CP713 Lesson 5

Rendering the Scene

Lesson Objective

- Integrate the bitmap CODEC from assignment 4 into the vector graphics system
- Implement the new classes required to render vector graphic scenes to a bitmap file

Rendering to a bitmap file

- We’re ready to integrate bitmap support into our vector graphics framework
- Our framework will be neutral about where it renders to
- Rendering involves converting vectors to bitmap data
    - Does not necessarily involve the bitmap file format
- Remember that bitmap file format and general notion of bitmap are distinctly separate

Rendering

- Rendering will be a two step process
  - First, each graphic will draw itself to a surface
  - Then, we'll iterate through the surface and create a bitmap file
    - Or put it on the screen if we have the libraries to support that
- This practice is called double buffering
  - Create complete images before updating screen
  - Then render to screen
  - This prevents user from seeing drawing as it occurs

Differentiating two drawing concepts

- Concept 1: Surface where drawing happens
  - We'll use the name Canvas for Concept 1
  - We'll ***draw*** on a canvas
- Concept II: Rendering drawing to final destination
  - Such as bitmap file, or screen
  - We'll use the name Projector for Concept II
  - We'll ***render*** to a projector

Interface approach

- We'll need to allow implementation independence
  - E.g., We might project to a bitmap file, a screen, or to a printer
  - For canvases we might optimize for different screens or video hardware
- To allow implementation independence we'll rely on interfaces

IProjecter interface

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

Canvas Interface Design I

- Canvas must provide an IBitmapIterator
  - It will be storing bitmapped information
- IBitmapIterator is read-only
  - Can read from canvas
  - But, no means to write to the canvas
- We could use iterator approach to write to canvas
  - It's awkward though
  - We don't want to iterate through and keep track of x and y ourselves

Canvas Interface Design II

- It would be preferable to have a higher level color setting concept
  - Get the color from a given x, y location
  - Set the color from a given x, y location
  - This will satisfy our drawing needs
- IBitmapIterator will suit our rendering needs

ICanvas

```
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;
};
```

ICanvas Discussion

- Canvas has width and height
  - Thus getWidth() and getHeight()
- Projector has no width or height
  - It takes size from canvas it is rendering
- Why no setWidth and setHeight
  - We're not defining set semantics for this assignment
    - E. g., would graphics stretch or reposition on width/height change?
  - Clients must create a second canvas and copy to change width and height
    - Ideally we would make this part of canvas, but we'll skip as there is enough to do already

What's missing from our design?

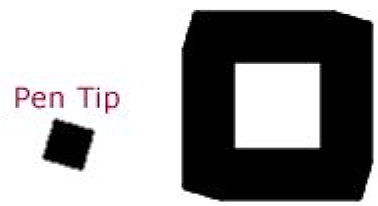
- If someone writes to canvas while it is being projected may get undesired results
  - E.g, in a multi-threaded environment
- If more than one iterator is created and used a bug will be caused
- Lesson – when designing a system you must have understanding of clients needs
  - May build wrong thing
  - May make client work too hard
  - May build unneeded capability

Strokes 1

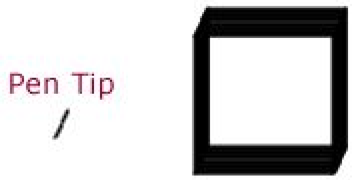
- Imaging drawing a rectangle
  - Stroke is the pen used to draw the rectangle lines
  - Fill is used to draw the rectangle center with color, pattern, or texture.
    - We'll hold off on fill until next lesson

Strokes II

- Example strokes (vary pen tip)



Rotated Square Tip



Forward Slash Tip

- We can also vary tip size and color
- We'll constrain ourselves the above stroke possibilities
- Future possibilities include
  - Pen pressure, texture, softness

Defining Stroke and Pen Interfaces

- Because strokes and pens will have a variety of implementations we'll use interface approach (again)

```
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;
};
```

IPen

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

Locating Strokes

- We'll create some known stroke classes
  - Such as SquareStroke
- We'll specify stroke in VectorGraphic XML

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

Drawing the VectorGraphic I

- Drawing request begins with Scene and flows eventually to VectorGraphic
- Remember that a VectorGraphic has relative coordinates
  - Thus its draw method provides an offset point

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

Drawing the VectorGraphic II

- Upon a draw request, VectorGraphic can create a Pen from its Stroke and trace along its lines
- Note that the pen concept makes it so drawing is independent of algorithms used to calculate which points to draw
- For now, we'll only support straight lines

Rasterize

- Rasterize: The process of converting a vector image into a bitmap image
- What's it mean to us?
  - During the process of the drawing, we determine each point we need to draw to generate the line between each end point

LineIterator Design

- To reuse rasterization code we'll create a LineIterator
  - Iterates through each point in a line and asks pen to draw it
- There are many line algorithms
  - Each generates slightly different lines
- In assignment 5
  - Determine your own line algorithm
  - Or find a line algorithm online

LineIterator

```
class LineIterator {
public:
    LineIterator (
        Point const& beginPoint,
        Point const& endPoint);

    bool isEnd () const;

    Point getBeginPoint () const;
    Point getEndPoint () const;

    Point getCurrentPoint () const;
    void nextPoint ();
};
```