UCI University of California, Irvine

Informatics 134

Software User Interfaces
Spring 2021

Mark S. Baldwin
baldwinm@ics.uci.edu
4/27/2021

Special Thanks: Brad Myers at CMU for his amazing lectures

Agenda

1. This Week

2. Basic Structured Graphics

3. Application of Structured Graphics

This Week

This Week

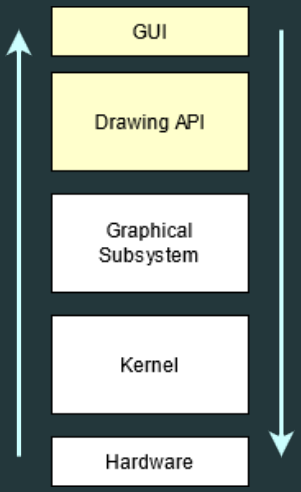
- Lecture on Thursday
- Launch T3 on Thursday
- Keep working on A3 (DUE ~~5/10~~ 5/18)

Basic Structured Graphics

Basic Structured Graphics

Requirements of a Graphical Program

- Manage what gets rendered to a screen
- Manage how it gets rendered
- Manage when it gets rendered



Requirements of a Graphical Program

As programmers of graphical user interfaces, we primarily concern ourselves with *what* is rendered, rather than *how* or *when*.

Any thoughts on why?

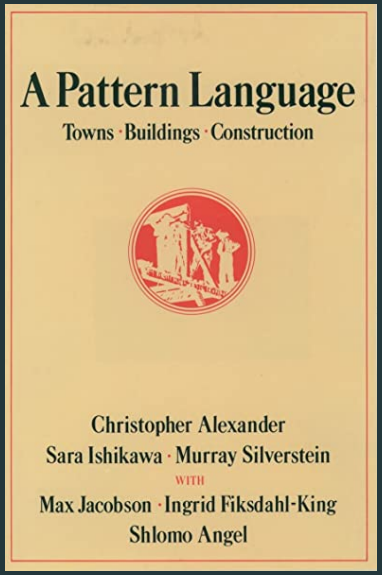
Requirements of a Graphical Program

The *how* and *when* are largely repeatable tasks that do not change across different user interfaces.

The *how* and *when* requirements, therefore, can be abstracted into reusable mechanisms that can support the *what* that programmers create.

This type of system is called ”Structured Graphics”

Basic Structured Graphics



Structured Graphics

Encapsulate a primitive (rectangles, lines, images, icons, etc.)

Expose reusable code for rendering how and when.

Enable programmer to create the what (e.g., a button)

Advantages of Structured Graphics

- Less code, more reusable
- Encapsulation of common mechanisms enables automation of required actions like redraw and refresh
- Hierarchical model supports custom encapsulation as well

Some Trade-offs

- Supporting reuse increases memory consumption
- Redraw and refresh can take more time
- Combined, can effect ’snappiness’ of UI

Though modern computing power negates most of these concerns

Redraw and Refresh Operations

Operation depends on underlying algorithm (the how and when)

- One approach is to redraw every object every time a change occurs to any graphical object. Trade-offs?
- Draws all objects in the hierarchy from back to front (from a display perspective), top down hierarchically

Redraw and Refresh Operations

Operation depends on underlying algorithm (the how and when)

Another approach is to only redraw the area of the display that has changed.
Trade-offs?

Capture all objects that intersect the area to be redrawn, redraw from back to front.

The Document Object Model

The DOM used in web browsers is an example of structured graphics.

- Maintains a hierarchical list, or ”retained object model” of all graphical objects.
  Update the screen by editing objects in the list.
- When added to an HTML page, SVG becomes part of DOM object model.

Basic Structured Graphics

The Hierarchical List

Graphical Primitives

- text, icons, and shapes

Aggregates

- collections of graphical objects
- “‘div”‘ or ’Groups’ in SVG
- Parent/child relationship

The Hierarchical List

Hierarchies are built through aggregate object types and inheritance.

One Example: CSS.

```css
body {
    color: green;
}
.my-class-1 a {
    color: inherit;
}
.my-class-2 a {
    color: initial;
}
.my-class-3 a {
    color: unset;
}
```

```html
<ul>
    <li>Default <a href="#">link</a> color</li>
    <li class="my-class-1">Inherit the <a href="#">link</a> color</li>
    <li class="my-class-2">Reset the <a href="#">link</a> color</li>
    <li class="my-class-3">Unset the <a href="#">link</a> color</li>
</ul>
```

[Mozilla, 2021]

The Hierarchical List

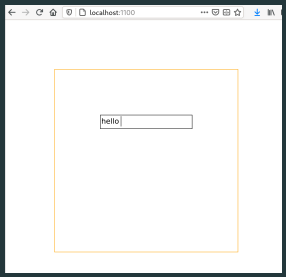
Issues and Design Considerations

- Complexity increases with features
- Which point in the hierarchy has responsibility for a given property
- Which hierarchy is responsible for themes? events propagation?
- Should objects change in appearance based on type or should each type be a new object?

Application of Structured Graphics

Hierarchy in SVG

Let’s start with a look at hierarchy in raw SVG using SVG.js




```
1  import {SVG} from './svg.min.js';
2
3  SVG.on(document, 'DOMContentLoaded', function(){
4      var draw = SVG().addTo('body').size('1000px','1000px');
5      var window = draw.group();
6      window.rect(400,400).stroke("orange").fill("white");
7
8      var group = draw.group();
9      var rect = group.rect(200, 30).fill("white").stroke("black");
10     var text = group.text("hello").move(2,4);
11     var caret = group.line(45, 2.5, 45, 25).
12         stroke({ width: 1, color: "black" });
13
14     group.move(100,100);
15
16     window.add(group);
17     window.move(100,100);
18 });
```

SVG and the DOM

Which language?

- Javascript or Typescript?
- Typescript is built on strong types and object oriented principles
- Javascript does not require transpiling

A Javascript Example

Adapted from A3 assignment description

```
var Button = function(){
    var draw = SVG().addTo('body').size('100%','100%');
    var group = draw.group();
    var rect = group.rect(200, 30).fill("white").
        stroke("black");
    var text = group.text("").move(2,4);
    var caret = group.line(45, 2.5, 45, 25).
        stroke({ width: 1, color: "black" }
    return {
        move: function(x, y) {
            group.move(x, y);
        },
        text: function(t) {
            text.text(t);
        }
    }
}
```

Demo

Application of Structured Graphics

SVG and the DOM

Object Oriented Model

- Already popular with GUI toolkits
- Conceptually similar to primitives and aggregates
- Easier (IMO) to reason about how a hierarchy should be structured.

Object Oriented Hierarchy in Typescript

- Abstraction
- Class == graphical object
- Instances
- Inheritance

```
interface IWidgetEvent{
    ...
}

class Window implements IWidgetEvent {
    private _objects: Widget[];

    constructor(height:any, width:any){
        ...
    }
    public addWidget(widget:Widget){
        this._objects.push(widget);
    }
}
```

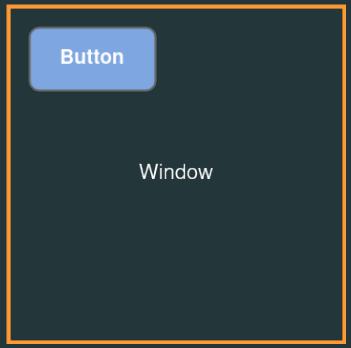
Object Oriented Hierarchy in Typescript

- Abstraction
- Class == graphical object
- Instances
- Inheritance

```
abstract class Widget implements IWidgetEvent {
    private _backcolor: string;

    constructor(height:any, width:any){
        ...
        this._backcolor = "black";
    }
    get backColor(): string{
        return this._backcolor;
    }
    set backColor(color:string){
        this._backcolor = color;
    }
}

class Button extends Widget{
    constructor(){
        super(100, 50);
        ...
    }
}
```




```
let w = new Window(500,500);
let btn = new Button();
btn.backColor = "blue";
w.addWidget(btn);
```

Application of Structured Graphics

Object Oriented Model

Already, you can see how decisions need to be made.

- Design and optimize classes and interfaces to avoid duplicate code (costly space/performance)
- Encapsulate when possible (the Button class should not need to worry about backcolor change)
- Sensible names and parameters

Any Questions Yet?

References

References i

Mozilla (2021).
Cascade and inheritance.