Chain of Responsibility and Iterator

2-3-2003

Opening Discussion

- Do you have any questions about the items that you read for today before we start the quiz?

Behavioral Patterns

- The third major type of pattern is that of behavior patterns. These patterns deal with algorithms and the communication between objects.
- These patterns typically do things like make it easy to switch what algorithm is used to do something or to abstract the way that we get information from a class so the implementation of that class can be easily changed.
- Remember, abstract that which varies.

Chain of Responsibility

- This pattern is something like a daisy chain in software. You use it to avoid coupling the sender to the receiver when a request is being made.
- We have a class of objects that can receive requests that are linked in a list. When an object receives a request it can either handle it, or pass it on to the next in the chain.
- The client object makes a request of the first in the chain, typically the most specific.

Example

- One example of this would be in event handling where an event might pertain to one of several objects.
- For example, a button in a panel, in a window. Some requests the button can handle, others it passes on to the panel. If the panel can't handle it, it will pass it all the way out to the window and the full application.
- The old Java event model was built something like this.

Benefits and Drawbacks

- The primary benefit of this pattern is that it provides lose coupling between the sender and the receiver.
- It also makes it very easy to add a new potential receiver for a message or new ways a message can be received. All one must do is add a new object to the list.
- The one real drawback is that there is no way to be sure that anyone handled it. In some applications this doesn't matter, but if the message must be dealt with then this pattern might not be ideal.

Iterator

- This is a pattern you should all be somewhat familiar with. It provides a generic way to sequentially examine the elements of different types of containers.
- The idea is that iterators adhere to a certain interface and each container class has one or more concrete iterator implementations for it.
- Outside code can then run through the contained objects sequentially without knowing anything about the structure of the container if it has the required interface.

Example

- This pattern probably doesn't need one, but imagine any set of collection types and code that should be able to traverse the elements of the collection regardless of its type.
- For example, we could store things in a linked list or a binary tree. As long as they can provide an Iterator, code can traverse either without knowing what type it has.
- You could also easily make Iterators for forward, backward, in-order, pre-order, post-order, etc.

Benefits and Drawbacks

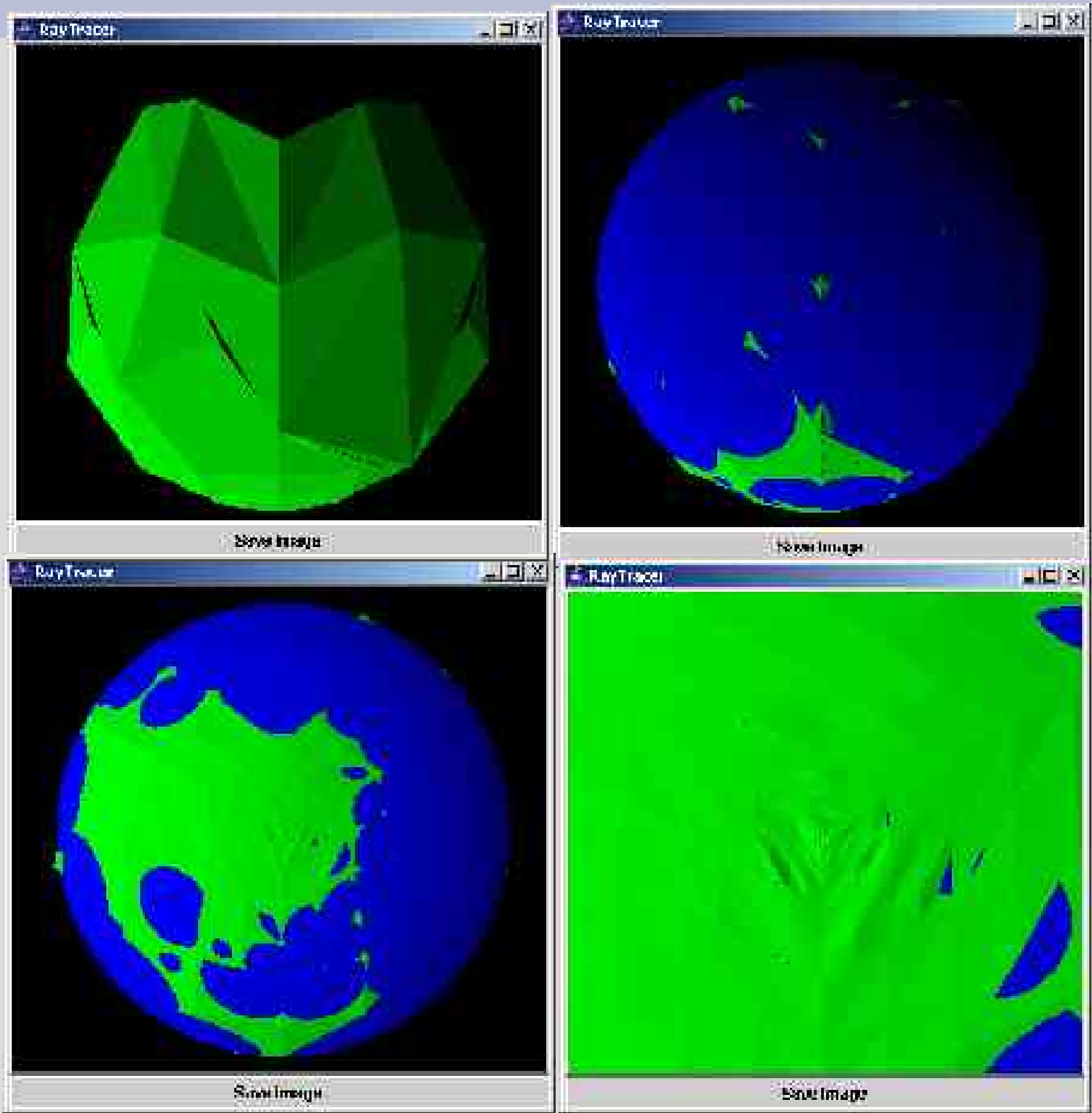
- Can have many different types of iterators on a container without complicating the container interface too much.
- Keeps the interface of the container simpler in general.
- Allows for multiple traversals of the container to be in progress at once. This can be helpful in many situations, but is essential if the application is multi-threaded.

Progress Report

- I'd like to get a quick update from the groups on what they are doing and what progress they have made. Also, I'd like people who make early progress or have neat designs to present them to the class each week.
- Chris Hohimer has volunteered to discuss the work that he is doing on L-Systems for trees in the C++ side of the project today.

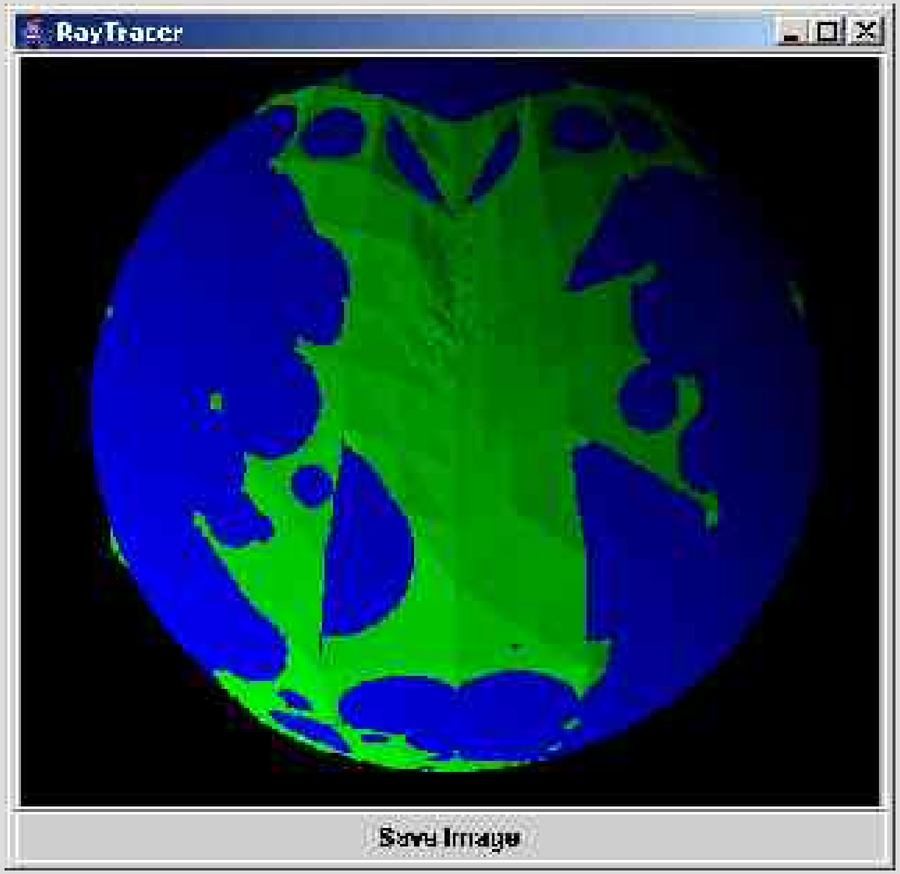
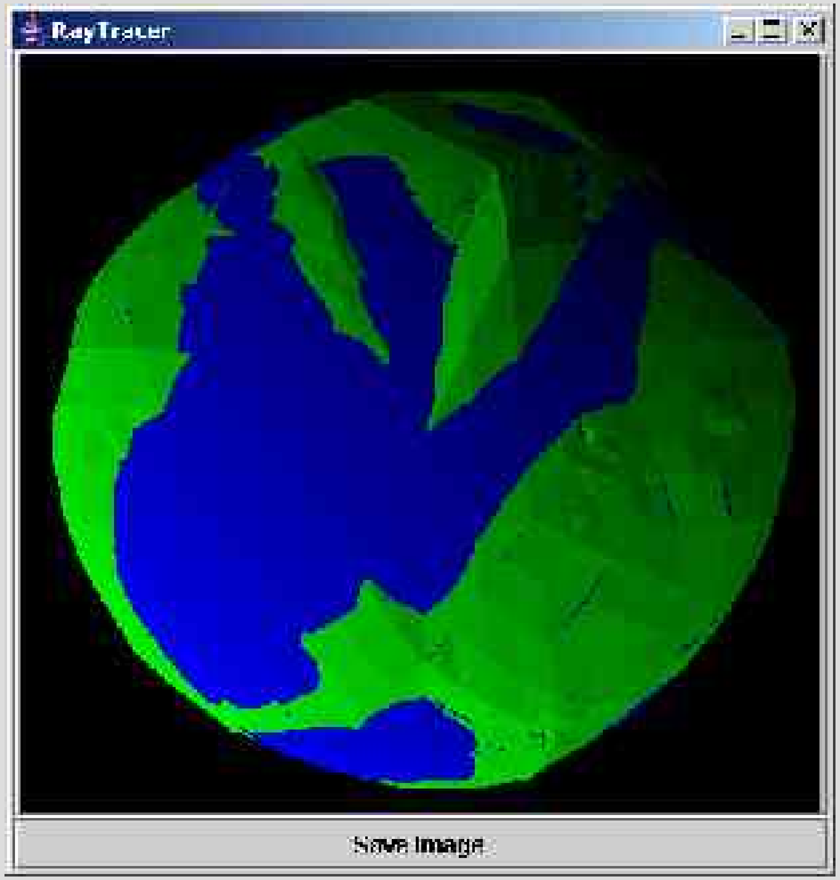
A Spherical World

- Here are some images showing hierarchical, dynamic loading in a spherical world.



Multiple Viewers

- These shots have 2 and 100 viewers.


RayTracer
Save Image

RayTracer
Save Image