Drawing

Drawing models

Graphics context

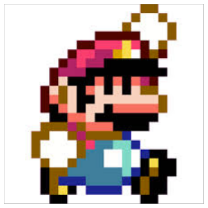
Display lists

Painter’s Algorithm

Clipping & Double-buffering

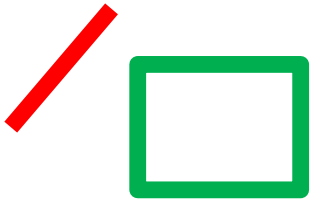
Drawing Primitives

- Three conceptual models for drawing:



Pixel

```
SetPixel(x, y, colour)
DrawImage(x, y, w, h, img)
```



Stroke

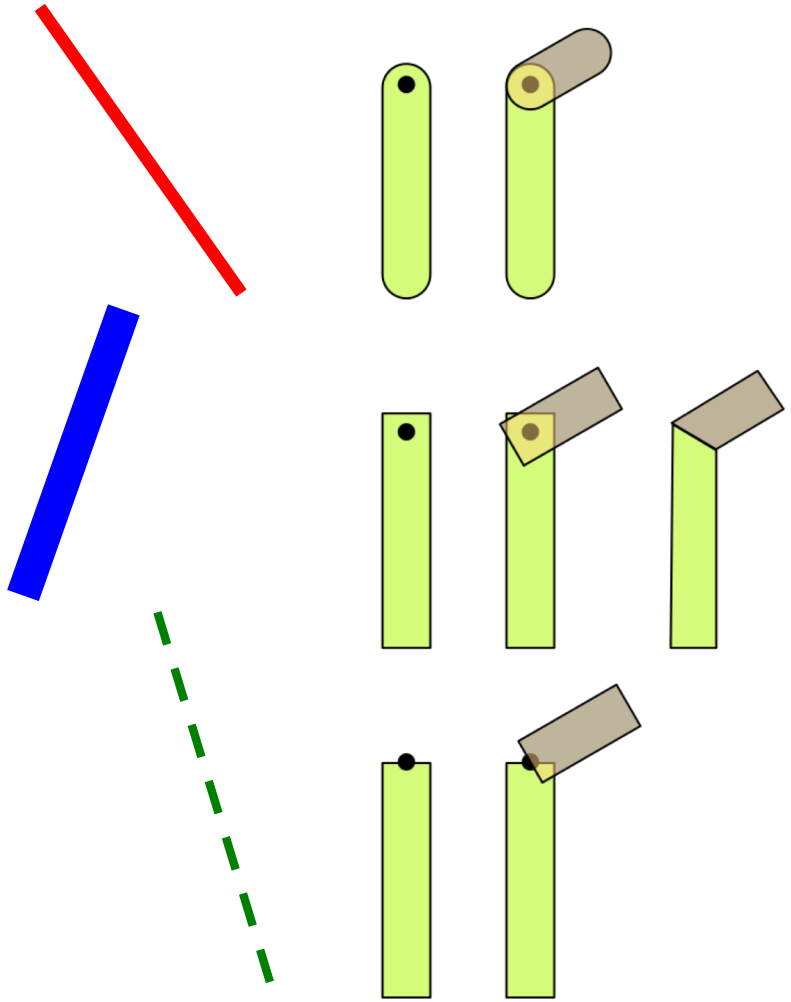
```
DrawLine(x1, y1, x2, y2, colour)
DrawRect(x, y, w, h, colour)
```

Region

```
DrawText("A", x, y, colour)
DrawRect(x, y, w, h, colour, thick, fill)
```

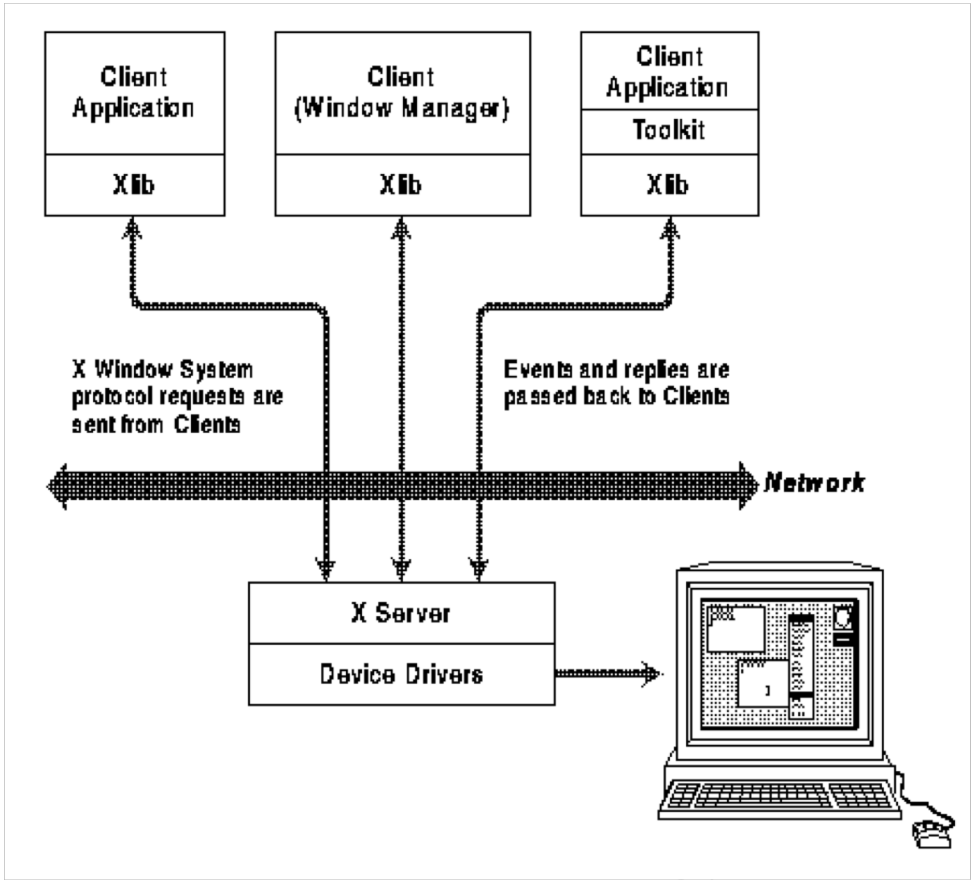
Drawing Options

- DrawLine()
- Many options:
  - what colour?
  - how thick?
  - dashed or solid?
  - where are the end points and how should the ends overlap?
- Observations:
  - most choices are the same for multiple calls to DrawLine()
  - lots of different parameters, but may only want to set one or two



Drawing Efficiently on XWindows

- Xwindows is a client-server architecture.
- Separates the *user interface* from *applications*:
  - an X Client handles all application logic (*application code*)
  - an X Server handles all display output and user input (*user interface*)
- Server handles request from client, process data as requested, and returns results to client
- We want the program running on one machine to efficiency draw on a second machine.
- How?



Graphics Context (GC)

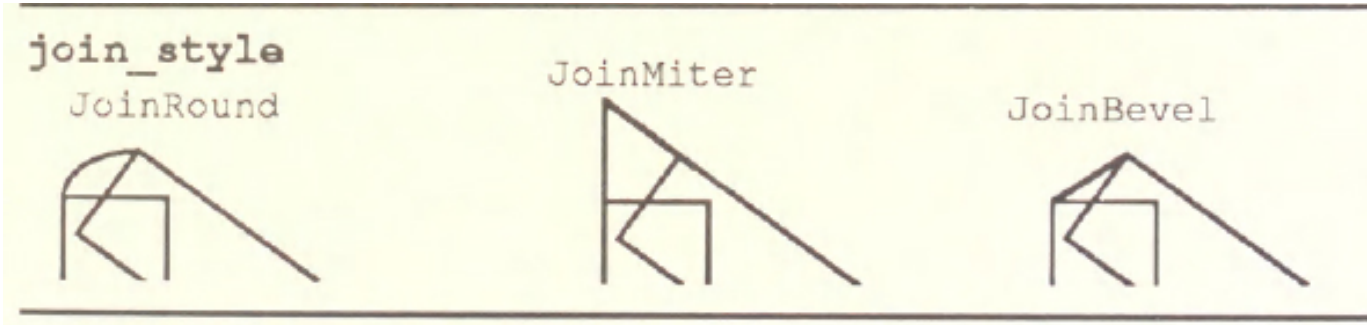
- Gather all options into a structure, pass it to the drawing routines
  - Easy to fall back to default parameters
  - Easy to only change only some parameters
  - Easy to switch between contexts
  - Efficient to pass once, since they don't often change

- In X, the Graphics Context (GC) is stored on the X Server
  - Inherit from a default global context for X Server
  - Fast to switch between contexts since reduced network traffic between X Server and X Client

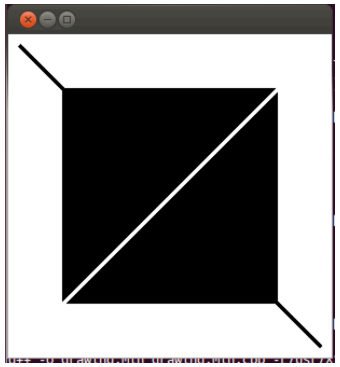
- Modern systems like Java and OpenGL *also* have a Graphics Context:
  - Java: Graphics Object
  - OpenGL: Attribute State

XGCValues (Xlib Graphics Context)

```
typedef struct {
    int function;  // how the source and destination are combined
    unsigned long plane_mask; // plane mask
    unsigned long foreground; // foreground pixel
    unsigned long background; // background pixel
    ...
    int line_width;     // line width (in pixels)
    int line_style;     // LineSolid, LineDoubleDash, LineOnOffDash
    int cap_style;      // CapButt, CapRound, CapProjecting
    int join_style;     // JoinMiter, JoinRound, JoinBevel
    int fill_style;     // FillSolid, FillTiled, FillStippled, …
    int fill_rule;      // EvenOddRule, WindingRule
    int arc_mode;       // ArcChord, ArcPieSlice
    ...
    Font font;          // default font
    ...
} XGCValues;
```

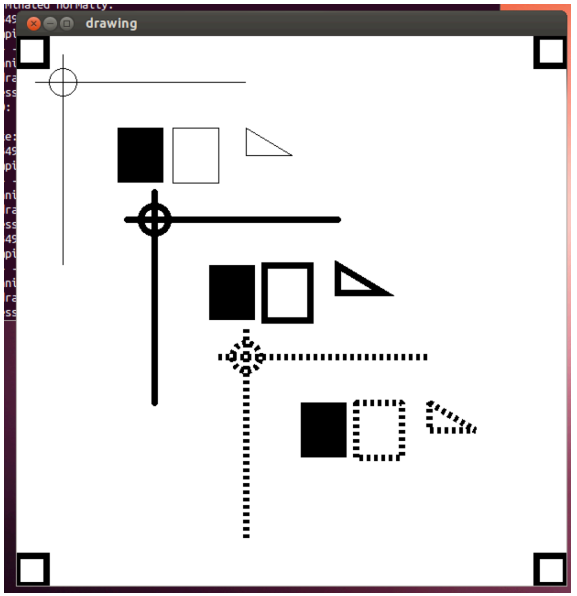


drawing.min.cpp



```
int w = 300;
int h = 300;
…
XFlush(display);
sleep(1); // give server time to setup before sending

// drawing demo with graphics context here ...
GC gc = XCreateGC(display, window, 0, 0); // graphics context
XSetForeground(display, gc, BlackPixel(display, screen));
XSetBackground(display, gc, WhitePixel(display, screen));
XSetFillStyle(display,  gc, FillSolid);
XSetLineAttributes(display, gc, 3,         // 3 is line width
        LineSolid, CapButt, JoinRound);    // other line options

// draw some things
XDrawLine(display, window, gc, 10, 10, w-10, h-10);
XFillRectangle(display, window, gc, 50, 50, w-(2*50), h-(2*50));
XSetForeground(display, gc, WhitePixel(display, screen));
XDrawLine(display, window, gc, w-10, 10, 10, h-10);
XFlush(display);
```

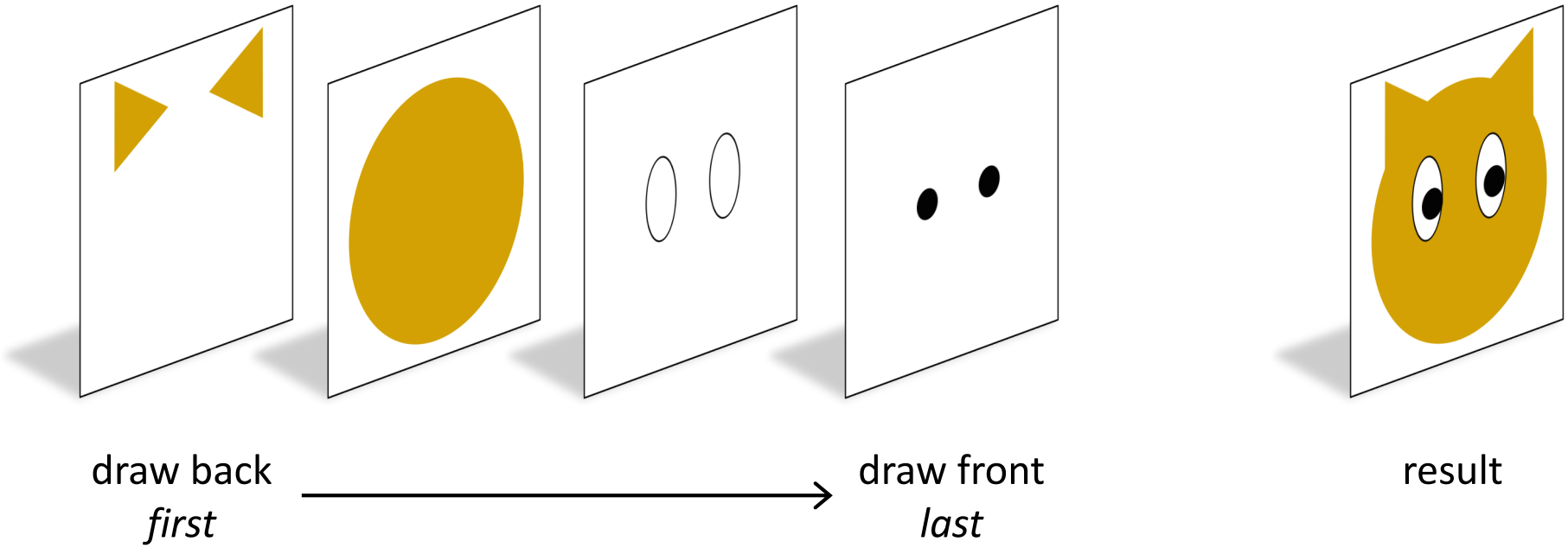
Code Review: drawing.cpp



- initX initializes three graphics contexts
- main calls several drawing functions
- drawRectanglesInCorners
  - get window attributes (e.g. width and height)
  - use of XDrawRectangle
- drawStuff
  - parameters say which GC and where to draw
  - use of XDrawLine, XDrawArc, XDrawRectangle, XFillRectangle
- Note: Minimize window and it vanishes
  - Need to redraw (need event to know when)

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"



Implementing Painters Algorithm

- Think about the things your program needs to paint:
  - can be low-level primitives like text, circle, polyline, etc.
  - or high level things like: game sprite, button, bar graph, etc.
- Package drawing of each thing into an object that can draw itself
  - Implement a `Displayable` base class with virtual "paint" method
  - Derive classes for the things you want to display
- Keep an ordered *display list* of `Displayable` objects
  - Order the list back-to-front (just us a FIFO stack for back-to-front drawing, or add "z-depth" field and sort on that)
- To repaint
  - Clear the screen (window)
  - Repaint everything in the display list (in back-to-front order)

Display List and “Displayables”

```
/*
 * An abstract class representing displayable things.
 */
class Displayable {

public:
      virtual void paint(XInfo &xinfo) = 0;

};
```

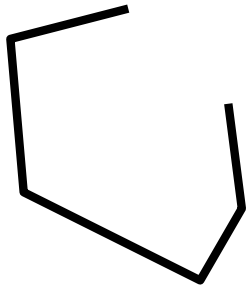
Displayable Text

ABC

```
class Text : public Displayable {

  public:
      virtual void paint(XInfo &xinfo) {
          XDrawImageString(xinfo.display, xinfo.window, xinfo.gc,
            this->x, this->y, this->s.c_str(),
            this->s.length() );
      }

      // constructor
      Text(int x, int y, string s) : x(x), y(y), s(s)  {}

  private:
      int x;
      int y;
      string s;
};
```

Displayable Polyline

```
class Polyline : public Displayable {
public:
    virtual void paint(XInfo& xinfo) {
       XDrawLines(xinfo.display, xinfo.window,
       xinfo.gc, &points[0],
       points.size(), CoordModeOrigin );
    }

    Polyline(int x, int y) {  add_point(x,y); }

    void add_point(int x, int y) {
       XPoint p; // XPoint is a built in struct
       p.x = x;   p.y = y;
       points.push_back(p);
     }

private:
     vector <XPoint> points; // 2D point struct
};
```



Displayable Face

```
class Face : public Displayable {
public:
  virtual void paint(XInfo& xinfo) {
    // draw head
    XFillArc(xinfo.display, xinfo.window, gc,
           x - (d / 2), y - (d / 2), d, d, 0, 360 * 64);

    // draw mouth either smiling or serious
    if (smile) {
       XDrawArc(xinfo.display, xinfo.window, gc, … );
    } else {
       XDrawLine(xinfo.display, xinfo.window, gc, … );
    }
  }

  // constructor
  Face(int x, int y, int d, bool smile)
    : x(x), y(y), d(d), smile(smile) {}

  private: int x; int y; int d; bool smile;
};
```

Displaying the Display List of Displayables

```
list<Displayable*> dList;  // list of Displayables

// draw in order you want
dList.push_back(new Text(10, 20, “Hello”));
dList.push_back(new Face(30, 40, 10, true));


// Function to repaint a display list
void repaint( list<Displayable*> dList, XInfo& xinfo) {
    list<Displayable*>::const_iterator begin = dList.begin();
    list<Displayable*>::const_iterator end = dList.end();

    XClearWindow(xinfo.display, xinfo.window);
    while( begin != end ) {
        Displayable* d = *begin;
        d->paint(xinfo);
        begin++;
    }
    XFlush(xinfo.display);
}
```

Events + Drawing = Animation

- A simulation of movement created by displaying a series of pictures, or frames.
- Goals:
  - Move things around on the screen
  - Repaint 24 - 60 times per second
    (frames-per-second, frame rate, or “FPS”)
  - Make sure events are handled on a timely basis
  - Don’t use more CPU than necessary
- Combination of drawing + event handling + timer handling

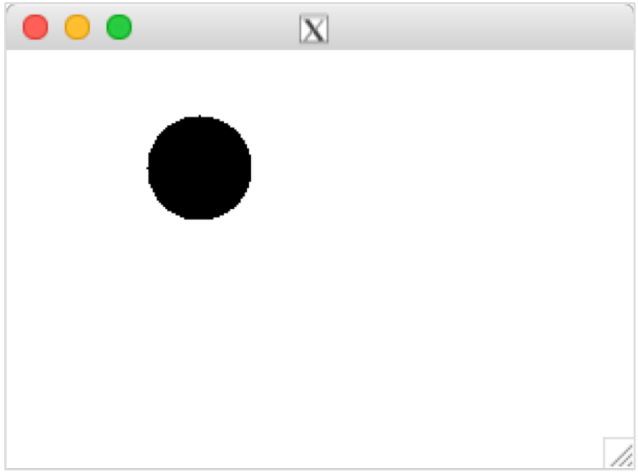
Animation Timing and Responding to Events (non-blocking)

```
while( true ) {

   if (XPending(display) > 0) {      // any events pending?
      XNextEvent(display, &event ); // yes, process them
      switch( event.type ) {
         // handle event cases here ...
      }
   }

   // now() is a helper function I made
   unsigned long end = now(); // time in microseconds

   if (end - lastRepaint > 1000000/FPS) { // repaint at FPS
      handleAnimation(xinfo); // update animation objects
      repaint(xinfo);         // my repaint
      lastRepaint = now();    // remember when last painted
   }

   // IMPORTANT: sleep for a bit to let other processes work
   if (XPending(xinfo.display) == 0) {
      usleep(1000000 / FPS - (end - lastRepaint));
   }
}
```

Code Review: animation.min.cpp

- Highlights:

```
XClearWindow(display, window);
ballPos.x += ballDir.x;
```



- Experiments to try:
  1. Resize the window.
  2. Comment out this:

```
XClearWindow(display, window);
```

  3. Comment out this (and closing bracket):

```
if (XPending(display) > 0) {
   XNextEvent( display, &event );
```

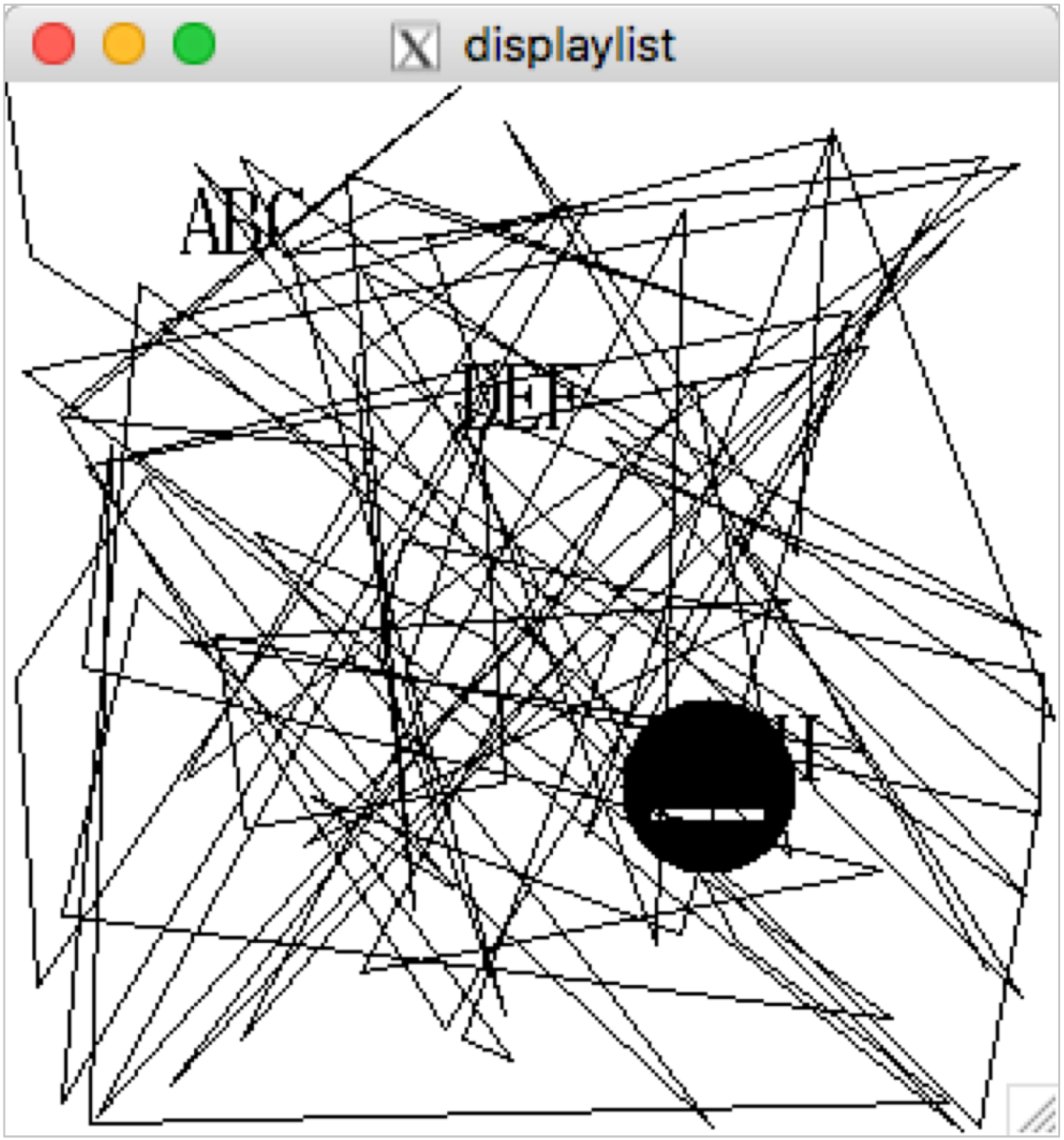
and try clicking mouse

  4. Comment out this:

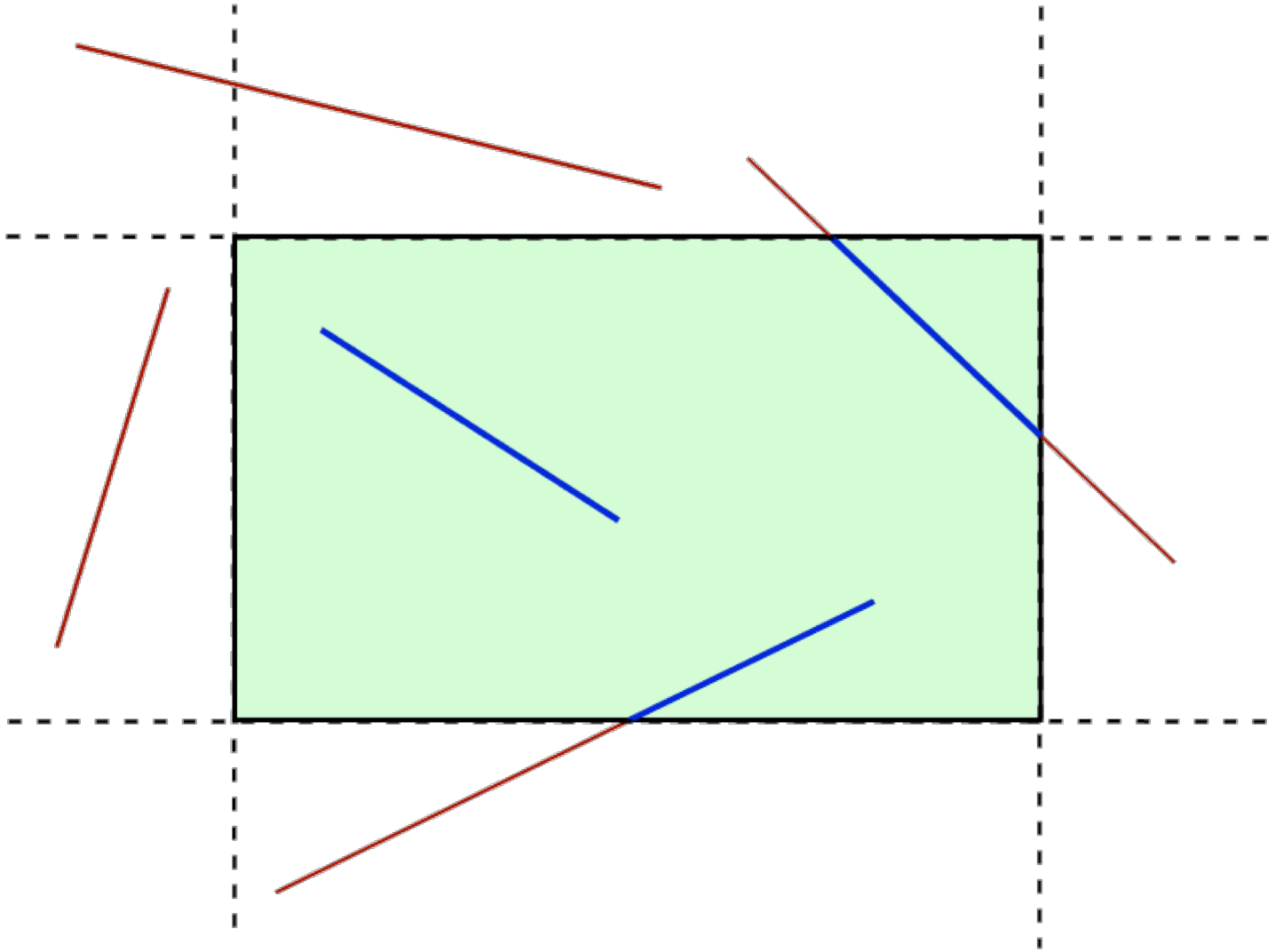
```
if (XPending(display) == 0) {
   usleep(1000000/FPS-(end-lastRepaint));
}
```

and look at CPU usage

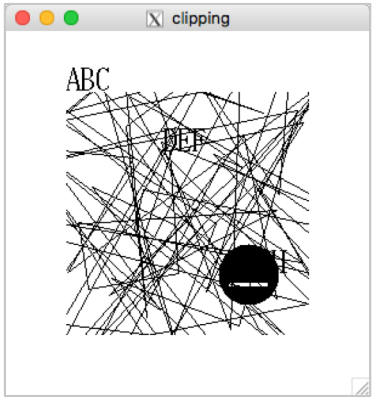
Code Walkthrough: displaylist.cpp



Clipping



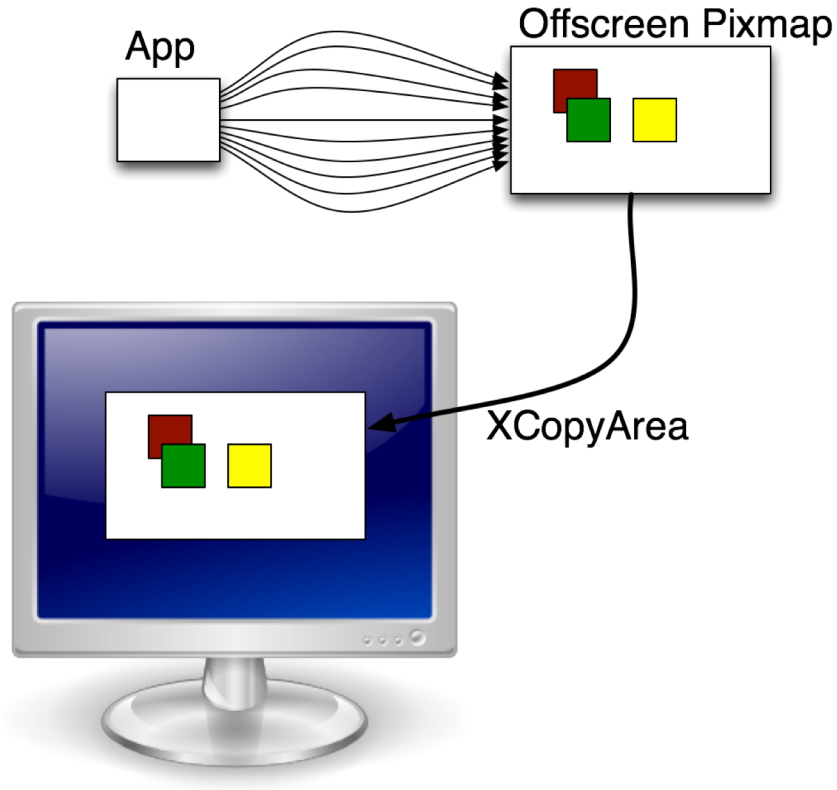
Clipping Code: clipping.cpp




```
// define clip window size
XRectangle clip_rect;
clip_rect.x = 50;
clip_rect.y = 50;
clip_rect.width = 200;
clip_rect.height = 200;

// clips all drawings using same GC after this call ...
XSetClipRectangles(xinfo.display, xinfo.gc,
                   0, 0, &clip_rect, 1, Unsorted);


// drawing commands here …


// turn clipping off again
XSetClipMask(xinfo.display, xinfo.gc, None);
```

Double Buffering

- Flickering when an intermediate image is on the display
- Solution:
  - Create an off screen image buffer
  - Draw to the buffer
  - Fast copy the buffer to the screen



Double Buffering: doublebuffer.cpp

```
// create off screen buffer
xinfo.pixmap = XCreatePixmap(xinfo.display, xinfo.window,
        width, height, depth); // size and *depth* of pixmap

// draw into the buffer
// note that a window and a pixmap are “drawables”
XFillRectangle(xinfo.display, xinfo.pixmap, xinfo.gc[0],
        0, 0, width, height);

// copy buffer to window
XCopyArea(xinfo.display, xinfo.pixmap, xinfo.window,
xinfo.gc[0],
        0, 0, width, height,  // pixmap region to copy
        0, 0);                // top left corner in window

XFlush( xinfo.display );
```

Painting Advice

- Keep it simple
  - Clear the window and redraw everything each frame
  - Use advanced methods (e.g. selective clearing, clipping) only if you *really* need them for performance
- Don’t repaint too often
  - remember framerate of display (60 FPS)
  - consider adding single “someChanged” bool flag
- Don’t flush too often
  - remember display framerate usually 60 FPS