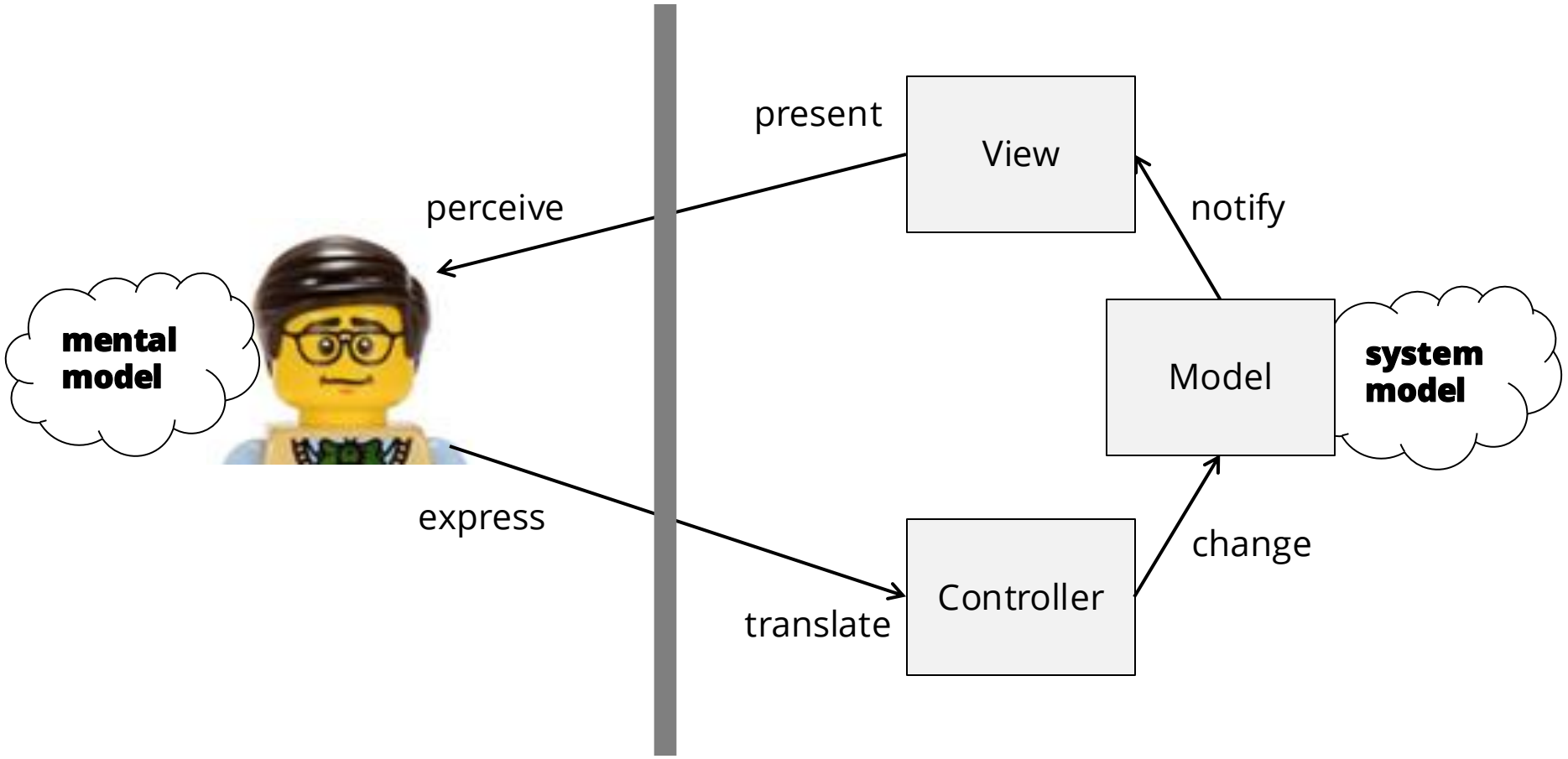
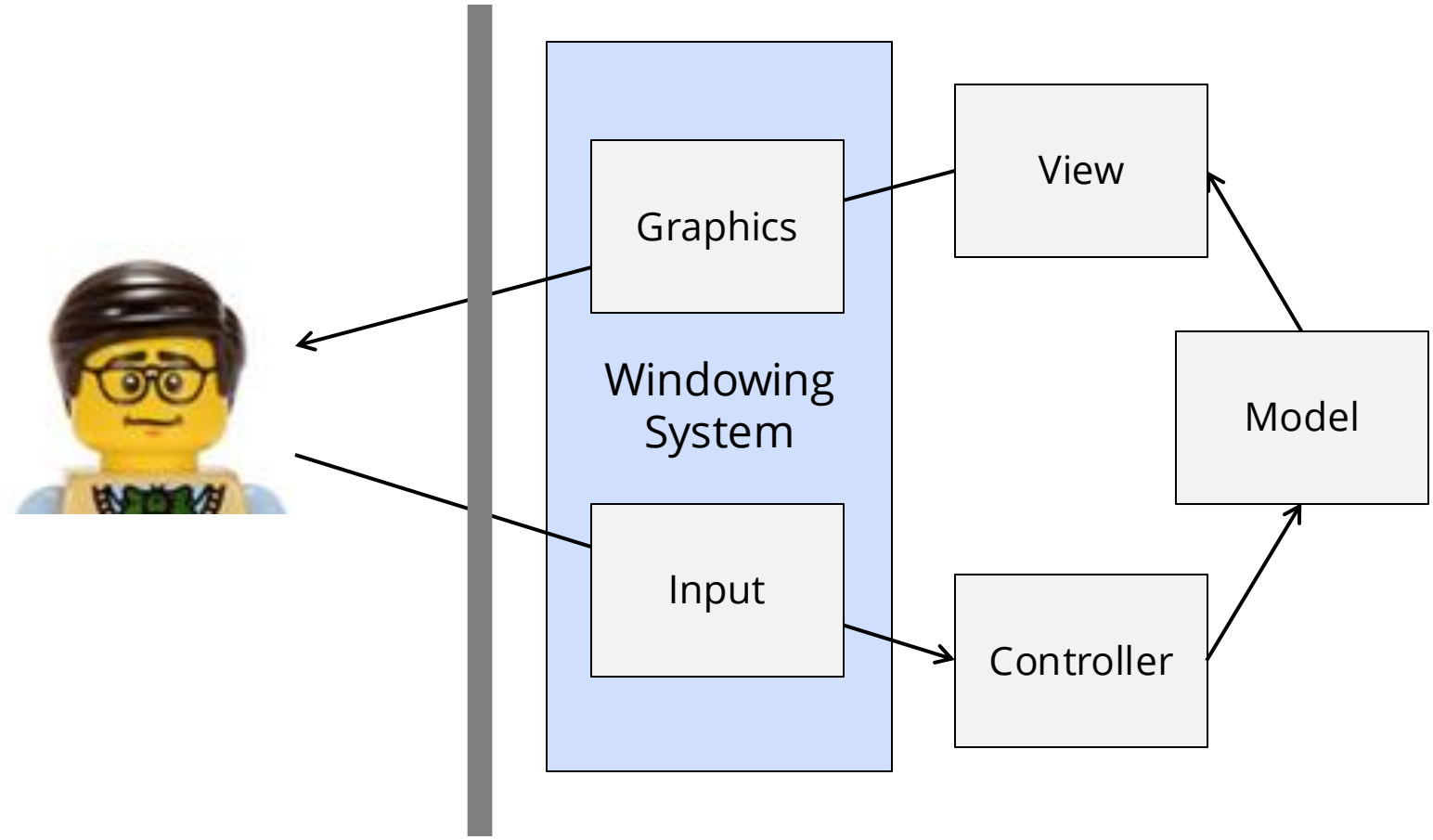
Drawing

- Drawing models
- SimpleKit
- Graphics context
- Drawable Object
- Painter's Algorithm
- Display List

Model-View-Controller (MVC)

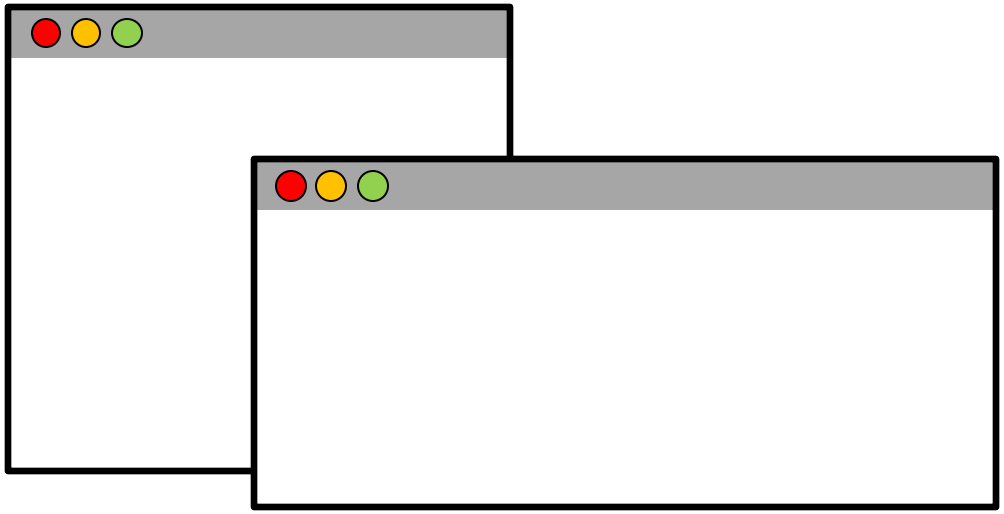


Graphical Presentation Architecture



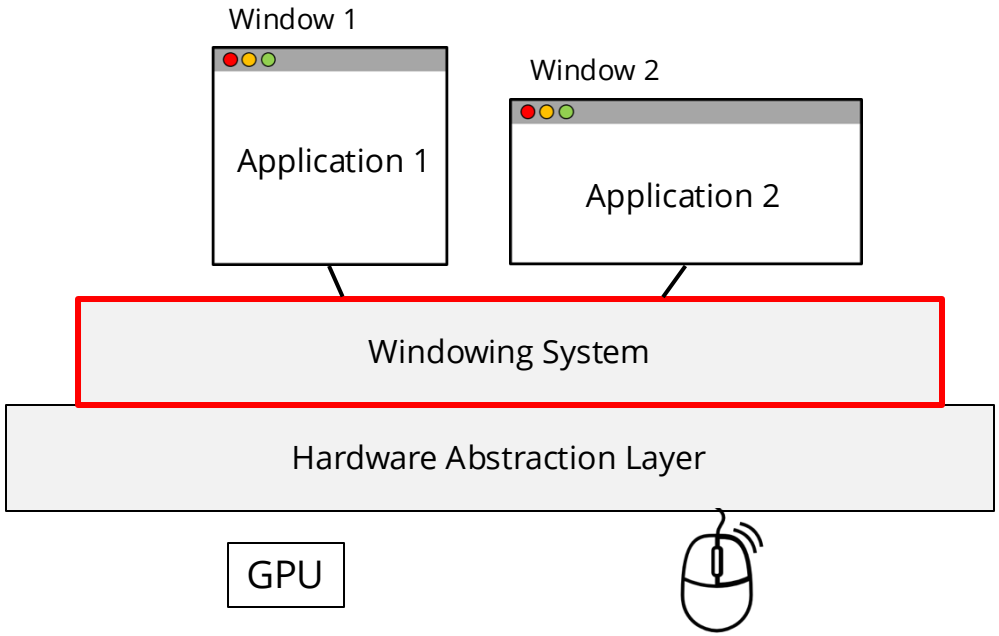
Windows

- Windows are a visual and architectural organizing principle
  - Each window has a location (or is hidden/off-screen)
  - Each window has a size
  - Each window has a "depth" (e.g. stacking order)
  - Only one window has "focus" to receive user input
  - Each window is associated with an application
- Applications running in windows can be isolated
  - Each has its own memory, resources, and **drawing canvas**



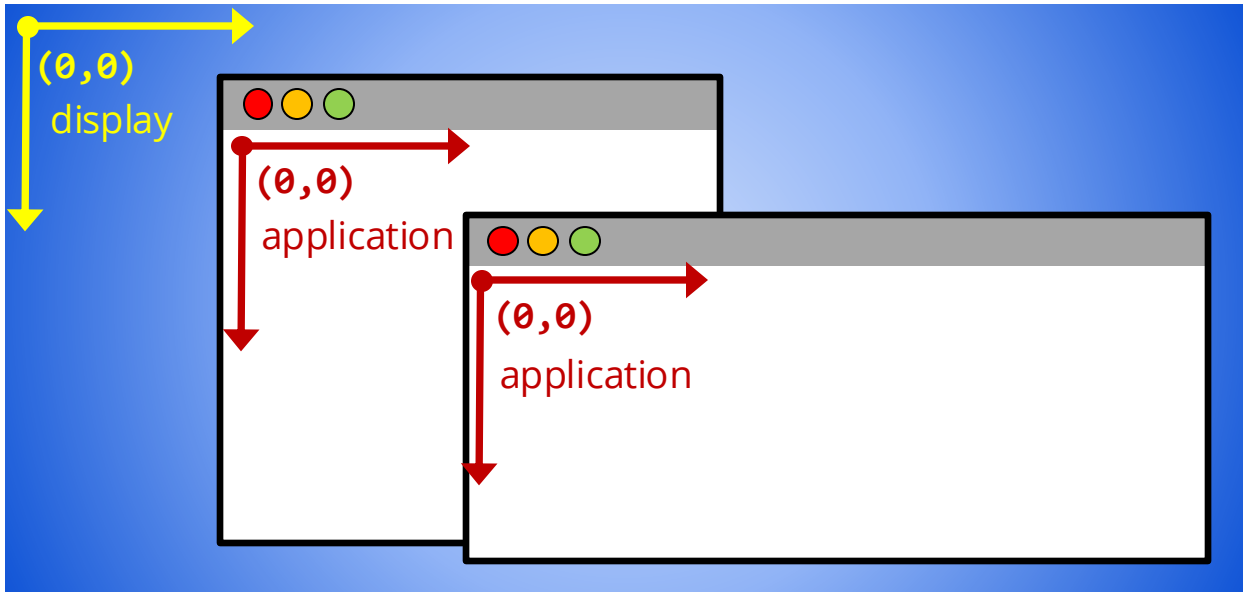
Windowing System

- The **windowing system** is an operating system layer to share screen space and user input among applications
- Provides three main services:
  1. Manage list of windows: creating, resizing, focusing, etc.
  2. Provide each application with an independent drawing area
  3. Dispatch low-level input events to the focused window



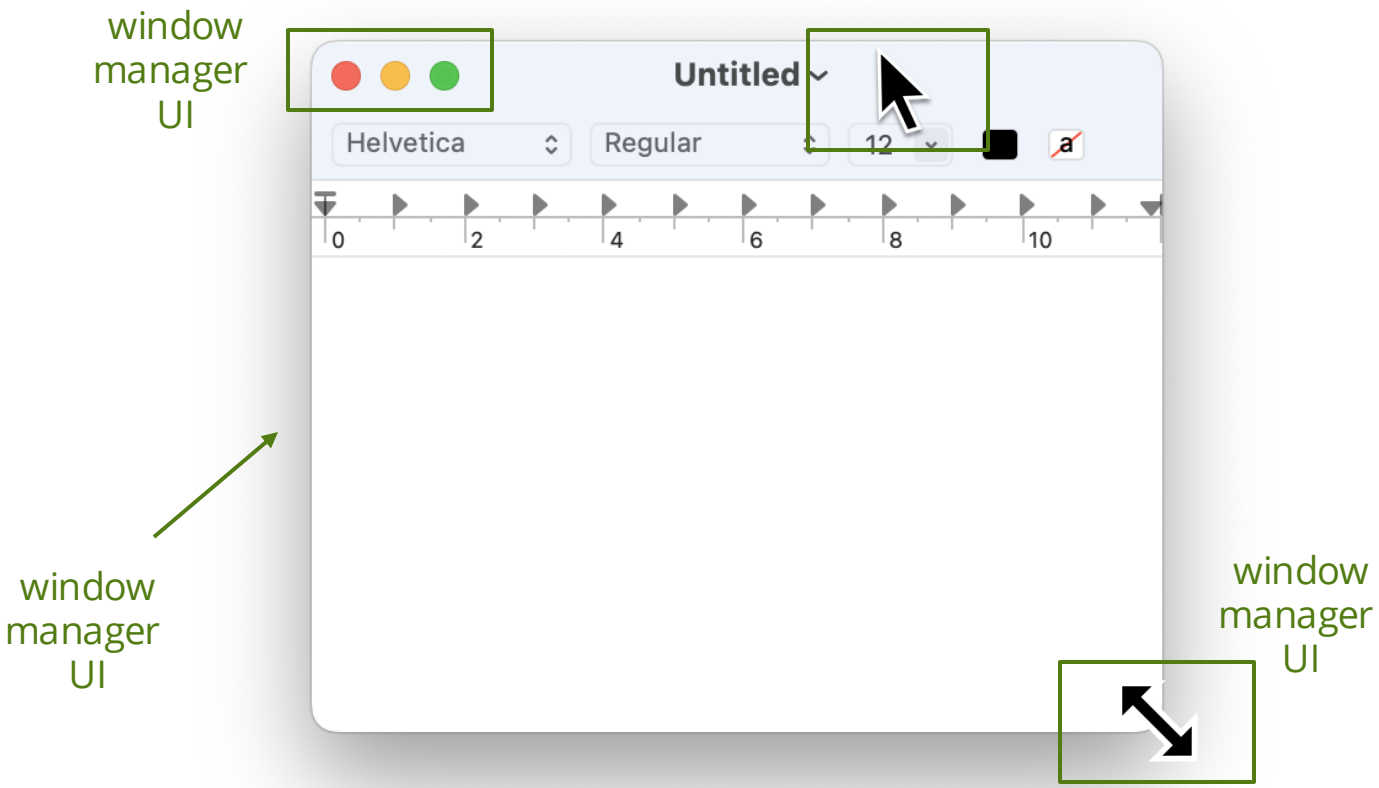
Canvas Abstraction

- The windowing system uses a **drawing canvas abstraction**
  - Each application has a defined drawing area within the window
  - The drawing area has its own local coordinate system (due to historical convention [0, 0] is at top-left)
  - The drawing area is typically implemented as a graphics buffer
- The windowing system renders the buffer at the window position
  - uses very fast method called **bitblt** (**bit bl**ock **t**ransfer)



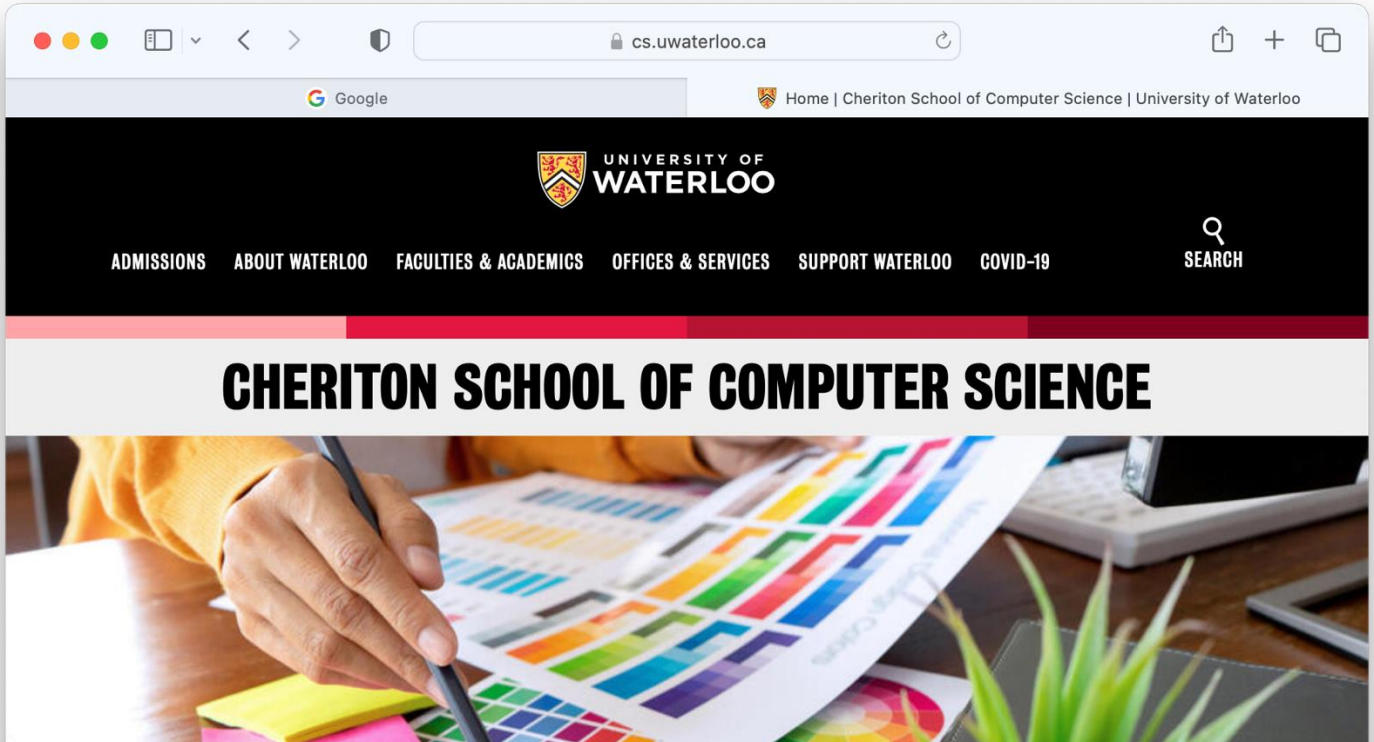
Window Manager

- The windowing system also has a **window manager** to render the window "chrome" and provide a window UI
  - buttons for closing, minimizing, maximizing window
  - draggable areas for resizing and moving window
  - rendering "look & feel"



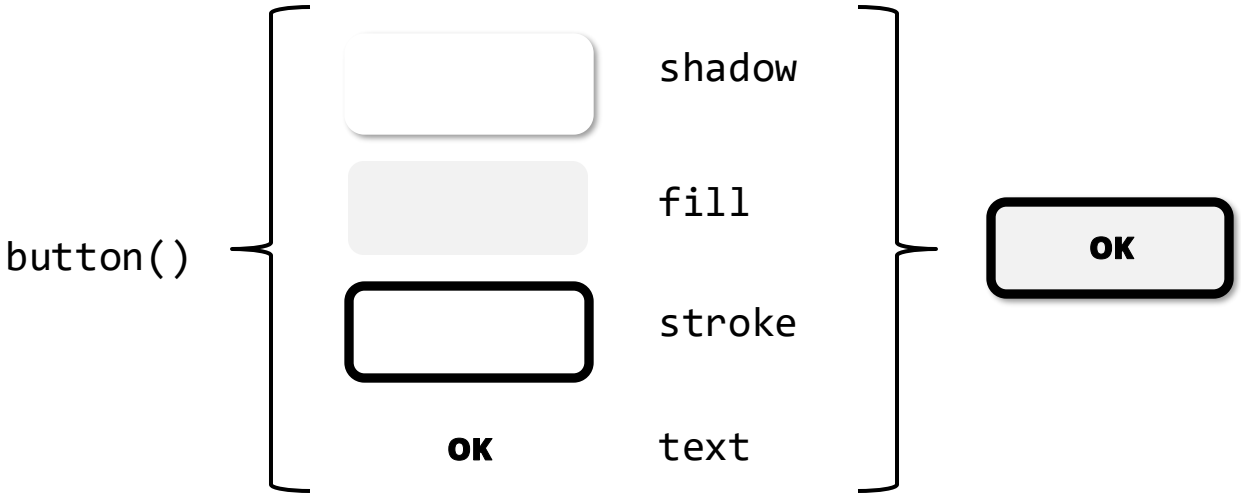
Browser as Windowing System

- A modern web browser is like a windowing system
  - it manages a list of tabs: creating, focusing, etc.
  - it provides each tab with an independent drawing area
  - it dispatches events to the focused tab
  - the browser interface (enter a URL, back button, refresh, etc.) is like an expanded window manager interface



Drawing and User Interface Toolkits

- A graphical user interface is essentially a drawing of shapes
  - rectangles, lines, text, fills, etc.
- User interaction is essentially how these shapes change
  - responding to user input, animation, or external data
- A **UI toolkit** provides a level of abstraction for programmers
  - e.g. translates programmer's concept of a "button" into shapes that represent a rendering of a button



We'll start by only drawing shapes ***without*** a UI toolkit

Drawing Primitives

- Three conceptual models for drawing:

Pixel

```
SetPixel(x, y, colour)
```

Stroke

```
DrawLine(x1, y1, x2, y2)
```

Region

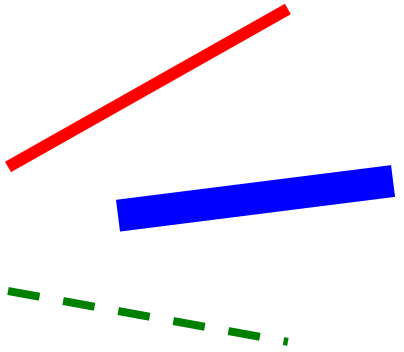
```
DrawRect(x, y, w, h)
```

Drawing Style

Consider `DrawLine( … )`

- what style options are there?

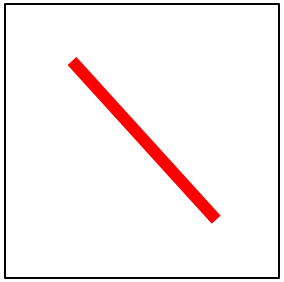
- Observations:
  - most choices are the same for multiple calls to `DrawLine()`
  - lots of different parameters, but may only want to set one or two



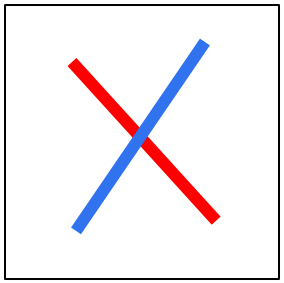
Graphics Context

- A common approach to manage state of drawing style options
- A drawing command like `DrawLine(x1, y1, x2, y2)` is rendered using the current state of style options

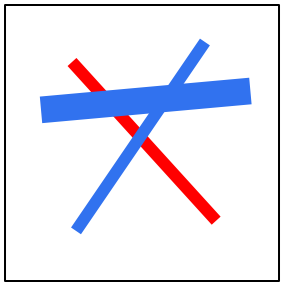
```
Stroke(RED)
StrokeThickness(5)
DrawLine( … )
```



```
Stroke(BLUE)
DrawLine( … )
```



```
StrokeThickness(10)
DrawLine( … )
```



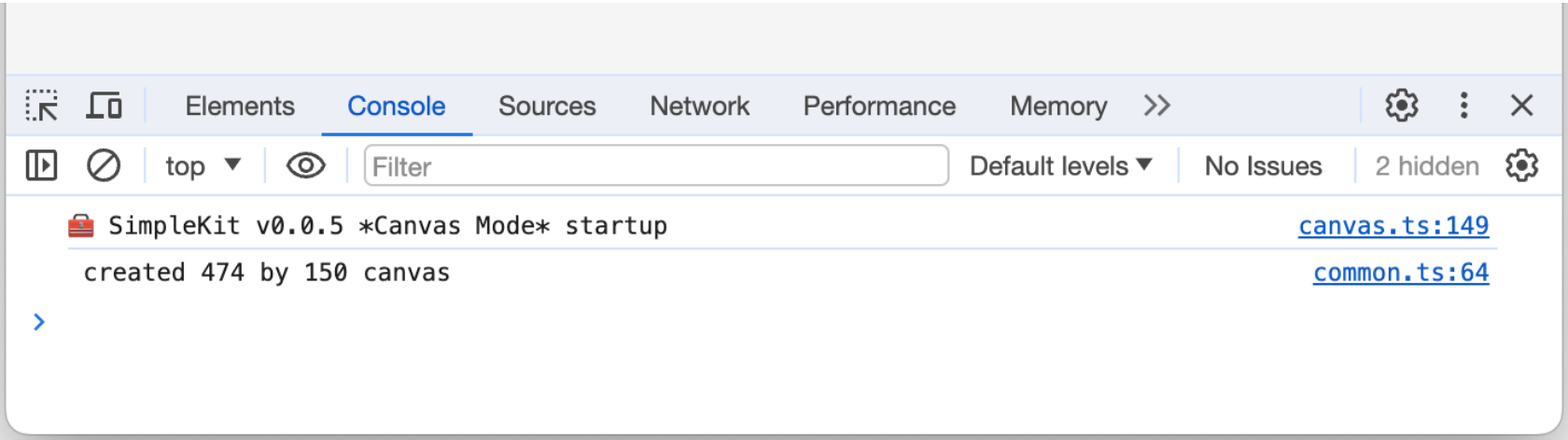
html-canvas

- HTML canvas (`HTMLCanvasElement`) is a literal "canvas abstraction"
- Can create with a <canvas> tag
- We'll always create it in code
  - Compare index.html to "Elements" in dev console
  - Using createElement and appendChild
  - Using getContext
  - type narrowing
- CanvasRenderingContext2D
  - Drawing commands
  - Set drawing style

SimpleKit

- We're using SimpleKit for first part of course (incl. A1 and A2)
  - Simulates a windowing system and other UI layers in browser
  - Different toolkit modes (e.g. canvas-mode, imperative-mode)
  - By design, it's somewhat incomplete and very limited
  - We'll examine how it's built to illustrate UI architecture
- The demo repo includes SimpleKit as a git submodule
  - See README for cloning and updating instructions
  - Vite projects for demos use the simplekit in the submodule

You'll use SimpleKit as an **npm package** in your assignments



simplekit-canvas

Most basic usage for SimpleKit “canvas mode”:

1. Import what you need

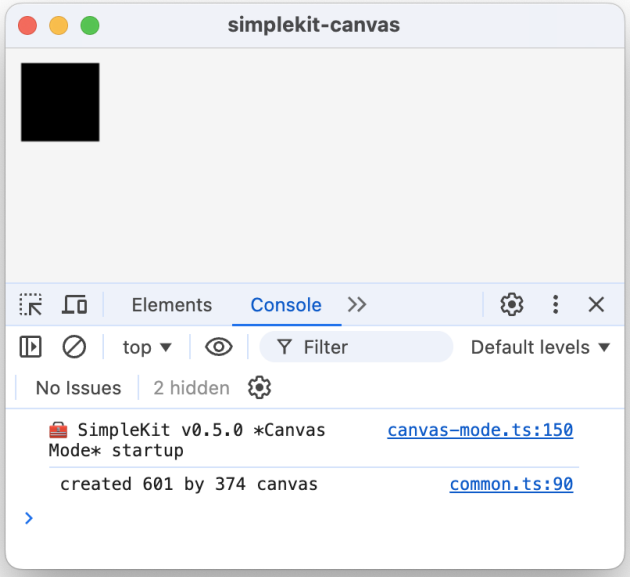
```
import { startSimpleKit, setSKDrawCallback }
  from "simplekit/canvas-mode";
```

2. Start it up (creates full page canvas, etc.)

```
startSimpleKit();
```

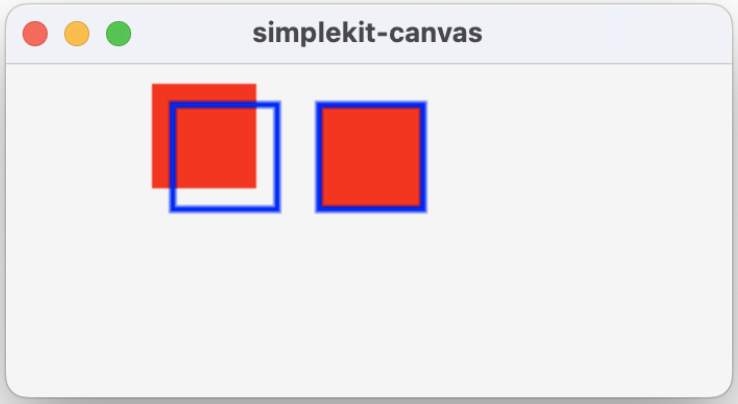
3. Set the drawing callback

```
setSKDrawCallback((gc) => {
  gc.fillRect(10, 10, 50, 50);
});
```



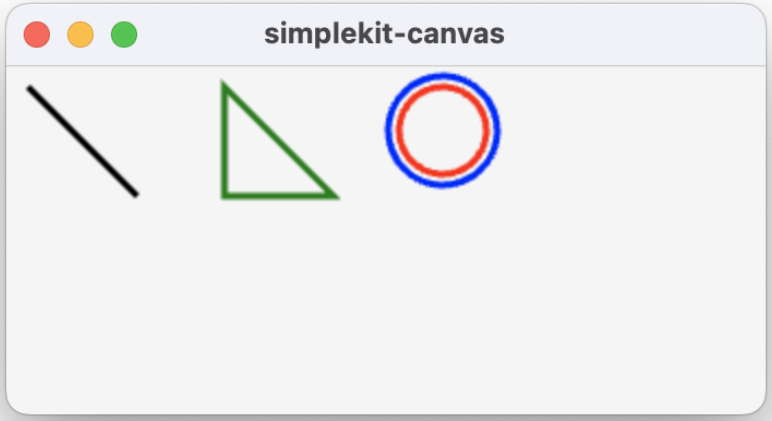
simplekit-canvas rectangleDemo()

- Different drawing orders
- What happens when gc changes state at end?



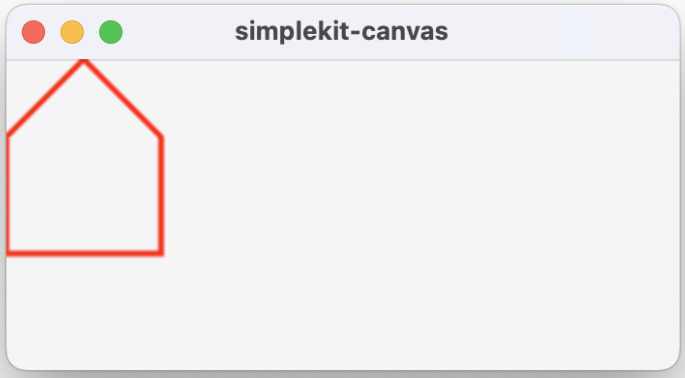
simplekit-canvas pathDemo()

- Draw line
  - moveTo vs lineTo
- Draw polyline or polygon
  - closePath
- Draw circle
  - Using arc
  - Using ellipse
- Draw rect “path”
  - fill then stroke



simplekit-canvas pathHouseDemo()

- Drawing from list of points
- How to position the shape?
- (TypeScript note) type for array of 2D points



simplekit-canvas textDemo()

- Setting font size (requires font name or type)
- Vertical and horizontal alignment also a gc state change

Specifying Colour

- `fillStyle` and `strokeStyle` properties use **CSS color syntax**
- **Named** colour (more than 100)

  `"red"`, `"blue"`, `"cornflowerblue"`, `"deeppink"`
- **Hexadecimal** colour as `#RGB`, `#RRGGBB`

  `"#f00"`, `"#0000ff"`, `"#6495ed"`, `"#ff1493"`
- **RGB**: Red, Green, Blue

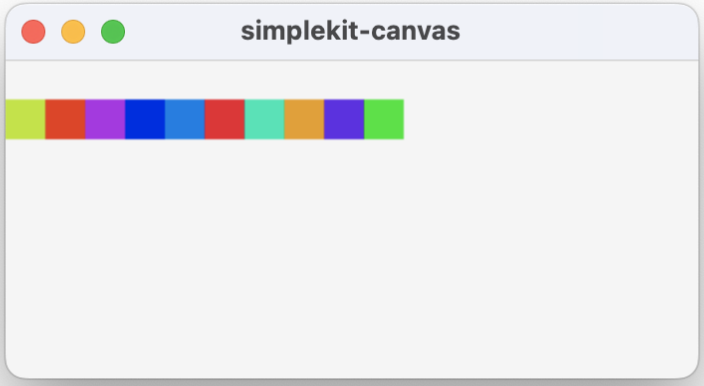
  `"rgb(255 0 0)"`, `"rgb(100, 149, 237)"`
- **HSL**: Hue, Saturation, Luminance

  `"hsl(0deg 100% 100%)"`, `"hsl(219deg 58% 93%")`
- (many other formats and variations)

useful colour guide

simplekit-canvas colourDemo()

- Using string literal to set colour
- How to prevent flicker?



CanvasRenderingContext2D State

- Convenient to save and restore the state of drawing styles
  - strokeStyle, fillStyle, lineWidth, font, textAlign, textBaseline, …

`save()` to push current drawing state to stack

`restore()` to pop last saved drawing state and restore it

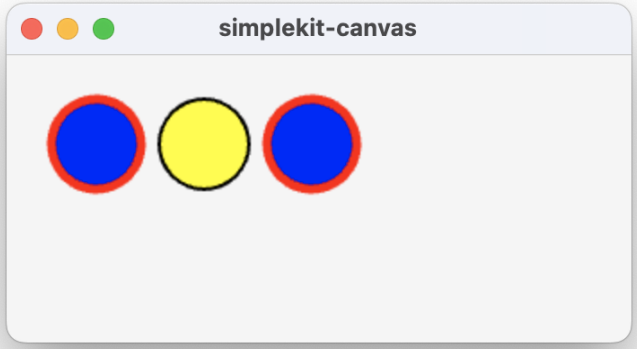
- Can call save() multiple times without restore(), each call pushes a state onto the stack that can be popped off later

simplekit-canvas saveState()

```
gc.fillStyle = "blue";
gc.strokeStyle = "red";
gc.lineWidth = 5;
circle(50, 50);

gc.save();

gc.fillStyle = "yellow";
gc.strokeStyle = "black";
gc.lineWidth = 2;
circle(110, 50);

gc.restore();

circle(170, 50);
```

save state: fill = blue, stroke = red, lineWidth = 5

restore state back to: fill = blue, stroke = red, lineWidth = 5



simplekit-canvas fpsDemo()

- Demonstrates 60 FPS draw loop
  - Frame number
  - Frame-per-second calculation (with smoothing)
  - Importance of gc.clearRect
  - gc.canvas.width and gc.canvas.height

Drawable Objects

- Drawable class
- Display list
- Painter's Algorithm

Drawable Object

Drawing using the graphics context API can be tedious, *instead*:

1. Define interface for an object that can be drawn:

```
export interface Drawable {
  draw: (gc: CanvasRenderingContext2D) => void;
}
```

2. Define drawable objects like:

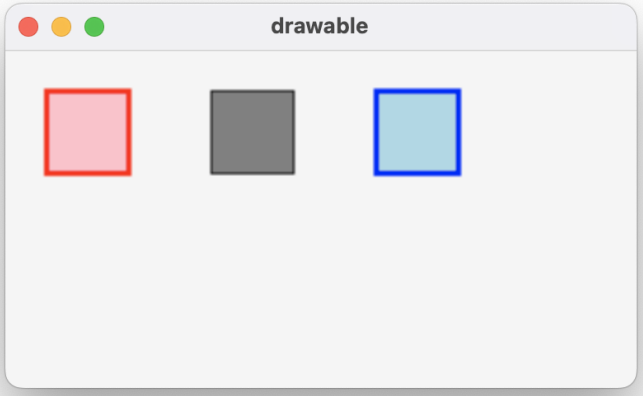
```
export class MyShape implements Drawable {
  ...
  draw(gc: CanvasRenderingContext2D) {
    // drawing commands go here
  }
}
```

3. Create the object and draw it using current graphics context:

```
const myShape = new MyShape( ... )
myShape.draw(gc);
```

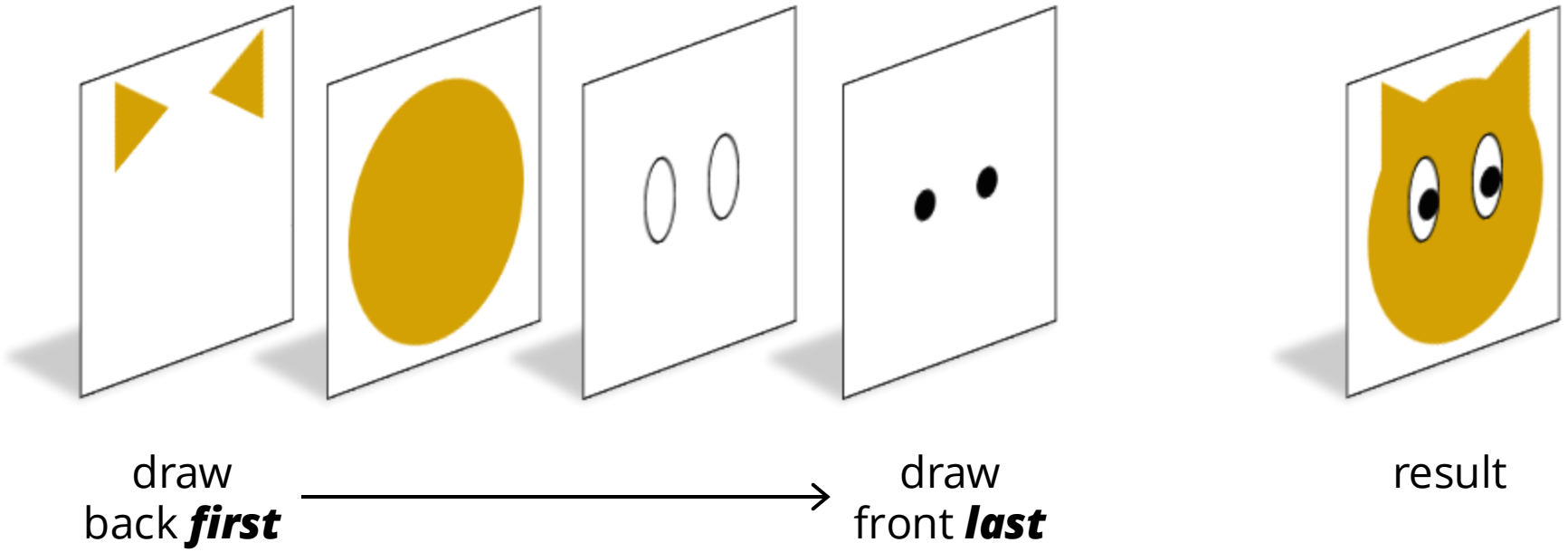
drawable squareDemo()

- ES module with objects
- Drawable interface
- Square1 is a basic drawable
- Add a fill property to Square1 and update draw code
- Square2 is drawable with props constructor
  - Convert Square1 calls to Square2



Painter's Algorithm

- Basic graphics primitives are (really) *primitive*.
- To draw more complex shapes:
  - Combine primitives
  - Draw back-to-front, layering the image
  - Called "Painter's Algorithm"

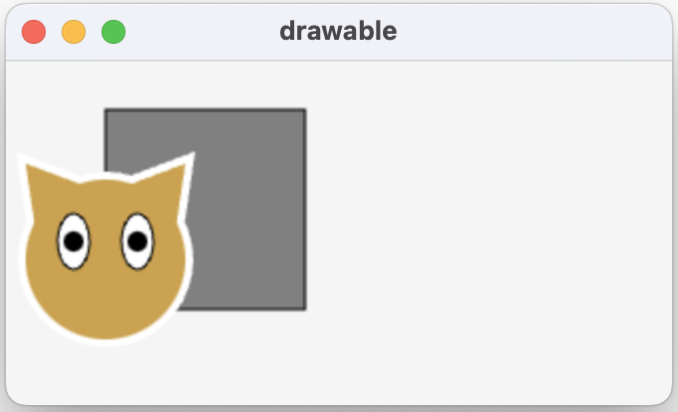


The 1 Minute Painting

- https://www.youtube.com/watch?v=0CFPg1m_Umg

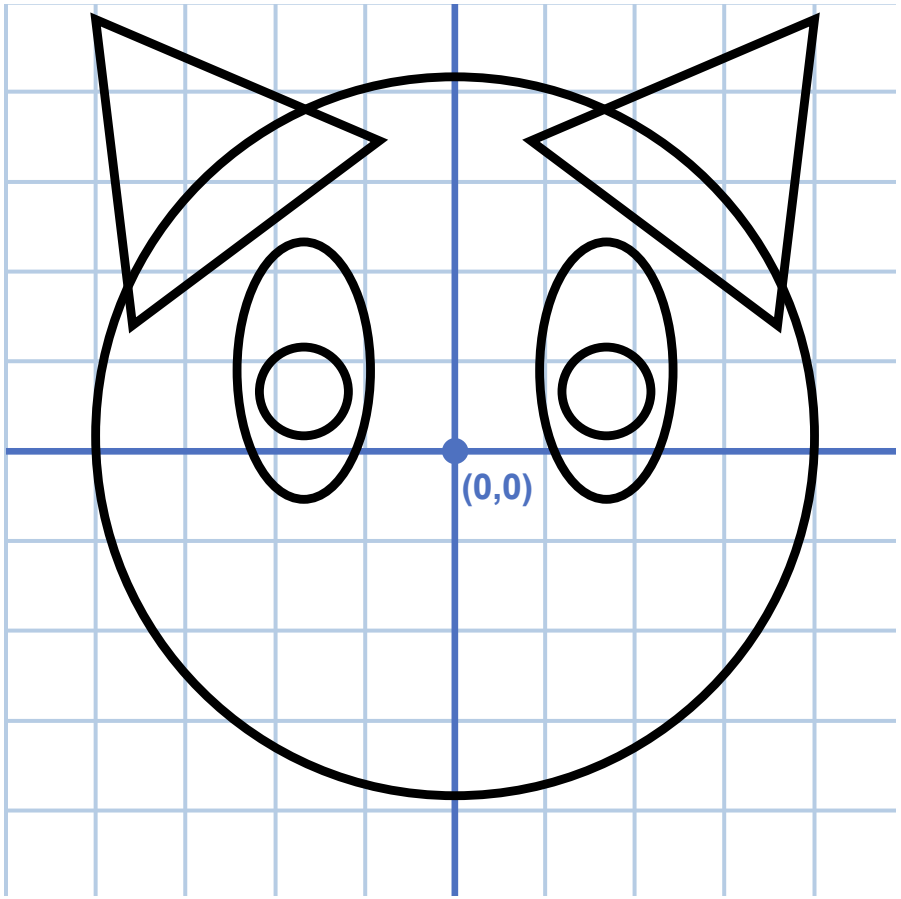
drawable paintersDemo()

- Draw order of square and cat
- Cat drawable example
  - not using props
  - drawing strategy (see next slide)
  - translate and scale in graphics context
  - need to save and restore state when transforming



Strategy to Draw Complex Shapes

- Draw with coordinates in convenient coordinate frame
- Transform shapes to desired location, e.g. `gc.translate( ..)`

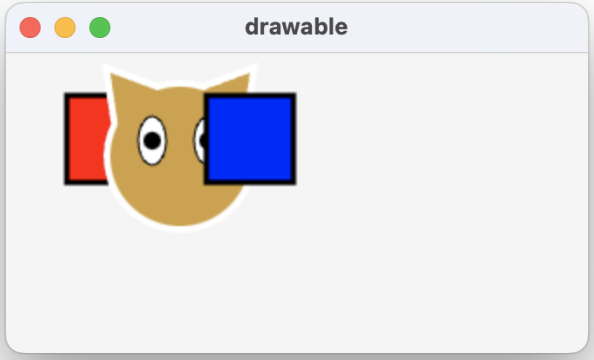


Display List

- Keep an ordered *display list* of `Drawable` objects
  - Add objects to array from back to front
  - (Could also add “z-depth” field and sort when object added)
- To draw all objects:
  - iterate through list and draw each one

drawable: displayListDemo()

- Create Cat and two Square2 objects, add to same displaylist
- The order added matters
- Adding many random Square2 shapes
  - moving object to front of displaylist



Efficiency

- Our common approach so far is to re-draw everything every frame
- Executing many graphics commands each frame is often fine
  - As a rule, don't optimize until you have to
  - When animating, you may have to re-draw everything
- With a very large number of drawables, each with a very large number of graphics context drawing commands, the frame rate may start to drop
  - In many cases, the drawable doesn't change each frame

Resources

MDN Canvas Tutorial

- https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial

Exercise

EXERCISE

1. **Create your own SimpleKit project**
   - Create a new minimal Vite project, install simplekit from npm
   - Import “canvas-mode” from simplekit, call startSimpleKit()
   - check console for the start up message
2. **Draw a button in SimpleKit canvas-mode**

   - Set your drawing callback function with SimpleKit
   - Use the painter’s algorithm to draw in layers
   - Add a conditional to optionally draw a yellow ”hover” highlight
3. **Make a Button drawable object**
   - Move your drawing code into a Drawable Button object
   - Parameterize the button’s position, size, and text (props in constructor)
   - Add a boolean property called “hover” to change how the button is drawn