Clipping algorithms: imagine a polyline that starts at the left edge of the window and ends at the right edge of the window. The line segments X coordinates increase monotonically but the y coordinates can be arbitrary. What would an efficient method be that would clip away any lines drawn in this window leaving only the part of those lines that are on or above the clipping line. The start and end points of lines to be clipped can be anywhere in the window. Describe a suitable algorithm and create corresponding C code.

For your scenario, where you want to clip a polyline such that only the part of each line segment that is on or above a given horizontal clipping line is kept, the most efficient approach is to use a parametric line clipping algorithm. This method checks each line segment for intersections with the clipping boundary and retains only the visible portion.

Suitable Algorithm

The parametric clipping algorithm works by:

- Representing each line segment parametrically as $P(t) = P_0 + t(P_1 - P_0)$, where $t \in [0, 1]$.
- For a horizontal clipping line at $y = y_{\text{clip}}$, find the value of t where the segment crosses y_{clip}.
- If both endpoints are above or on the clipping line, keep the segment as is.
- If both endpoints are below, discard the segment.
- If the segment crosses the clipping line, compute the intersection point and clip the segment to the visible part.

C Code Example

Here's a C function that clips a single line segment to the region above or on a horizontal clipping line:

```c
#include <stdio.h>

typedef struct {
    double x, y;
} Point;

// Clip a line segment to be above or on y = yclip
int clip_segment(Point *p0, Point *p1, double yclip) {
    // If both points are above or on the clipping line, keep the segment
    if (p0->y >= yclip && p1->y >= yclip) {
        return 1; // Segment is visible
    }
    // If both points are below, discard
    if (p0->y < yclip && p1->y < yclip) {
        return 0; // Segment is invisible
    }
    // Segment crosses the clipping line; compute intersection
    double t = (yclip - p0->y) / (p1->y - p0->y);
    if (p0->y < yclip) {
        // p0 is below, p1 is above or on
        p0->x = p0->x + t * (p1->x - p0->x);
        p0->y = yclip;
    } else {
        // p1 is below, p0 is above or on
        p1->x = p0->x + t * (p1->x - p0->x);
        p1->y = yclip;
    }
    return 1; // Segment is visible after clipping
}

// Example usage
int main() {
    Point p0 = {10, 20}, p1 = {50, 10};
    double yclip = 15;
    if (clip_segment(&p0, &p1, yclip)) {
        printf("Visible segment: (%.2f, %.2f) to (%.2f, %.2f)\n", p0.x, p0.y, p1.x, p1.y)
    } else {
        printf("Segment is invisible.\n");
    }
    return 0;
}
```

This code efficiently clips any line segment to the region above or on the horizontal clipping line, handling all possible cases for start and end points within the window.[1] [2]

⁂

The polyline is what is used to clip against so the segments of the polyline are clearly not horizontal. The lines to be clipped are not polylines, they are simple lines between two arbitrary points which may or may not intersect with any of the polyline segments. They may intersect with more than one polyline segment since the polyline can vary up and down at each of its line segments.

Thanks for the clarification. With a polyline clipping boundary (not horizontal) and arbitrary query lines that may intersect multiple polyline segments, a robust approach is to clip each query line segment against the polyline edge-by-edge, using a general polygon/polyline clipping framework.

Approach

- Treat the polyline as a sequence of consecutive line segments (p0_i to p1_i for i = 0..n-1).
- For each input line segment (a, b), compute its intersection with each polyline segment in order, collecting the portions that lie on or above the polyline boundary according to a chosen inside-test half-plane convention.
- Since the boundary is a polyline, the “inside” side is defined by a consistent winding or by a per-segment half-plane test if the polyline represents a closed clip region; for an open polyline you’ll need a rule to determine which side counts as visible (e.g., consistently use the left side when walking the boundary in its defined order).
- When a query segment intersects multiple polyline segments, accumulate clipped sub-segments in order, ensuring continuity where the query segment crosses the boundary.
- If the polyline is non-self-intersecting and the clipping region is the area on one side of the polyline, a standard polygon clipping algorithm can be adapted by converting the polyline into a polygon with a sensible closing path (e.g., add a far boundary edge to form a closed clipping region) and applying a general-purpose line clipping against that polygon.

Implementation outline

- Represent the polyline as an array of points poly[i].
- For each query segment q0–q1:
  - Initialize a list of kept subsegments with the full segment.
  - For i from 0 to m-2 (poly edges poly[i]–poly[i+1]):
    - Clip each current subsegment against the half-plane defined by the edge poly[i]–poly[i+1].
    - This yields new subsegments that are on the desired side with respect to that edge.

  - After processing all edges, the remaining subsegments are the portions of the query line that lie on or above the polyline boundary.
- Use a robust line-segment–segment intersection routine to determine intersections and compute intersection points.
- Handle edge cases: touching at endpoints, parallelism, and numerical tolerance.

C code sketch

The following sketch demonstrates clipping a single query segment against a polyline boundary by sequentially clipping against each edge’s half-plane. It assumes a consistent inside-test defined by the left side of each poly edge when traversing poly[i] to poly[i+1].

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct { double x, y; } Pt;

static double cross(Pt a, Pt b) { return a.x*b.y - a.y*b.x; }
static Pt operator_sub(Pt a, Pt b) { Pt r = {a.x-b.x, a.y-b.y}; return r; }

double orient(Pt a, Pt b, Pt c) {
    // cross product (b - a) x (c - a)
    Pt ab = operator_sub(b, a);
    Pt ac = operator_sub(c, a);
    return cross(ab, ac);
}

// Intersection of segments p1-p2 and q1-q2, if exists, returns true and sets r
int seg_intersect(Pt p1, Pt p2, Pt q1, Pt q2, Pt *r) {
    double d1 = orient(p1, p2, q1);
    double d2 = orient(p1, p2, q2);
    double d3 = orient(q1, q2, p1);
    double d4 = orient(q1, q2, p2);

    if (((d1>0 && d2<0) || (d1<0 && d2>0)) && ((d3>0 && d4<0) || (d3<0 && d4>0))) {
        // compute intersection parameterized (p = p1 + t*(p2-p1))
        // Solve using standard line intersection formulas
        double A1 = p2.y - p1.y;
        double B1 = p1.x - p2.x;
        double C1 = A1*p1.x + B1*p1.y;

        double A2 = q2.y - q1.y;
        double B2 = q1.x - q2.x;
        double C2 = A2*q1.x + B2*q1.y;

        double det = A1*B2 - A2*B1;
        if (fabs(det) < 1e-12) return 0; // parallel

        r->x = (B2*C1 - B1*C2) / det;
        r->y = (A1*C2 - A2*C1) / det;
        return 1;
    }
    // For non-strict intersections (including touching), handle epsilon cases as needed
    return 0;
}

// Clip a segment against the half-plane left of edge (a->b)
int clip_against_edge(Pt p, Pt q, Pt a, Pt b, Pt *out_p, Pt *out_q) {
    // Evaluate orientation relative to the edge
    double o_p = orient(a, b, p);
    double o_q = orient(a, b, q);
    const double eps = 1e-9;

    int in_p = o_p >= -eps;
    int in_q = o_q >= -eps;

    if (in_p && in_q) {
        // both inside
        *out_p = p; *out_q = q;
        return 1;
    }
    if (!in_p && !in_q) {
        // both outside
        return 0;
    }
    // They are on different sides; compute intersection with edge line
    Pt r;
    // Segment p-q with edge a-b
    if (seg_intersect(p, q, a, b, &r)) {
        if (in_p && !in_q) {
            // keep p to r
            *out_p = p; *out_q = r;
        } else if (!in_p && in_q) {
            // keep r to q
            *out_p = r; *out_q = q;
        } else {
            return 0;
        }
        return 1;
    }
    return 0;
}

// Clip a query segment against a polyline boundary (left side of each edge)
int clip_segment_against_polyline(Pt p0, Pt p1, const Pt *poly, int n, Pt *c0, Pt *c1) {
    Pt a = p0, b = p1;
    Pt t0 = a, t1 = b;
    int has_segment = 1;
    for (int i = 0; i < n-1; ++i) {
        Pt na = poly[i], nb = poly[i+1];
        Pt out_p, out_q;
        int kept = clip_against_edge(t0, t1, na, nb, &out_p, &out_q);
        if (!kept) return 0;
        t0 = out_p; t1 = out_q;
    }
    *c0 = t0; *c1 = t1;
    return 1;
}
```

```
// Example usage
int main(void) {
    // Polyline boundary (open or closed depending on convention)
    Pt poly[] = { {0,0}, {5,2}, {10,0}, {15,4} };
    int n = 4;

    Pt q0 = {2, 8}, q1 = {12, -1};
    Pt clip0, clip1;
    if (clip_segment_against_polyline(q0, q1, poly, n, &clip0, &clip1)) {
        printf("Clipped segment: (%.2f,%.2f) -> (%.2f,%.2f)\n",
               clip0.x, clip0.y, clip1.x, clip1.y);
    } else {
        printf("Segment lies completely outside the clipping polyline.\n");
    }
    return 0;
}
```

Notes

- The approach treats the polyline as a sequence of half-planes defined by consecutive edges. This works best when the clipping region is the inside of a closed polygon formed by the polyline; otherwise, a clear inside definition must be established for the open polyline.
- Numerical robustness can be improved with epsilons, and you may want to collect all inside portions if a single query line crosses the boundary multiple times.
- If performance is critical and many segments are clipped, consider batching operations or converting the polyline to a proper clipping polygon once, then applying a standard polygon clipping algorithm to each query segment.

rework this code assuming a screen coordinate space of -128:127 in X and Y, and fixed point arithmetic using no more than 16 bit integers.

A practical way to handle this in $[-128, 127]$ screen space with 16-bit integers is to keep all coordinates as signed 16-bit fixed-point and avoid division except in the line–line intersection, where a 16×16→32 multiply and a 32/32→32 divide are still safe on typical 16-bit/32-bit C targets.[20]

Below is an integer fixed-point rework using:

- Screen range: $-128..127$ in both axes.
- Fixed-point scale: $1/256$ (8 fractional bits) stored in `int16_t`.
- All intermediate products in `int32_t`.
- The polyline still defines the clipping “inside” as the left side of each edge; inputs and outputs are in fixed-point with the same scale.

```
#include <stdint.h>

typedef int16_t  fx16;   // Q8.8 fixed point
typedef int32_t  fx32;   // for intermediate math

typedef struct {
    fx16 x, y;   // Q8.8
} Pt;

// Fixed-point helpers: Q8.8
#define FX_ONE   ((fx16)256)   // 1.0 in Q8.8
#define FX_FROM_INT(i) ((fx16)((i) << 8))
#define FX_TO_INT(x)   ((int16_t)((x) >> 8))

static inline fx16 fx_mul(fx16 a, fx16 b) {
    // (a * b) >> 8, both Q8.8, result Q8.8
    return (fx16)(((fx32)a * (fx32)b) >> 8);
}

static inline fx16 fx_div(fx16 a, fx16 b) {
    // (a << 8) / b, both Q8.8, result Q8.8
    // caller must ensure |b| not too small
    return (fx16)(((fx32)a << 8) / (fx32)b);
}

// Vector helpers
static inline Pt pt_sub(Pt a, Pt b) {
    Pt r; r.x = a.x - b.x; r.y = a.y - b.y; return r;
}

// Cross product in Q16.16 (but we keep in 32-bit int)
static inline fx32 cross32(Pt a, Pt b) {
    // a,b are Q8.8; product is Q16.16
    return (fx32)a.x * (fx32)b.y - (fx32)a.y * (fx32)b.x;
}

// Orientation sign using 32-bit cross (no rescaling needed for sign)
static inline fx32 orient32(Pt a, Pt b, Pt c) {
    Pt ab = pt_sub(b, a);
    Pt ac = pt_sub(c, a);
    return cross32(ab, ac);    // Q16.16 in 32-bit
}

// Segment-segment intersection in fixed point.
// Returns 1 if proper intersection, sets *r; 0 otherwise.
static int seg_intersect(Pt p1, Pt p2, Pt q1, Pt q2, Pt *r) {
    fx32 d1 = orient32(p1, p2, q1);
    fx32 d2 = orient32(p1, p2, q2);
    fx32 d3 = orient32(q1, q2, p1);
    fx32 d4 = orient32(q1, q2, p2);

    if (((d1 > 0 && d2 < 0) || (d1 < 0 && d2 > 0)) &&
        ((d3 > 0 && d4 < 0) || (d3 < 0 && d4 > 0))) {

        // Solve intersection of two lines in fixed point.
        // Line 1: p1 + t*(p2 - p1)
        // Line 2: q1 + u*(q2 - q1)
        fx32 A1 = (fx32)p2.y - (fx32)p1.y;
        fx32 B1 = (fx32)p1.x - (fx32)p2.x;
        fx32 C1 = A1 * (fx32)p1.x + B1 * (fx32)p1.y;

        fx32 A2 = (fx32)q2.y - (fx32)q1.y;
        fx32 B2 = (fx32)q1.x - (fx32)q2.x;
        fx32 C2 = A2 * (fx32)q1.x + B2 * (fx32)q1.y;

        fx32 det = A1 * B2 - A2 * B1;
        if (det == 0) return 0; // parallel or nearly so

        // x = (B2*C1 - B1*C2) / det
        // y = (A1*C2 - A2*C1) / det
        fx32 numx = B2 * C1 - B1 * C2;
        fx32 numy = A1 * C2 - A2 * C1;

        // Both numerator and denominator are Q16.16-like; do 32/32 → Q16.16,
        // then truncate to Q8.8 by shifting 8 bits.
        fx32 x32 = numx / det;
        fx32 y32 = numy / det;

        r->x = (fx16)(x32 >> 8);   // convert Q16.16 → Q8.8
        r->y = (fx16)(y32 >> 8);
        return 1;
    }
    return 0;
}

// Clip a segment against half-plane left of edge (a->b).
// Inputs/outputs Q8.8. Returns 1 if some part survives.
static int clip_against_edge(Pt p, Pt q, Pt a, Pt b, Pt *out_p, Pt *out_q) {
    fx32 o_p = orient32(a, b, p);
    fx32 o_q = orient32(a, b, q);

    int in_p = (o_p >= 0);
    int in_q = (o_q >= 0);

    if (in_p && in_q) {
        *out_p = p; *out_q = q;
        return 1;
    }
    if (!in_p && !in_q) {
        return 0;
    }

    // One in, one out: compute intersection with edge
    Pt r;
    if (!seg_intersect(p, q, a, b, &r)) {
        return 0;
    }

    if (in_p && !in_q) {
        *out_p = p; *out_q = r;
    } else if (!in_p && in_q) {
        *out_p = r; *out_q = q;
    } else {
        return 0;
    }
    return 1;
}

// Clip a single segment p0->p1 against polyline poly[0..n-1] (left side inside).
// Returns 1 and sets *c0,*c1 if visible portion exists; 0 if fully outside.
int clip_segment_against_polyline(Pt p0, Pt p1,
                                  const Pt *poly, int n,
                                  Pt *c0, Pt *c1)
{
    Pt t0 = p0, t1 = p1;
    for (int i = 0; i < n - 1; ++i) {
        Pt na = poly[i];
        Pt nb = poly[i + 1];
        Pt out_p, out_q;
        if (!clip_against_edge(t0, t1, na, nb, &out_p, &out_q)) {
            return 0;   // fully clipped away
        }
        t0 = out_p;
        t1 = out_q;
    }
    *c0 = t0;
    *c1 = t1;
    return 1;
}

/* Example usage:

   // Coordinates given as integer pixels in [-128,127],
   // convert to Q8.8 with FX_FROM_INT.
   Pt poly[^3_4] = {
       { FX_FROM_INT(-128), FX_FROM_INT(-10) },
       { FX_FROM_INT(  -64), FX_FROM_INT( 20) },
       { FX_FROM_INT(   64), FX_FROM_INT(-15) },
       { FX_FROM_INT(  127), FX_FROM_INT(  5) }
   };

   Pt q0 = { FX_FROM_INT(-100), FX_FROM_INT( 40) };
   Pt q1 = { FX_FROM_INT( 100), FX_FROM_INT(-40) };

   Pt c0, c1;
   if (clip_segment_against_polyline(q0, q1, poly, 4, &c0, &c1)) {
       printf("Clipped: (%d,%d)->(%d,%d)\n",
              FX_TO_INT(c0.x), FX_TO_INT(c0.y),
              FX_TO_INT(c1.x), FX_TO_INT(c1.y));
   } else {
       printf("Segment fully outside\n");
   }
*/
```

This keeps all state and outputs in 16-bit fixed-point, uses only 32-bit intermediates for robustness in the intersection math, and respects the $[-128, 127]$ screen domain while clipping arbitrary segments against an arbitrary polyline “ceiling”.[20]

⁂

32 bit arithmetic is possible on the target system but extremely expensive and should be avoided whenever possible. Can you find a solution that eliminates or at worst minimizes the use of 32 bit arithmetic?

A good way to minimize 32-bit use is to avoid “analytic” segment–segment intersection and instead reuse your (presumably already optimized) 16-bit line rasterizer as the intersection engine. The idea is:

- Work entirely in 16-bit integer pixel space $[-128, 127]$ with no subpixel fixed-point.
- For each query segment, march along it with a Bresenham-style integer stepper.
- For each x that you step to, compute the clip polyline’s y at that x (also with pure 16-bit arithmetic) and only emit pixels where `y_query >= y_clip`.
- Optionally, coalesce consecutive visible pixels into “spans” if you still want line segments rather than per-pixel drawing.

This trades a bit more per-pixel work for the ability to stay in 16-bit and use only occasional 32-bit multiplies if you decide you need subpixel precision.[21]

Core idea in this setting

Given:

- Screen: integer coordinates $[-128, 127]$ in both x and y.
- Clipping polyline: vertices (x_i, y_i) in the same integer space, with strictly increasing x.
- Query segment: endpoints $(x0, y0)$, $(x1, y1)$ in integer space.

Use these properties:

- Because the polyline’s x increases monotonically, for any integer x you need only the unique polyline segment that spans that x to get its y by linear interpolation.[22]
- Integer linear interpolation for such small ranges can be done with 16-bit numerators and a 16-bit divisor, needing at most one 32-bit multiply and one 32-bit divide per step (or per segment-change of the polyline), rather than full 32-bit intersection math.

16-bit friendly interpolation helpers

All coordinates are plain `int16_t` pixels, no fractional bits. To avoid divide per pixel, the inner loop uses incremental updates.

```
#include <stdint.h>

typedef int16_t s16;
typedef int32_t s32;

typedef struct { s16 x, y; } Pt;

/* Bresenham-style stepper for a segment from p0 to p1.
   Returns next point on the segment each call until done.
   All arithmetic is 16-bit except the error term which is 32-bit but cheap. */

typedef struct {
    s16 x, y;
    s16 x1, y1;
    s16 sx, sy;
    s16 steep;
    s32 err;
    s16 dx, dy;
    s16 done;
} LineStepper;

void line_stepper_init(LineStepper *st, Pt p0, Pt p1)
{
    st->x  = p0.x;
    st->y  = p0.y;
    st->x1 = p1.x;
    st->y1 = p1.y;
    st->done = 0;

    s16 dx = p1.x - p0.x;
    s16 dy = p1.y - p0.y;

    s16 adx = dx >= 0 ? dx : -dx;
    s16 ady = dy >= 0 ? dy : -dy;

    st->steep = (ady > adx);   // step mostly in y

    if (st->steep) {
        // swap roles of x and y
        st->x  = p0.y;
        st->y  = p0.x;
        st->x1 = p1.y;
        st->y1 = p1.x;
        s16 t = dx; dx = dy; dy = t;
        adx = dx >= 0 ? dx : -dx;
        ady = dy >= 0 ? dy : -dy;
    }

    st->sx = (dx >= 0) ? 1 : -1;
    st->sy = (dy >= 0) ? 1 : -1;
    st->dx = adx;
    st->dy = ady;
    st->err = (s32)st->dx / 2;
}

/* Returns 0 when finished, 1 when produced a point in out.
   Outputs screen coords (un-swapped) in *out. */
int line_stepper_next(LineStepper *st, Pt *out)
{
    if (st->done) return 0;

    // current logical (possibly swapped) point
    s16 lx = st->x;
    s16 ly = st->y;

    // convert back if steep
    if (st->steep) {
        out->x = ly;
        out->y = lx;
    } else {
        out->x = lx;
        out->y = ly;
    }

    if (st->x == st->x1 && st->y == st->y1) {
        st->done = 1;
        return 1;
    }

    st->x += st->sx;
    st->err -= st->dy;
    if (st->err < 0) {
        st->y += st->sy;
        st->err += st->dx;
    }
    return 1;
}
```

This is standard Bresenham with one 32-bit add/sub per step, no multiplies or divides.[21]

Getting the polyline y(x) with minimal 32-bit work

Because `poly[i].x` increases monotonically, maintain a “current segment” index `k` such that:

- `poly[k].x <= x < poly[k+1].x`

Then approximate the clip y at integer x by incremental interpolation along that segment:

```
typedef struct {
    Pt *poly;
    s16 n;
    s16 k;          // current segment index
    Pt a, b;
    s16 dx, dy;
    s16 step_dir;   // +1 or -1 depending on scan direction in x
    s32 acc;        // 32-bit error accumulator
} PolySampler;

// Initialize to first segment that covers x0; direction is +1 or -1 for x progression.
void poly_sampler_init(PolySampler *ps, Pt *poly, s16 n, s16 x0, s16 dir)
{
    ps->poly = poly;
    ps->n = n;
    ps->step_dir = dir;

    // find initial segment index k such that [poly[k].x, poly[k+1].x] spans x0
    s16 k = 0;
    if (dir > 0) {
        while (k < n-2 && poly[k+1].x <= x0) k++;
    } else {
        k = n-2;
        while (k > 0 && poly[k].x > x0) k--;
    }
    ps->k = k;
    ps->a = poly[k];
    ps->b = poly[k+1];

    ps->dx = ps->b.x - ps->a.x;  // > 0 if monotone increasing
    ps->dy = ps->b.y - ps->a.y;
    ps->acc = 0;
}

// Advance sampler to match new x; assumes x steps by +/-1 each call.
void poly_sampler_step(PolySampler *ps, s16 x)
{
    // Move to next segment when passing its end x
    if (ps->step_dir > 0) {
        while (ps->k < ps->n-2 && x > ps->b.x) {
            ps->k++;
            ps->a = ps->poly[ps->k];
            ps->b = ps->poly[ps->k+1];
            ps->dx = ps->b.x - ps->a.x;
            ps->dy = ps->b.y - ps->a.y;
            ps->acc = 0;
        }
    } else {
        while (ps->k > 0 && x < ps->a.x) {
            ps->k--;
            ps->a = ps->poly[ps->k];
            ps->b = ps->poly[ps->k+1];
            ps->dx = ps->b.x - ps->a.x;
            ps->dy = ps->b.y - ps->a.y;
            ps->acc = 0;
        }
    }
}

// Get approximate clip y at given x: integer linear interpolation with Bresenham-like up
s16 poly_sampler_y(const PolySampler *ps, s16 x)
{
    // Basic integer linear interpolation along [a.x, b.x]:
    // parameter t = (x - a.x) / dx; y = a.y + t * dy
    // Implemented as: maintain running integer y and 32-bit accumulator outside this fur
    // For simplicity here, compute directly using small ranges; one 32-bit multiply+divi

    s16 dx = ps->dx;
    if (dx == 0) return ps->a.y; // vertical edge, arbitrary

    s16 dx_local = x - ps->a.x;
    s32 num = (s32)dx_local * (s32)ps->dy;   // max about (255 * 255) = 65025 << safe
    s16 delta = (s16)(num / dx);            // integer slope * dx_local
    return (s16)(ps->a.y + delta);
}
```

- Note `dx` is at most 255 in your window, and `dy` is at most 255, so `num` fits comfortably in 32 bits with a single multiply.[23]
- This is one 32-bit multiply and one 32-bit divide per sample; you can reduce that further by precomputing slope for each poly segment once and then doing integer DDA in the inner loop instead of per-sample divide.

Putting it together: clip via per-pixel test

The draw/clip routine:

```
// User-provided: draw one pixel or accumulate spans.
void draw_pixel(s16 x, s16 y);

void clip_and_draw_segment(Pt q0, Pt q1, Pt *poly, s16 n)
{
    LineStepper st;
    line_stepper_init(&st, q0, q1);

    s16 dir = (q1.x >= q0.x) ? 1 : -1;

    PolySampler ps;
    poly_sampler_init(&ps, poly, n, q0.x, dir);

    Pt p;
    while (line_stepper_next(&st, &p)) {
        // keep poly sampler in sync with x
        poly_sampler_step(&ps, p.x);
        s16 cy = poly_sampler_y(&ps, p.x);

        if (p.y >= cy) {
            draw_pixel(p.x, p.y);
        }
    }
}
```

Characteristics:

- All line stepping is 16-bit except the small 32-bit `err` term in Bresenham (add/sub only, no mul/div).
- The only 32-bit multiply/divide appears in `poly_sampler_y`, and its operands are tightly bounded ($|dx_local|, |dy| \le 255$), so the cost and overflow risk are minimal.[23]
- If even that is too expensive, replace `poly_sampler_y` with a DDA per poly segment: precompute fixed-point slope once per segment, then add an integer delta per x step; that removes per-sample division and reduces multiplies to “per segment”, not “per pixel”.

This fits your constraints: 16-bit coordinates in $[-128, 127]$, no 16×16→32 heavy geometry everywhere, and the polyline can wiggle arbitrarily as long as its x is monotonic.

⁂

There is no rasterizer. This is for a vector drawing system where the hardware handles drawing lines between pairs of 8-bit x,y coordinates.

The cheapest way to use your line-drawing hardware and still avoid general 32-bit segment–segment intersections is to clip in parameter space for the *query* line only, and use cheap 16-bit tests against the polyline to keep or reject intervals. The idea is:

- Parameterize the query line as $P(t) = A + t(B - A)$, $t \in [0, 1]$.
- For each polyline segment, find the parameter t_i where the query line intersects that segment's *supporting line* using a 2×2 determinant formula.
- Filter those t_i to only the ones where the intersection is actually within both segments' bounding boxes.
- Sort all valid t_i in ascending order, and between them test a representative point to know whether that interval lies above or below the polyline.
- Emit only the intervals of t where the query line is above the polyline; each such interval becomes a pair of endpoints you feed directly to your hardware.

All “heavy” math happens once per polyline edge, not per pixel, and everything stays safely in 16-bit with a few carefully placed 32-bit multiplies.

Geometry with bounded 16-bit/32-bit operations

Screen: 8-bit coordinates $[-128..127]$ for both x and y.

Represent everything as `int16_t`. Differences (dx, dy) are at most 255 in magnitude. Products like dx*dy are then at most about 65025, which fits comfortably in 32-bit. Using standard line intersection in determinant form:

- Query line: A = (x1, y1), B = (x2, y2).
- Poly edge: C = (x3, y3), D = (x4, y4).

Define:

- $\mathrm{den} = (x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)$
- $\mathrm{num} = (x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)$

Then the parameter along the query line is $t = \mathrm{num}/\mathrm{den}.$[41]
All these terms are sums/differences of products of two 8-bit differences, so each product uses 32-bit but is very cheap. The division to get a rational t can be kept in fixed-point (e.g. Q0.15 or Q0.14 in `int16_t`) so you never store 32-bit parameters.

Fixed-point parameter representation

Use t as Q0.15 in `int16_t`: 0 → 0.0, 32767 → ~0.9999.

- Compute `num` and `den` as 32-bit `int32_t`.
- If `den == 0`, the lines are parallel; skip.
- Compute `t32 = (num << 15) / den` as 32-bit, then clamp to `int16_t`.
- Reject if `t32 <= 0` or `t32 >= (1<<15)` (outside the segment).
- Then check the intersection also lies within the poly edge's bounding box with simple 16-bit comparisons (using approximate intersection with 32-bit once).

This uses:

- A few 16×16→32 multiplies and a 32/32 divide per edge.
- No 32-bit trigonometry or square roots.
- Only 16-bit storage for parameters and coordinates.

Algorithm outline

1. Collect intersections:
    - For each polyline edge `poly[i]-poly[i+1]`:
        - Compute `den` and `num` and thus `t` (Q0.15) for intersection along the query line.
        - If 0 < t < 1 and intersection point is within the edge's bounding box, store `t` in an array.
2. Sort the array of `t` values ascending (simple insertion sort: polyline count is small on your system).
3. Add implicit boundaries at `t = 0` and `t = 1` to form intervals: $[t_0, t_1], [t_1, t_2], \ldots$
4. For each interval $[t_i, t_{i+1}]$:
    - Take a midpoint parameter $t_m = (t_i + t_{i+1})/2$.
    - Evaluate the query point $P(t_m)$ in 16-bit fixed-point, then round to nearest `int16_t` coordinate.
    - Compute the polyline y at that x (same small integer interpolation as before, but now only once per interval).

    - If `P(t_m).y` is *above or on* the polyline, keep this interval: convert `t_i` and `t_{i+1}` to actual endpoints and send those to hardware.

This gives you a small set of line segments that are exactly the portions of the original query line above the polyline.

C code sketch with minimal 32-bit

This sketch assumes:

- Coordinates are `int16_t` in $[-128..127]$.
- You have a `draw_hw_line(x1,y1,x2,y2)` that accepts int8 or int16 endpoints.

```
#include <stdint.h>

typedef int16_t s16;
typedef int32_t s32;

typedef struct { s16 x, y; } Pt;

// Q0.15 fixed-point parameter t in [0,1)
#define T_ONE  ((s16)0x7FFF)

// Compute determinant (a.x*b.y - a.y*b.x) with bounded 32-bit
static s32 det2(s16 ax, s16 ay, s16 bx, s16 by)
{
    return (s32)ax * (s32)by - (s32)ay * (s32)bx;
}

// Intersection parameter t (Q0.15) along AB with CD
// Returns 1 if proper intersection with 0<t<1 and inside CD bounding box, sets *t_out.
static int intersect_param(Pt A, Pt B, Pt C, Pt D, s16 *t_out)
{
    s16 x1 = A.x, y1 = A.y;
    s16 x2 = B.x, y2 = B.y;
    s16 x3 = C.x, y3 = C.y;
    s16 x4 = D.x, y4 = D.y;

    s16 dx1 = x1 - x2;
    s16 dy1 = y1 - y2;
    s16 dx2 = x3 - x4;
    s16 dy2 = y3 - y4;

    s32 den = det2(dx1, dy1, dx2, dy2);
    if (den == 0) return 0; // parallel

    s16 dx13 = x1 - x3;
    s16 dy13 = y1 - y3;

    s32 num = det2(dx13, dy13, dx2, dy2);

    // Compute t in Q0.15
    s32 t32 = (num << 15) / den;
    if (t32 <= 0 || t32 >= T_ONE) return 0;
    // Optional: check intersection lies within CD bounding box
    // Compute intersection point once:
    s32 ux_num = det2(x1, y1, x2, y2);
    s32 vx_num = det2(x3, y3, x4, y4);
    s32 den2 = den;

    // x = det(ux,dx) / den; y = det(ux,dy) / den, but that’s more 32-bit.
    // To reduce cost, approximate using the param along CD similarly if needed.
    // For many shapes, skip and rely on monotone x and segment geometry.

    *t_out = (s16)t32;
    return 1;
}

// Evaluate query line at parameter t (Q0.15).
static Pt eval_at_t(Pt A, Pt B, s16 t)
{
    s16 dx = B.x - A.x;
    s16 dy = B.y - A.y;
    // x = A.x + (dx * t) >> 15
    s32 x32 = (s32)dx * (s32)t;
    s32 y32 = (s32)dy * (s32)t;

    Pt P;
    P.x = (s16)(A.x + (s16)(x32 >> 15));
    P.y = (s16)(A.y + (s16)(y32 >> 15));
    return P;
}

// Simple insertion sort for small arrays
static void sort_t(s16 *t, int n)
{
    for (int i = 1; i < n; ++i) {
        s16 v = t[i];
        int j = i - 1;
        while (j >= 0 && t[j] > v) {
            t[j+1] = t[j];
            --j;
        }
        t[j+1] = v;
    }
}

// Polyline y(x) via small integer interpolation; one 32-bit mul+div per call.
static s16 poly_y_at_x(const Pt *poly, int n, s16 x)
{
    // Find segment [k, k+1] such that x in [poly[k].x, poly[k+1].x]
    int k = 0;
    while (k < n-2 && poly[k+1].x < x) k++;

    Pt A = poly[k];
    Pt B = poly[k+1];

    s16 dx = B.x - A.x;
    if (dx == 0) return A.y;
    s16 dxl = x - A.x;
    s16 dy = B.y - A.y;

    s32 num = (s32)dxl * (s32)dy;
    s16 delta = (s16)(num / dx);   // integer interpolation
    return (s16)(A.y + delta);
}

// Hardware line draw placeholder
void draw_hw_line(s16 x1, s16 y1, s16 x2, s16 y2);

void clip_line_against_polyline(Pt A, Pt B, const Pt *poly, int n)
{
    // Collect t where query line intersects poly segments
    s16 tvals[^5_32];    // adjust size as needed
    int  tcount = 0;

    for (int i = 0; i < n-1 && tcount < 32; ++i) {
        s16 t;
        if (intersect_param(A, B, poly[i], poly[i+1], &t)) {
            tvals[tcount++] = t;
        }
    }

    // Add boundaries 0 and 1
    tvals[tcount++] = 0;
    tvals[tcount++] = T_ONE;

    sort_t(tvals, tcount);

    // For each interval, test midpoint and draw if above polyline
    for (int i = 0; i < tcount-1; ++i) {
        s16 t0 = tvals[i];
        s16 t1 = tvals[i+1];
        if (t1 <= t0) continue;

        s16 tm = (s16)(((s32)t0 + (s32)t1) >> 1);

        Pt Pm = eval_at_t(A, B, tm);
        s16 yc = poly_y_at_x(poly, n, Pm.x);

        if (Pm.y >= yc) {
            Pt Pstart = eval_at_t(A, B, t0);
            Pt Pend   = eval_at_t(A, B, t1);
            draw_hw_line(Pstart.x, Pstart.y, Pend.x, Pend.y);
        }
    }
}
```

Properties vs. your constraints

- All coordinates are 8-bit in range; all stored as `int16_t`.
- 32-bit arithmetic is used only where mathematically necessary:
    - A handful of 16×16→32 multiplies and a single 32-bit divide per poly segment to compute each intersection parameter.

    - A few 16×16→32 multiplies for evaluating points on the query line and polyline.
- No 32-bit storage; no matrices; no rasterization.
- The hardware still just receives clean `(x,y)` pairs for straight lines, and you get exactly the visible sub-segments above the polyline.

⁂

Thank you, we will try that solution.

Glad it helps. If you run into any precision or performance quirks when you integrate it with your vector hardware, share a few representative polylines and test segments and it can be tuned further (e.g., reduce divides, tighten bounds on t, or specialize for mostly upward/downward polylines).[42]

⁂

1. https://pmc.ncbi.nlm.nih.gov/articles/PMC9605407/
2. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-688.pdf
3. https://www.thiyagaraaj.com/tutorials/computer-graphics-programs-using-c-programming/line-clipping-program-using-c-programming
4. http://www.richardssoftware.net/Home/SearchTag?tag=Line+Clipping
5. https://wavoowajeehacollege.in/assets/pdf/e-content/cs/UNIT 3 - CLIPPING OPERATIONS.pdf
6. https://www.cs.montana.edu/courses/spring2009/425/dslectures/clipping.pdf
7. https://cgl.ethz.ch/teaching/former/gdv1_04/GDV1_WS04/lecture/09_clipping_4.pdf
8. https://redirect.cs.umbc.edu/~ebert/435/notes/435_ch4.html
9. https://www.scribd.com/document/856754653/5-Clipping-Algorithms
10. https://open.library.ubc.ca/media/stream/pdf/831/1.0096928/1
11. https://www.slideshare.net/slideshow/line-clipping-76333693/76333693
12. https://zappysautowashes.com/blogs/news/the-ultimate-guide-to-choosing-the-right-soap-for-your-car-wash?srsltid=AfmBOoo9TNlVUyv5QpwzI_0WWOtUY3sfNbzur4TnVTbRQsmBZXy0QdHu
13. https://www.academia.edu/31245761/Fast_algorithms_for_clipping_lines_and_line_segments_in_E_2
14. https://stackoverflow.com/questions/31430544/fixing-the-code-for-line-clipping-algorithm
15. https://github.com/topics/clipping-algorithm?l=c%2B%2B&o=asc&s=forks
16. https://programmerbay.com/program-for-clipping-a-line-using-cohen-sutherland-algorithm/
17. https://www.thecrazyprogrammer.com/2017/02/cohen-sutherland-line-clipping-algorithm.html
18. https://lodev.org/cgtutor/lineclipping.html
19. https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/Clipping.html
20. https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/Clipping.html
21. https://www.pascalgamedevelopment.com/showthread.php?5143-16-16-integer-fixed-point-line-drawer-)
22. https://pmc.ncbi.nlm.nih.gov/articles/PMC9605407/
23. https://www.cs.jhu.edu/~misha/Spring16/15.pdf

24. https://www.cs.princeton.edu/~chazelle/pubs/IntersectLineSegments.pdf
25. https://www.sciencedirect.com/science/article/abs/pii/009784939290029U
26. https://www.realtimerendering.com/resources/GraphicsGems/gemsiii/insectc.c
27. https://resources.mpi-inf.mpg.de/departments/d1/teaching/ws09_10/CGGC/Notes/Arrangements.pdf
28. https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1140241
29. https://www.gorillasun.de/blog/an-algorithm-for-polygon-intersections/
30. https://www.semanticscholar.org/paper/Fixed-Point-Arithmetic-Line-Clipping-Vayá-Jorquera/afde293dc2f470f7420a650d5f59e2098aa37566
31. https://www.cs.ubc.ca/sites/default/files/tr/1993/TR-93-32_0.pdf
32. https://www.youtube.com/watch?v=5FkOO1Wwb8w
33. https://stackoverflow.com/questions/31430544/fixing-the-code-for-line-clipping-algorithm
34. https://cp-algorithms.com/geometry/check-segments-intersection.html
35. https://stackoverflow.com/questions/36658998/line-segment-intersection-algorithm
36. https://www.topcoder.com/community/data-science/data-science-tutorials/geometry-concepts-line-intersection-and-its-applications/
37. https://www.youtube.com/watch?v=TFl4xQO_rw4
38. https://ti.inf.ethz.ch/ew/courses/CG08/materials/v3.pdf
39. https://jvm-gaming.org/t/fastest-linesintersect-method/35387
40. https://github.com/AngusJohnson/Clipper2/discussions/334
41. https://cp-algorithms.com/geometry/check-segments-intersection.html
42. https://pmc.ncbi.nlm.nih.gov/articles/PMC9605407/