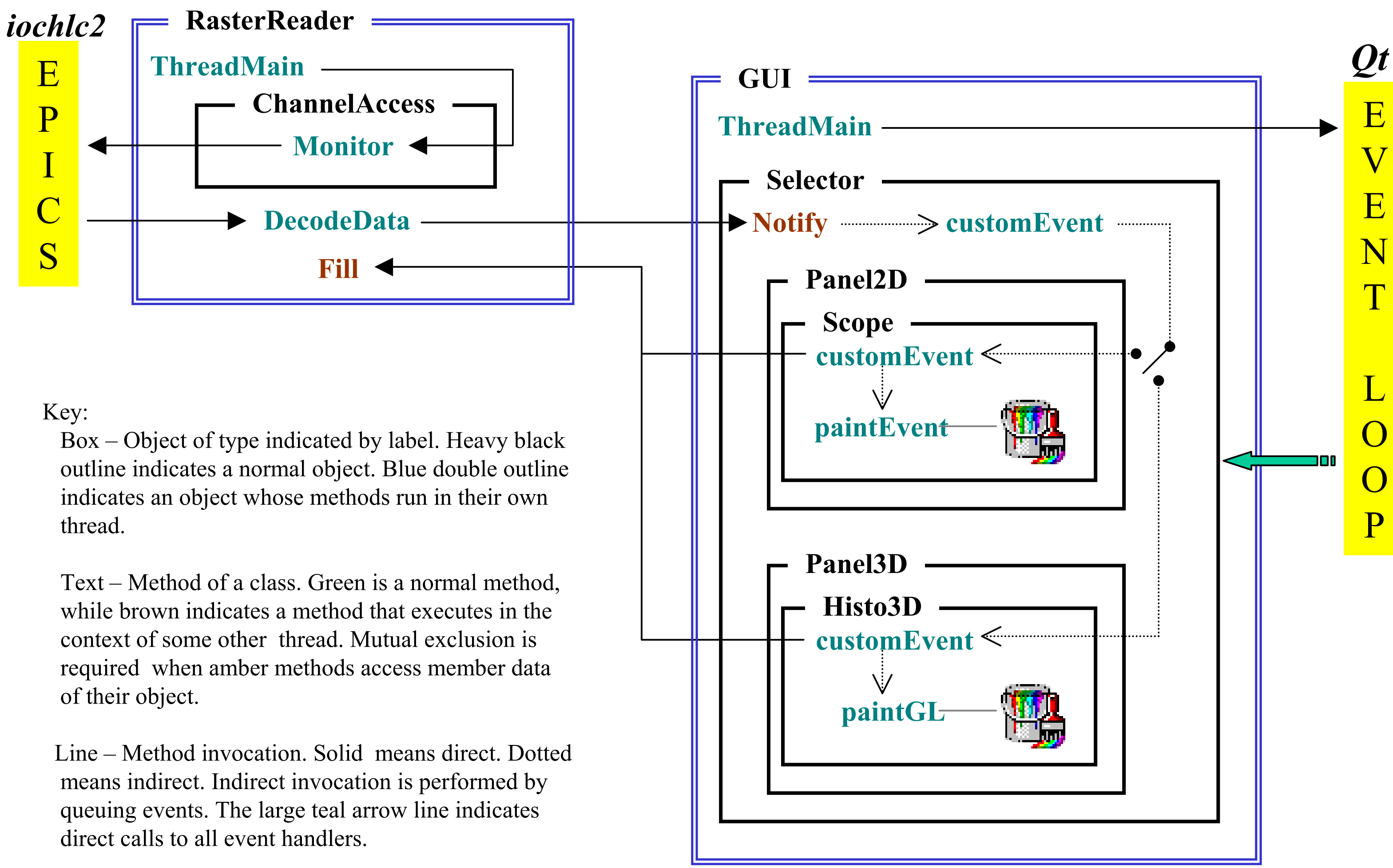
The Beam Raster Display

2D Display


Beam Raster Display v2.0
In
Out
Reset
Mm/Div
0.2
X Origin
0
Y Origin
0
3D
Help
Exit

3D Display


Beam Raster Display v2.0
Pitch
Yaw
Roll
Zoom
Freeze
2D
Help
Exit



Key:

Box – Object of type indicated by label. Heavy black outline indicates a normal object. Blue double outline indicates an object whose methods run in their own thread.

Text – Method of a class. Green is a normal method, while brown indicates a method that executes in the context of some other thread. Mutual exclusion is required when amber methods access member data of their object.

Line – Method invocation. Solid means direct. Dotted means indirect. Indirect invocation is performed by queuing events. The large teal arrow line indicates direct calls to all event handlers.