CSCE 441 Computer Graphics

scan conversion of lines

- horizontal, vertical lines are easy
- for general lines, assume $0 < slope < 1$ (flat to diagonal)
  - you can transform any line to fit this
- naive algorithm would just use floating point and round off
  - floating point is sometimes slow (especially back when not every computer did it in hardware)
- slope from two points:

$$m = \frac{y_H - y_L}{x_H - x_L}a$$

- $s\frac{a}{b}a$
- intercept from two points: $b = y_L - m * x_L$
- **Simple Algorithm**
  - start from (xL, yL) and draw to (xH, yH)
    - where $xL < xH$

```
def draw_line(xL, yL, xH, yH):
    x, y = (xL, yL)
    for i in range(0, xH - xL):
        draw_pixel(x, round(y))
        x = x + 1
        y = m * x + b # simplifies to
        y = y + m
```

  - problem: uses floating point math
  - problem: rounding
- **Midpoint Algorithm**
  - given a point, we just need to know whether we will move right or up and right on the next step (N or NE)
  - we can simplify this to whether the actual line travels above or below the point $(x + 1, y + 1/2)$
    - so we derive formula from $y = m * x + b$
  - formula: $f(x, y) = c * x + d * y + e$
    - $c = yL - yH$
    - $d = xL - xH$
    - $e = b * (xL - xH)$
    - $f(x, y) = 0$: (x, y) is on the line
    - $f(x, y) < 0$: (x, y) below line
    - $f(x, y) < 0$: (x, y) above line
  - don't want to recalculate formula at every step, so do it iteratively
    - that is, use $f(x + 1, y + 1/2)$ to calculate $f(x + 2, y + 1/2)$ or $f(x + 2, y + 3/2)$ depending on right or up-right choice last time
  - went right last time, now calculate $f(x + 2, y + 1/2)$

    * $f(x+2, y+1/2) = c + f(x+1, y+1/2)$
  - went up-right last time, now calculate $f(x+2, y+1/2)$
    * $f(x+2, y+3/2) = c + d + f(x+1, y+1/2)$
  - starting value: $f(x+1, y+1/2) = f(xL, yL) + c + (1/2)d = c + (1/2)d$
    * we can eliminate $f(xL, yL)$ because we know it is on the line
    * furthermore, we can use $f(x+1, y+1/2) = 2*c+d$ because multiplying by 2 does not change the sign of f. Also, this saves an expensive division
  - full algorithm:

```
def midpoint_algorithm_line(xL, yL, xH, yH):
    x = xL
    y = yL
    d = xH - xL
    c = yL - yH
    sum = 2*c + d
    draw_pixel(x,y)
    while x < xH:
        if sum < 0:
            sum += 2*d
            y += 1
        x += 1
        sum += 2*c
        draw_pixel(x,y)
```

  - pro:
    * only integer operations
    * extends to other kinds of shapes, just need formula to tell if inside/outside shape (called implicit formula)
  - same as Bresenham's algorithm (more common algorithm)

scan conversion of polygons

- to deal with overlap, we do not draw the top and right of a polygon
  - this means artifacts are possible. This doesn't really matter since pixels are very small
- rectangles (aligned with axes) are easy
- scan line: one row of pixels
- general polygons: basic idea (**scanline method**)
  - intersect scan lines with edges of polygon
  - this means you must keep track of which edges intersect with which scan lines
    * this is easy to do: just look at the y coordinate
  - consecutive scan lines will usually intersect with a similar set of edges
    * so we can use coherence to speed stuff up
  - we can throw out horizontal lines. They are implicitly represented by

start and end, connecting to the other edges
  - data structures
    - edge: maxY, currentX, xIncr (increment)
      - calculate these from the two points
      - xIncr is inverse of slope, but you can't calculate the slope and invert it, because divide by 0
      - maxY: y value of higher point
      - currentX: x value of lower point
    - active edge table
      - has entry for every scanline on the screen
      - initialize with edges by indexing by minY of edge
    - active edge list
      - stores edges that intersect with the current scan line being processed
      - edges must always be sorted by current x value
  - at each step of the algorithm, you must update the active edge list
    - remove edges where maxY is less than or equal to the current scan line
      - less or equal because we don't draw the top and right of the polygon
    - add edges from the current scan line to the edge list
    - sort all edges by currentX
  - then draw the scan line
    - take pairs of edges and fill in between their currentX values
      - do not include the right point (because we don't draw the top and right of the polygon)
    - if you ever have an odd number of edges in the active edge list, you made a mistake
  - disadvantages
    - does not handle long, thin polygons well
    - incremental updates are not suitable for massively parallel GPUs
- boundary fill
  - draw the boundary of the polygon, then fill in interior
    - fill in interior wherever it is not the same color as you are drawing
  - need to be sure filling can't escape out from an edge or corner
  - need to be able to choose arbitrary interior point to start from
- flood fill
  - starting at point, recursively replace one color with another
  - paint bucket tool

openGL data CPU to GPU

- openGL can accept data various ways, with different speed impacts
- speed depends on driver implementation

    - GPUs only render triangles, and triangles usually share vertexes with other triangles, so saving lots of bandwidth is possible
- fastest is usually vertex buffer objects?
    - stores data directly on GPU?

clipping lines

- it's not really possible to draw things that are outside of the viewing area
- clipping points is easy (when comparing to rectangular window)
- clipping lines:
    - if both end points are inside window, draw it
- window intersection method:
    - if either or both is outside, intersect line with each window border in sequence
    - $(x_1, y_1), (x_2, y_2)$ intersect with vertical edge at x_{right}: $y_{intersect} = y_1 + m * (x_{right} - x_1)$, where $m = (y_2 - y_1)/(x_2 - x_1)$
    - $(x_1, y_1), (x_2, y_2)$ intersect with horizontal edge at y_{bottom}: $x_{intersect} = x_1 + (y_{bottom} - y_1)/m$, where $m = (y_2 - y_1)/(x_2 - x_1)$
    - all these intersections are costly to compute
        - we would like to efficiently handle trivial accepts and trivial rejects
- cohen-sutherland algorithm
    - classify two points p_1, p_2 using 4-bit codes `c0` and `c1`
    - if `c0 & c1 != 0`: trivial reject
        - bitwise AND
        - both points are outside one of the boundaries
    - if `c0 | c1 == 0`: trivial accept
        - bitwise OR
        - none of the coordinates of either point is outside any boundary => line is entirely within window
    - otherwise split line until it is a trivial case
    - bits: `| top | bottom | right | left`
        - doesn't matter as long as you're consistent? TODO
        - you can determine each of these by just comparing one coordinate with the axes
        - thus the comparison is fast
    - disadvantages
        - repeated clipping is expensive
    - advantages
        - considers all possible trivial accept/reject
- laing-barsky algorithm
    - use parametric form of line for clipping
        - means that lines are oriented (have a direction)
    - need to classify lines as moving into or out of the window

  - since lines are parametric, we will be finding the parameter value of the intersection
    - we can put that back into the formula to get the actual point
  - parametric lines
    - $x(t) = x_0 + (x_1 - x_0) * t$
    - $y(t) = y_0 + (y_1 - y_0) * t$
    - $0 \leq t \leq 1$
    - solve 2d matrix to intersect lines:

      ```
      [ x1-x0, x2-x3 ][ t ] == [ x2-x0 ]
      [ y1-y0, y2-y3 ][ s ] == [ y2-y0 ]
      ```
  - algorithm:
    - start with t on range $[0, 1]$
      - this is t_{min}, t_{max}
    - iteratively intersect each line with each edge
      - find intersection at t
      - if line is moving in to out: $t_{max} = min(t_{max}, t)$
      - else: $t_{min} = max(t_{min}, t)$
      - if $t_{min} > t_{max}$: reject line
  - moving out vs moving in can be determined by looking at coordinates
    - different for each boundary
    - e.g. for right boundary, $x_1 < x_2$ is moving in
    - does not depend on where either point is, or whether either point is inside/outside window boundary, just relative positions of the points
  - disadvantages
    - does not consider trivial accept/reject
  - advantages
    - computation of (x, y) is done only once at the end
    - computation of parametric intersections is fast (only one division)
- note: clipping line and then rounding to integer coordinates may not produce the correct result, due to round-off error
  - can account for this by calculating sum for use in midpoint algorithm

clipping polygons

- clipping a polygon can change the number of sides it has
  - minimum number of sides is 3 (triangle)
  - maximum number of sides is $2n + 1$? TODO
  - e.g. maximum number of sides of triangle after clipping is 7 sides
- when clipping convex polygons, you could end up with multiple polygons
- sutherland-hodgman clipping
  - clip polygon vs each edge of window individually
  - TODO can this algorithm handle non-rectangular windows?
  - is not guaranteed to handle convex polygons correctly

    * does not split into multiple polygons
    * but usually looks about right
  - output is mixture of old/new vertexes
    * will be exactly old vertexes if polygon was entirely inside the window
    * will be only new vertexes if all vertexes were outside the window (but not necessarily all edges)
  - process each side of the rectangular window separately
    * and also, process each edge in polygon iteratively
  - 4 cases for an edge from S to E:
    * S and E both outside: no output
    * S and E both inside: output only E
    * S inside, E outside: compute intersection with border, and output that
    * S outside, E inside: output intersection with border, and output E
  - output of one intersection is used as input for next intersection
    * you can kind of do these in parallel, with the partial output from the previous stage
      · pipeline
    * then you need a end-of-polygon marker, and you need to use that along with the first edge to make the last edge
- weiler-atherton algorithm
  - general intersection between any two kinds of polygons
  - handles non-convex polygons
    * thus can output more than one polygon for a single input polygon
  - not as efficient as sutherland-hodgman
    * all those intersections are expensive
    * difficult to parallelize
  - algorithm
    * start at point on polygon
    * follow polygon edges counterclockwise until an edge crosses out of the window
    * follow window edges from the intersection point until the polygon intersects again
    * now that part is a polygon. Go back to the first intersection point and follow the polygon until it re-enters the window, and find more polygons

transformations in 2D

- coordinates
  - need point of origin (0,0) and axes (x and y)
  - we want to define transformations generally, without need for coordi-

nates
  - but hardware uses coordinates, so we must use them eventually
- dot product
  - product of magnitudes and cosine of angle between
    * or sum of product of coordinates along each axes
  - when dot product is 0, vectors are perpendicular
- 2d cross product
  - the cross product we normally think of only makes sense in 3d
  - our 2d cross product is just a vector of same magnitude, perpendicular to original
  - unary operation
  - represented by v superscript perpendicular-sign
  - $vp = (-v.y, v.x)$
  - v dot product with (v cross product) == 0
- there are two kinds of transformations
  - conformal:
    * preserves angles
    * translation, rotation, uniform scaling
  - affine
    * can be represented by matrix multiplication
    * TODO is affine transform a superset of conformal transform?
    * translation, rotation, uniform/non-uniform scaling, shear
- translation
  - add a vector to every point
- uniform scaling
  - scale about a point (about an origin) by a scale factor
  - the point (origin) about which you scale will be unaffected by the scaling
  - the farther something is from the point (origin), the more it's position will change
- non-uniform scaling
  - same as uniform scaling, but you now have a vector that you're scaling along
  - so take the vector from transform-origin to point, find parallel to transform vector, and scale that
  - scaling along a vector is not the same as scaling along the x and y components of that vector separately
- rotation
  - q = vector from transform-origin to p
  - new point is transform-origin + linear combination of q and q-cross determined by sin and cos of theta
- shear
  - not the same as non-uniform scaling
  - move point in direction of v, proportional to distance to o perpendicular to v
- reflection

- TODO do we need to know this?

- matrix representation
  - compact
  - allows multiple transforms to be composed to single matrix (efficient)
  - if you have 3 points and those 3 points after some transformation, you can solve for the transformation
    * TODO assuming it is an affine transform?
    * TODO do we need to know how to solve that on the exam?
- TODO how much of the transformation equations do we need to know?

fractals and iterated function systems

- affine transform fractal is defined by set of contractive transformations
- contractive transform: transform F is contractive if for any two compact sets $X1, X2$, the distance between them is less after transforming them
  - that is, $D(F(X1), F(X2)) < D(X1, X2)$
- hausdroff distance:
  - if two sets are equal, their distance is 0
  - distance of a,b is same as distance b,a
  - hausdroff distance is the maximum distance of a point in one set to the closest point in the other
- attractor: shape that fractal approaches after a large (ideally infinite) number of iterations
  - if transforms are contractive, attractor is independent of starting point(s)
- fractal tennis:
  - algorithm to draw fractal by randomly applying transforms to the same point
    * but need to iterate point for a few hundred iterations first to get it into the attractor
  - resulting fractal is not perfect
  - can be made better by weighting fractal transform random choice by area
    * difficult to calculate the area of a transform (TODO do you just guess?)
- condensation set: basically a thing you add in at every iteration
  - allows shape to build on itself
- fractal dimension
  - like spatial dimension, but for fractals
  - $dim = -log(\#transformations)/log(scalefactor)$
- fractal curves can have infinite length but enclose finite surface area
  - and that's fine
  - fractal paint bucket would not work because paint atoms have finite size

transformations in 3D

- very similar to transformations in 2d
- things that are the same:
  - dot product, translation, uniform/non-uniform scaling
- cross product
  - now is binary operator
  - produces third vector that is perpendicular to both input vectors
  - magnitude: product of magnitudes and sine of angle between
    * mag represents area of 4-sided polygon formed by the two vectors
  - uses special matrix called _ (underscore)
    * v cross _ = put components of v in special places
    * then (v cross _) cross w == v cross w
- rotation
  - input: axis to rotate about (specified as point and unit vector), theta to rotate
  - thus you're rotating in the plane that is perpendicular to the axis
  - component of q parallel to axis does not change, perpendicular component is rotated (in plane)
- mirror image
  - the same as non-uniform scaling with $a = -1$
  - reflect about plane formed by normal vector v and point o
- orthogonal projection
  - flatten things straight down onto plane
- perspective transformation
  - flatten things onto plane, but as if seen by an observation point e (an eye)
  - not defined for vectors (depends on where the vector is how much it gets scaled by)
  - **not an affine transformation!**
  - therefore you need to use the bottom row of the matrix also
  - the final 3d location is found by taking the normal 3d output and dividing by the 4th element (a scalar)
- hierarchical animation
  - split body into components joined by joints
  - each joint has a transformation associated with it, so you can apply the transformations corresponding to what position of components you want, and then render that
- skeletal animation
  - skeleton inside model mesh has hierarchical animation stuff
  - every vertex on mesh has a list of weights of how it's position depends on transformations of bones
  - thus allowing mesh to deform like the skeleton does
- OpenGL matrices
  - view, model, projection, viewport

  - view: position the camera
  - model: position model in world
  - projection: flatten world into 2d plane
  - viewport: transform projection into window pixel coordinates
  - opengl uses ModelView matrix
    * $ModelView = V^{-1}M = T^{-1}R^{-1}M$
    * viewer always views from origin
      · TODO looking down negative z axis?
    * then you push a matrix for the models so that they're positioned correctly

color

lighting