C++ Advanced

Scenes and Layers

Lesson Objective

- Understand Scenes
  - A collection of graphics which produce an image
- Understand Layers
  - Simplifies complex scenes by grouping related graphics
- Be able to implement and test Scene and Layer classes

Scenes

- A collection of objects and their orientation to each other
- Gives the sense that objects may be moved
- Multiple images may be made from the same scene

Layers

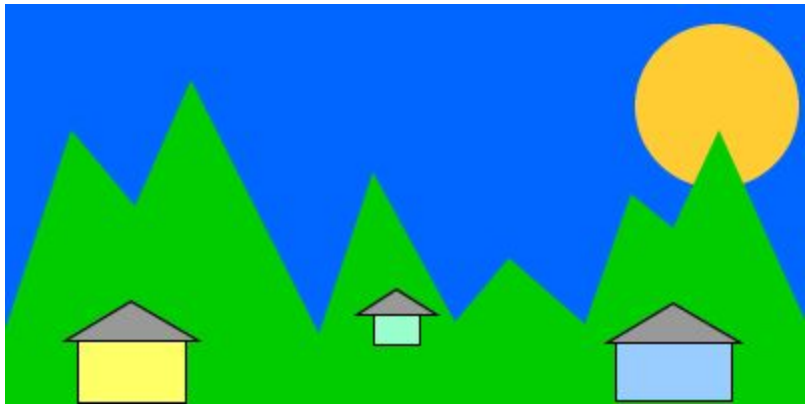
- A collection of related graphics that can be referred to as a whole
- A layer has no width or height
  - It always takes the size of a scene
- Primary features
  - Locking – (i.e., disabling modification)
  - Hiding
- Primary purpose for now is organizational

Layer Example

Sky
Layer

Houses
Layer

Mountain
Layer



Entire
Scene

Relative Placement & Ordering

- One way to look at a scene
  - Collection of vector graphics
  - Graphic ordering
  - Graphic positioning
- What we'll do
  - Define graphics relative to graphic origin
  - Make layer class responsible for graphic placement
  - For each graphic a layer must know the graphics position

Relative Placement Ideas Pros/Cons

- Option A) Primitive
  - `typedef list<pair<Point, VectorGraphic> PlacedGraphicCollection`
- Option B) Class
  - class PlacedGraphic

```
class PlacedGraphic {
    public:
        void setPlacementPoint (Point const& placement);
        Point const& getPlacementPoint () const;

        void setGraphic (HVectorGraphic const& graphic);
        HVectorGraphic const& getGraphic () const;

    private:
        Point placementPoint;
        HVectorGraphic graphic;
};
```

Placement Depth

- Z-Layering
  - Each graphic has a unique z-component
  - Graphics are drawn in z order back to front
  - Our application won't use this style
- Relative Ordering
  - Allows user to move object to back or front
  - Conceptually like ordering sheets of paper
  - Fits user understanding well

Understanding collections and iterators

- Collections provided by standard library
- You'll likely never need write your own collection – use standard ones
- C++ collections are provided in the library – they are not part of language
- Except – arrays are part of language
  - For backward compatibility
  - Prefer library collections over arrays (why?)

Collections

- Ordered collections
  - Vector
  - List
  - Deque
- Sorted collections
  - Set (no duplicates)
  - Multi-set (allows duplicate elements)
  - Map (no duplicate keys)
  - Multi-map (allows duplicate keys)

Collections are not inter-changable

- Each do some things well
- Each do some things poorly
- Use the collection interface
  - Provides methods for what the collection does well
  - No methods for what the collection does not do well
- Examples
  - Vector does not provide remove methods
  - List does not provide random access methods

Misguided use of vector

```
class Principal {
    private:
        typedef std::vector StudentCollection;
        typedef StudentCollection::iterator StudentIterator;

    public:
        void enrollStudent (HStudent const& student) {
        getStudents ().insert (getStudents ().begin (), student); }

        void expellStudent (HStudent const& student) {
        for (StudentIterator iStudent = getStudents ().begin ();
            iStudent != students.end ();
            ++iStudent) {
            if (*iStudent == student) {
               students.erase (iStudent);
               return; // if we continue to iterate we will crash
                       // because our iterator is invalidated
                       // should use remove algorithm with erase. }
            }
        }

    private:
        StudentCollection students;
};
```

Using a list instead

```
class Principal {
    private:
        typedef std::list StudentCollection;
        typedef StudentCollection::iterator StudentIterator;

    public:
        void enrollStudent (HStudent const& student) {
            getStudents ().push_front (student);
        }

        void expellStudent (HStudent const& student) {
            getStudents ().remove (student);
        }

    private:
        StudentCollection students;
};
```

Example Summary

- The list is a better choice
  - Operations we require fit a list better
  - Less lines of complicated code
- To use iterators effectively we must know which operations invalidate them
- Avoid code where active instances become invalid
  - E.g., Vector solution must handle invalid iterators

List

- ISO standard does not specify list implementation
- Likely implementation a doubly linked list
- Adds and removes elements quickly
- Shrinks and grows efficiently
- List iterators are persistent
  - Iterators always valid (except when iterator references a removed element)
- Poor random access of elements
  - May only move forward or back one element at a time

List continued

- splice – move all elements within a range to another list
- merge – combined two sorted lists maintaining the sort
- sort – sorts the elements
- reverse – reverses the order of elements
- unique – removes duplicate consecutive elements

Vector

- A better array
- Provides fast random access
- Good performance adding/removing elements at end
- Poor performance adding or removing elements from the middle

Deque

- Pronounced “deck”
- Similar to vector
- Provides fast random access
- Good performance adding/removing elements at beginning or at end
- Poor performance adding or removing elements from the middle

Choose the right collection

- All collections will perform reasonably well with a small number of elements
- With a large number of elements performance depends on
  - Common operations
  - Collection choice
- Choose the collection
  - That results in the most simple code
  - That performs best with the most common operations your application requires

Layer class (so far)

```
namespace Framework {
    class Layer {
        private:
            typedef std::list PlacedGraphicCollection;
        public:
            typedef PlacedGraphicCollection::iterator
                PlacedGraphicIterator;

            // insert, remove, iteration support.
            // accessors to alias.
        private:
            PlacedGraphicCollection graphics;
            std::string alias;
    };
}
```

Scene class (so far)

```
Namespace Framework {
    class Scene {
        private:
            typedef std::list LayerCollection;
        public:
            typedef LayerCollection::iterator
                LayerIterator;
            // insert, remove, iteration support.
            // accessors to width and height.
        private:
            LayerCollection layers;
            int width;
            int height;
    };
}
```

Scenes and Layers in XML

```
<Scene width="800" height="600">
  <Layer alias="sky">
    <PlacedGraphic x="0" y="0">
      <VectorGraphic closed="true">
        <Point x="0" y="10" />
        <!-- etc... -->
      </VectorGraphic>
    </PlacedGraphic>
    <PlacedGraphic x="700" y="0">
      <VectorGraphic closed="true">
        <!-- etc... -->
      </VectorGraphic>
    </PlacedGraphic>
  </Layer>
```

```
  <Layer alias="mountains">
      <PlacedGraphic x="0" y="0">
        <VectorGraphic closed="false">
          <!-- etc... -->
        </VectorGraphic>
      </PlacedGraphic>
    </Layer>
    <Layer alias="houses">
      <!-- etc... -->
    </Layer>
</Scene>
```

Simplifying XML File Support

- First some definitions
- First-class concept
  - A concept represented in a well defined manner such as a class
- Second-class concept
  - Concepts that pervade system but not defined or consistently used

Second class concept example

```
class Person {
    string name;
    float weight; // in pounds
    float height;
    int age;
};
```

- Units are used second class concept
  - What are the units for each member data?
- Is there a programmatic way to determine the units?

Make it a first class concept

```
enum Units {
    Kilograms,
    Inches,
    Years
};
template <typename T> class UnitValue {
    UnitValue(T value, Units u);
    T value;
    Units units;
};

class Person {
    string name;
    UnitValue<float> weight;
    UnitValue<float> height;
    UnitValue<int> age;
};
```

Missing First Class Concepts

- Implementing missing first class concepts will improve code maintainability, robustness, and readability
- E.g., Time as a second class concept as a string – "9/10/08"
  - Code that uses this must parse the date
  - Must understand format
  - Must translate to other formats (such as German)

Improving design

- Adding more first class concepts often improves the design
- XML Document Object Model
  - First level XML concept
  - A tree structure of elements and attributes

The XML Element class

- An XML element contains a collection of element children (may be empty)
- An XML element contains a collection of attributes

The XML Attribute class

- An attribute contains a name and value