Lesson Five: Rendering the Scene

Lesson Objective

By the end of this lesson, you should be able to integrate the bitmap CODEC support from assignment 4 into the vector graphics system, and implement the new classes needed to render vector graphic scenes to a bitmap file.

Required Reading

The C++ Programming Language 4th edition: Chapter 21 (Class Hierarchies)

Rendering to a Bitmap File

At this point, we're ready to integrate our bitmap support into our vector graphics framework. This will allow us to see the fruits of our labor from now on by rendering to a bitmap file. Our framework needs to be neutral about where exactly it renders to—or at least, how the rendering surface is implemented. In earlier lessons, the actual drawing or rendering was considered a flattening of information. No matter what, drawing will involve a bitmap-like concept somewhere in the system, but it should not necessarily involve the bitmap file format. **Remember that the bitmap file format and general notion of a bitmap are distinctly separate.**

Our rendering will actually be a two-step process because of the need to work with a read-write surface before actually rendering. In other words, we'll want each graphic to draw itself onto a surface. Then we'll take that surface, iterate through it, and create a bitmap file from it, or put it on the screen if we have the libraries to support that. This practice is commonly called ***double buffering***. Generally, you wish to create complete images before updating the screen, printing, or whatever may be done with the final image. This prevents the user from seeing the drawing as it happens. It also is simply a necessity in many cases, as you can only print linearly; files work best linearly as well.

Key Term

- double buffering

It should be clear that we need to differentiate these two concepts: one as the surface where the drawing happens, and another that does the rendering of the finished drawing to its final destination (which might be decoupled and beyond the bounds of software, such as a piece of paper; this is why we can't model the destination, just the means to get the image to the destination).

For the first concept, we can use the non-technical term `Canvas` as an appropriate name for a work surface for drawing. For the second concept, we can use the term `Projector`, which emphasizes that we are projecting this finished image elsewhere—somewhere that might just not be part of the same system.

Note that we can use the idea of "drawing" for canvases and "rendering" for projectors. Drawing has no connotations of being scan-line at a time or linear in any way, whereas rendering is slightly more in the direction of what our projector will actually be doing.

Since a projector does an all-at-once rendering, it makes sense for it to receive a canvas with the request to project, rather than an iterator. Similarly, it wouldn't make very much sense for our projector to have much beyond this; the ability to set the colors of individual points doesn't work for an all-at-once rendering.

For both canvases and projectors, we'll need to allow a great deal of implementation independence, which will require an interface-based approach. This way, we can provide implementations of projectors that might print, generate a bitmap file, or render to the screen. For our canvases, we might have implementations that are optimized for different platforms or video hardware.

We should now have enough information to define our projector interface as follows.

Code Example 5.1

```
class IProjector
{
public:
    virtual void projectCanvas (HCanvas const& canvas) = 0;
};
```

Our canvas needs a bit more analysis, though. We know that the canvas will need to be able to provide an `IBitmapIterator`, since it will be storing flat, bitmapped information. But since the `IBitmapIterator` is a read-only iterator, we're still missing our actual means of writing to the canvas. We could use an iterator-based approach, but this is likely awkward. Generally, we want to draw to a canvas by actually saying at what point we want to set the color. We don't want to have to iterate through and keep track of the `x` and `y` ourselves until we get to the point we want, and then modify it. Instead, it would be preferable to have a higher level concept of setting the color at a given location, or getting the color at a given location. This will suit most of our drawing needs, whereas the `IBitmapIterator` will suit our final rendering needs. Our canvas interface is then as shown in example 5.2 below.

Code Example 5.2

```
class ICanvas
{
public:
    virtual void setPixelColor (Point const& location, Color const& color) = 0;
    virtual Color getPixelColor (Point const& location) const = 0;
    virtual int getWidth () const = 0;
    virtual int getHeight () const = 0;
    virtual HBitmapIterator createBitmapIterator () const = 0;
};
```

The addition of `getWidth` and `getHeight` operations should be self-explanatory: a canvas has to have a width and height. The reason the projector has no width and height is that it is taken from the canvas it is rendering.

It might not be clear why `setWidth` and `setHeight` should not be provided—and this decision is subjective. The semantics of changing the canvas's width and height are very ill-defined, so much so that it will likely be safer and more appropriate for clients to create a second canvas and copy to it in the way they wish. Otherwise, it is unclear whether `setWidth` and `setHeight` will stretch the existing image; or if they do not, how they will align the image on the canvas (upper-left or centered, for example).

Note that since we don't have a GUI client to drive our development, our design might be missing a few features. For example, if someone attempts to write to the canvas while it is still being projected (in a multi-threaded environment), they might get undesired results. Similarly, if someone attempted to create and work with more than one iterator to a decoder in the last assignment, a bug would most certainly be caused. These issues are important, but addressing them shouldn't change our design radically. We'll ignore many of these issues throughout this project, partially because we cannot support multi-threading and many important synchronization features that are platform-specific; but remember that, when you design a system, it is essential for you to have a deep understanding of your clients. You can read deeper into this advice by remembering that the term "clients," for us, means the applications that might sit on top of our vector graphics framework; but for the application, the term "clients" means the full spectrum of potential users. In the same way a vector-graphics application might fail for lack of research to determine what features graphic designers do and don't need, our vector-graphics framework could fail for lack of enough research about what the vector-graphics applications may and may not need.

Strokes

In Lesson One, when we looked at the properties of a vector graphic, we failed to get into much depth on the stroke and fill. As we said then, think of the ***stroke*** as the pen you use (and the way you use it) to draw the lines of a rectangle and ***fill*** its center with color, pattern, or texture. Because filling algorithms can be relatively complex and are dependent somewhat on line algorithms, we'll continue to hold off on fill support until the next lesson. We can't really put off the stroke, on the other hand; we'll need it to render a scene properly—otherwise our vector graphics (with no stroke and no fill) won't have much to draw.

Key Terms

- stroke
- fill

It might help to think about a few of the potential strokes you can create. At the simplest level, you can vary the shape of the pen tip, so to speak. The figures below show the same square rendered with different pen tips.

Figure 5.1-A—Rotated Square Tip

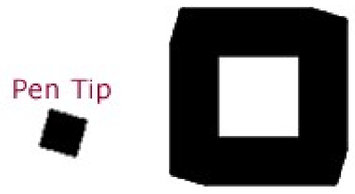
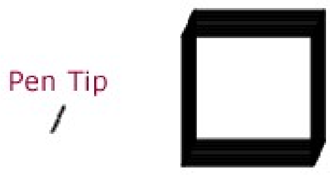


Figure 5.1-B—Forward Slash Tip



Not only can we vary the shape of the pen tip, but also its size and color. For the moment, we'll limit our support of strokes to these possibilities. Additional possibilities include pen pressure, texture (or pattern), and anti-aliasing, or softness.

Defining the Stroke and Pen Interfaces

Because strokes and pens will have a variety of implementations, we will use an interface-based approach again. Based on our analysis, our stroke interface should look as shown in example 5.3 below.

Code Example 5.3

```
class IStroke
{
public:
    virtual void setSize (int size) = 0;
    virtual int getSize () const = 0;
    virtual void setColor (Color const& color) = 0;
    virtual Color getColor () const = 0;
    virtual HPen createPen (HCanvas const& canvas) = 0;
};
```

Our pen interface will be more minimal, as defined in example 5.4 below.

Code Example 5.4

```
class IPen
{
public:
    virtual void drawPoint(Point const& point) = 0;
};
```

Locating Strokes

For the moment, we can hard-code in our parsing the creation of different strokes defined by name in the XML. In other words when we encounter a tip of "square," we can simply create a new instance of a known stroke class called `SquareStroke`. In the long run we would need a means of creating libraries of user-defined strokes and, mainly, a way to locate strokes based on their name.

In the XML, we can specify "stroke" for now as the required first child of the `VectorGraphic` element. We can define our stroke and its attributes as shown in example 5.5 below:

Code Example 5.5

```
<VectorGraphic closed="false">
    <Stroke tip="square" size="5" color="00FF00" />
    <Point x="0" y="0" />
    <Point x="100" y="100" />
</VectorGraphic>
```

Here, the color is in the form of hexadecimal RGB values, as is conventional in HTML and other technologies. The example color above would be green, based on `0xFF` (255 being the green value, and zero being the red and blue values).

Drawing the VectorGraphic

The drawing process would begin with a draw request to the `Scene` and flow eventually to each `VectorGraphic`. As discussed in the previous lessons, the vector graphics have relative coordinates, so their draw method, unlike the `Layer`'s and `Scene`'s draw method, would receive not only a canvas but a point describing their offset, as in example 5.6 below.

Code Example 5.6

```
void VectorGraphic::draw (Point const& upperLeftOrigin, HCanvas const& canvas);
```

Upon a draw request, the `VectorGraphic` can create a `Pen` from its stroke and then simply trace along its lines with the pen. So how does this tracing happen? First, one of the pluses of our stroke design is that it will work regardless of how the vector graphic might trace itself. If we added support for curves, no matter what algorithms we used to draw them, our pen approach would work perfectly. The pen makes it so that our actual drawing is highly independent of the algorithms we use to calculate which points we draw.

Of course, for the moment we only support straight lines, and up until this point we've only had to describe them geometrically in terms of their endpoints. During the process of the drawing, we determine each point we need to draw to generate the line between each end point. Since we need to reuse this code—and for conceptual simplicity—we can create a `LineIterator` class that allows us to iterate through each point in the line, and ask the pen to draw it. We can partially define this `LineIterator` class as shown in example 5.7 below.

Code Example 5.7

```
class LineIterator
{
public:
    LineIterator(Point const& beginPoint, Point const& endPoint);
    bool isEnd() const;
    Point getBeginPoint() const;
    Point getEndPoint() const;
    Point getCurrentPoint() const;
    void nextPoint();
};
```

There are a great variety of line algorithms, many of which will generate slightly different lines, and not all of which are useful in implementing our `LineIterator`. In assignment 5, you will need to either determine your own algorithm or formulae for calculating the next point or find one that will work via research online.

Beyond Fundamental Collections

In your XML parsing and your `Canvas` and `Projector` implementation, you will be (or have been) using a few as-yet undisguised new standard library collections. Unlike the fundamental collections we covered in Lesson One, these collections are not of universal value, but have discrete usages. Some of these collections have no implied underlying implementation (or data structure), but allow you to parameterize which fundamental collection to use for this purpose. We will avoid trying to categorizing these, though they might be best thought of as domain-specific collections. Instead, we'll look at each, how they are sometimes categorized, and what practical role they play in software development.

Queue, stack, and priority queue are often referred to as "collection adaptors" because they allow you to parameterize their underlying implementation. Each builds on top of the fundamental collections to implement its methods. This might be an unexpected design, since the fundamental collections have no well-defined underlying commonality. For this reason, the queue, stack, and priority queue use the lowest common denominator of collection features on the parameterized collection implementation.

Key Terms

- queue
- stack
- priority queue

Queue

A ***queue*** corresponds to the commonly-accepted notion of "queue" in the real world. Most service lines, as at a cafe cash register or an airline ticket counter, illustrate how a queue works: the first person who gets there is the first person who is served, and so forth. This idea is sometimes called "FIFO" for "First In, First Out."

As you will see in the stack and priority queue, clients of this loose group of collections are most concerned with insertion and removal, which are termed "push" and "pop" respectively in computer-science lingo. All that usually matters in a line is getting in it, and not getting out of it. Finding out where in the line you are can be helpful at times, but it's not the essence or purpose of the structure. The client of a queue doesn't care about who's in line, just who is next.

For this reason, a queue is particularly useful when we want to decouple the insertion and removal of elements while keeping them in the order in which they were inserted. This focuses us less on the elements in the collection and more on the queue itself. A good example of using a queue is the systems that maintain a "who's next" kind of list. Of course, it can be straining to try to think of a realistic example for this, as it's not an entirely common task (nor are the tasks we perform with a stack or priority queue). One example might be a queue of incoming connections to a game server. In this case, our queue might have a maximum waiting length. When someone connects and the queue is full, they simply must try again; but a limited number of people can remain in the queue until one of the current players leaves.

Stacks

There aren't many situations in our current system that could benefit from the use of a queue, but there is one that could use a ***stack***. One thing we didn't mention during the last lesson is that the bitmap file format stores scan-lines from bottom to top, not top to bottom as you might expect. This wasn't an issue in the last assignment because the effects we applied would work the same even if the image was upside down (which it was while we were applying them). Now that we are creating our own scene, however, rather than reading in an image and modifying it, we'll certainly notice which way should be up and which way should be down.

To fix this in the one place that currently requires it—the bitmap file projector you will implement in the assignment—we could use a stack. Once again, a stack works just like those situations that use the term "stack" in the real-world. With a stack of plates, for example, you take off the last plate you put on when you remove the next available plate. This is why stacks are referred to as "LIFO" for "Last In, First Out." Stacks are often used as a quick-and-simple means of reversing for this reason. In our case, if we put each scan line we read from the `IBitmapIterator` onto a stack and then proceed to write them to the bitmap file in the order we remove them, we will effectively reverse the scan line order.

Priority Queue

Continuing with the airline ticketing counter example for a queue, let's say business-class customers get precedence over coach customers, who get precedence over standby customers. Each group still gets first-come-first-serve priority, but only at their given priority level. This behavior is really not any different from creating three queues: one for business class, one for coach, and one for standby, and calling from each line first. If anything, in the real world, people would prefer separate lines to being asked "Are you business class or coach?" when they get to the front. This is why a ***priority queue*** would be one way of implementing this, but it might not be the cleanest.

Priority queues work via comparators. Each element is compared against all other elements and placed in order. If two elements are equal, then they are in order of insertion. So, in this case, we'd have to create a comparator for our airline customers. There benefit of using the priority queue over three separate queues is that of ignorance: our code can be oblivious to who inserted and what the sorting criteria are and instead just say "Get me the next customer."

Set

The ***set*** provided by C++ is similar to the idea of a set in mathematics; but it does provide some operations that do not co-align with the math world's idea of a set. The one particularly useful thing the set does provide is an efficient comparator-based `find` method. This is useful only for classes that behave like primitive types (and sometimes, only for classes that are unsafe). An example is in order.

Key Term

- set

Consider our list of attributes in the `XML::Element` class. When we need to look up a attribute's value in the interface, we'd like to efficiently locate which attribute we want based on its name. It would be nice if there were a simple, efficient way in the C++ standard library to optimize searching on a per-order basis on a variety of attributes or "fields" of classes. The problem is, the `find` method always takes an object of the `Element`'s type. So, in our case, when we do our lookup for the right attribute, we need to pass in a fully operational instance of the `Attribute` class. Since we only care to search by name, we have to construct a "dummy" attribute instance with an invalid value. We also have to define a comparator that ignores the values of attributes and instead only considers equality based on the name. The latter requirement is not unreasonable, but constructing a "dummy" object is never a good idea. Also, in many cases, we **can't** just create a "dummy" object. If we wanted to store a set of decoder or encoders, with the MIME type being the key for the comparator, we'd be out of luck. To call the `find` method, we'd need a handle to a decoder instance that had the same MIME type. The only way to do this would not be to just make a "dummy" instance, but a "dummy" class—an implementation of `IDecoder` that simply allowed us to set the MIME type and did nothing for the rest of its operations. Clearly, this would not be a good idea.

Our list of attributes is reasonable enough to change to a set and optimize the `find`, but you should exercise caution in using sets. Often, a map might be a better alternative.

There are a few features of a set that aren't obvious from our conversation. These are as follows.

- A set does not allow duplicate elements (elements that are equal, based on the comparator supplied or the `less-than` operator). This is one of its major features, although multiset (covered next), another "type" of set, does allow duplicate elements. This violates the mathematical definition of a set.
- All elements are automatically sorted on insertion.
- Lexicographical comparison operators (see the reading assignment for details) are used in very specific domains and are not of general value.

The set is not easily categorizable. Traditional texts consider it an "associative collection," since its implementation is similar to these (it is always implemented as a binary tree, similar to the map) and it provides an optimized `find` method.

Multiset

A ***multiset*** allows multiple instances of the same element (where similarity is determined based on the comparator). Therefore, it's really rather close to a list in behavior, with the exception of automatic sorting. It provides all the operations a set does, plus the following.

Key Terms

- multiset
- map
- multimap

- Counting of equal elements. In other words, you can see how many instances of the same element there are.
- A `lower_bound` method that returns the first position where an element might be inserted.
- An `upper_bound` method that returns the last position where an element might be inserted.
- An `equal_range` method that returns a range (as pair) where an element might be inserted. This, like the `lower_bound` and `upper_bound` method, is introduced because, when you insert an element that is equal to existing elements, it might be placed anywhere in that range.

Also, make note that a multiset's `find` method returns the first element, since there may be more than one.

Map

Most object-oriented systems do not rely heavily on the idea of "mapping" or "lookup." Since we don't need to use arbitrary tokens for information in object-oriented systems, maps don't provide the same value that they may once have. Continuing with our windows example, the more modern, object-oriented, form of windows programming simply uses a `window` object that has all its own attributes directly and allows navigability to other important information.

Maps in object-oriented systems usually are for one of the following specific purposes (though there are more, these are the most common). In Windows API there are "handles" to windows—a token with which you perform operations, which takes the place of an object in procedural languages. In UNIX, a file descriptor is a similar concept.

- When you need to map from one reality to another, a map is a natural solution. Although you can still create objects that would provide the equivalent, sometimes there's no clear concept that bridges the two worlds. For example, if you were mapping a tag name in XML (one reality) to the actual object that the tag represented (another reality), you might use a map of prototypes, as you did in *C++: Intermediate*. This could not be done in this course's project because of the reasons we mentioned in Lesson One.
- When you need to optimize searching on a particular field, a set can be used, although, as we discussed, a map is often more straightforward. The downside to using the map is that we separate the field from the class in some ways. In other words, if we store attributes as a map, we have a second instance of the name, which is detached from our cohesive `Attribute` class and will cause a bit of confusion, especially if the names manage to become out of sync (easy to do). All in all, this can be why maps and sets are sometimes less preferable than a list. Sets and maps are best for objects that behave like primitive types. If we use a reference object in a map or set, we always run the risk that the reference object could change independently; for example, imagine if a decoder's MIME type changed dynamically. Although this situation is a bit contrived (it's the only map to a reference object that we have in our system), you can see how similar situations could cause problems when using a map or set and an optimized `find` with reference objects.

Although it is often tempting, try to avoid using a map to relate two disparate objects. Generally, if you need to do this, it should be a hint that you need a new first-class concept in your model. For example, consider a command console in which there are strings that represent commands and *functors* (objects, like comparators, that behave like a function) for the action that should be taken. Although a map could be used to relate these, having a set of `Command` objects would be more appropriate. Clearly, there is an idea of a "Command" in this model that would have the command's name as well as its action, and maybe help information and other important attributes. For our basic canvas implementation in assignment 6, we could use a map of `Point` to `Color` instead of a map or set of `Pixels` (which might have a location and color attribute) in our basic canvas implementation. Similarly, we could have (and may have initially) used a map of string to string for our attributes in the `XML::Element` class.

A map relates two concepts: a "key" and a "value." These can be (and usually are) separate types, as in our examples above. The optimized `find` method in a map searches based on key. The map's iterators are unique in that their underlying element is actually an `std::pair<key, value>`, since you might need either.

The map also provides an array indexing operator, which allows simplified semantics of lookup, but should be used with caution. When we set **or** get a value via the indexing operator, if the key that's used as the index does not already exist, it will be created. This is why you should avoid ever using the indexing operator for looking up values, but instead use the `find` method to do so.

There are a couple of important notes about maps that might not be obvious from their interface:

- You may not change the key of an element because it will disturb ordering; you have to remove it and reinsert.
- For this reason, when iterating, the key will be constant.

Multimap

A ***multimap*** is mostly identical to a map, with one difference: multiple values per key. The uses for multimaps are few and far between. Consult your texts for information on multimaps if you find yourself in a situation that merits using them.