Compositing, Clipping, Curves

Week 3, Thu May 26

http://www.ugrad.cs.ubc.ca/~cs314/Vmay2005

Q 8: confusion on push/pop and complex operations
we will put them in bin in lab, next to extra handouts
keep it, can write on other side too for final
don’t just say “rotate 90”, say around which axis, and in which direction (CCW vs CW)
be clear on whether actions are in old coordinate frame or new coordinate frame
Q 4, Q 5: too vague
we’ll try to get H2 back tomorrow
we will put them in bin in lab, next to extra handouts
solutions will be posted
you don’t have to tell us you’re using grace days
only if you’re turning it in late and you do “not” want to use up grace days
grace days are integer quantities

Poll

- which do you prefer?
  - P4 due Fri, final Sat
  - final Thu in-class, P4 due Sat

Midterm Logistics

- Tuesday 12-12:50
  - sit spread out: every other row, at least three seats between you and next person
  - you can have one 8.5x11” handwritten one-sided sheet of paper
  - keep it, can write on other side too for final
  - calculators ok

Schedule Change

- HW 3 out Thu 6/2, due Wed 6/8 4pm

Homework 1 Common Mistakes

- Q4, Q5: too vague
  - don’t just say “rotate 90”, say around which axis, and in which direction (CCW vs CW)
  - be clear on whether actions are in old coordinate frame or new coordinate frame
- Q8: confusion on push/pop and complex operations
  - wrong: object drawn in wrong spot!
    - glPushMatrix();
    - gllTranslatex (a.3);
    - gllRotate (4);
    - draw things
    - glPop();
  - correct: object drawn in right spot
    - glPushMatrix();
    - gllTranslatex (a.3);
    - gllRotate (4);
    - draw things
    - glPop();
  - both: nice modular function that doesn’t change modelview matrix
Midterm Topics
- H1, P1, H2, P2
- first three lectures
- topics
  - Intro, Math Review, OpenGL
  - Transformations I/II/III
  - Viewing, Projections I/II

Reading: Today
- FCG Chapter 11
  - pp 209-214 only: clipping
- FCG Chap 13
- RB Chap Blending, Antialiasing, ...
  - only Section Blending

Reading: Next Time
- FCG Chapter 7

Errata
- p 214
  - \( t(p) > 0 \) is “outside” the plane
- p 234
  - For quadratic Bezier curves, \( N=3 \)
  - \( w_i^N(t) = \frac{(N-1)!}{(i!)(N-i-1)!} \)...

Review: Illumination
- transport of energy from light sources to surfaces & points
- includes direct and indirect illumination

Review: Light Sources
- directional/parallel lights
  - point at infinity: \((x,y,z,0)^T\)
- point lights
  - finite position: \((x,y,z,1)^T\)
- spotlights
  - position, direction, angle
- ambient lights
Review: Light Source Placement
- geometry: positions and directions
- standard: world coordinate system
  - effect: lights fixed wrt world geometry
- alternative: camera coordinate system
  - effect: lights attached to camera (car headlights)

Review: Reflectance
- specular: perfect mirror with no scattering
- gloss: mixed, partial specularity
- diffuse: all directions with equal energy
 specular + glossy + diffuse = reflectance distribution

Review: Reflection Equations
\[ I_{\text{diffuse}} = k_d I_{\text{light}} (n \cdot l) \]
\[ I_{\text{specular}} = k_s I_{\text{light}} (v \cdot r) n_{\text{shiny}} \]
\[ 2 (N \cdot L) - L = R \]

Review: Reflection Equations 2
- Blinn improvement
  \[ I_{\text{specular}} = k_s I_{\text{light}} (h \cdot n) n_{\text{shiny}} \]
  \[ h = (l + v)/2 \]
- full Phong lighting model
  - combine ambient, diffuse, specular components
  \[ I_{\text{total}} = k_s I_{\text{ambient}} + \sum_{i=1}^{N_{\text{lights}}} k_d I_i (n \cdot l_i) + k_s (v \cdot r_i) n_{\text{shiny}} \]
  - don’t forget to normalize all vectors: n,l,r,v,h

Review: Lighting
- lighting models
  - ambient
    - normals don’t matter
  - Lambert/diffuse
    - angle between surface normal and light
  - Phong/specular
    - surface normal, light, and viewpoint

Review: Shading Models
- flat shading
  - compute Phong lighting once for entire polygon
- Gouraud shading
  - compute Phong lighting at the vertices and interpolate lighting values across polygon
- Phong shading
  - compute averaged vertex normals
  - interpolate normals across polygon and perform Phong lighting across polygon
Correction/Review: Computing Normals

- per-vertex normals by interpolating per-facet normals
- OpenGL supports both
- computing normal for a polygon
  - three points form two vectors
  - pick a point
  - vectors from
    - A: point to previous
    - B: point to next
  - \( \text{A} \times \text{B} \): normal of plane direction
  - normalize: make unit length
  - which side of plane is up?
    - counterclockwise
    - point order convention

Review: Non-Photorealistic Shading

- cool-to-warm shading
  \[ k_e = \frac{1 + n_1}{2} c - \frac{1 - n_1}{2} c_e \]
- draw silhouettes: if \( \langle \text{e} \times \text{n}_0 \rangle \leq 0 \)
- draw creases: if \( \langle \text{n}_0 \times \text{n}_1 \rangle \leq \text{threshold} \)

End of Class Last Time

- use version control for your projects!
  - CVS, RCS
- partially work through problem with lighting

### Compositing

- how might you combine multiple elements?
- foreground color \( \text{A} \), background color \( \text{B} \)

#### Premultiplying Colors

- specify opacity with alpha channel: \( \langle r, g, b, a \rangle \)
  - \( a=1 \): opaque, \( a=0.5 \): translucent, \( a=0 \): transparent
- \( \text{A} \) over \( \text{B} \)
  \[ \text{C} = \alpha \text{A} + (1-\alpha) \text{B} \]
- but what if \( \text{B} \) is also partially transparent?
  - \( \text{C} = \alpha \text{A} + (1-\alpha) \beta \text{B} + \alpha \beta \text{A} + \alpha(1-\beta) \text{B} \)
  - \( \gamma = \beta + (1-\beta) \gamma + \alpha - \gamma \beta \)
  - 3 multiplies, different equations for alpha vs. RGB
- premultiplying by alpha
  - \( \text{C} = \gamma \text{C} \), \( \text{B} = \beta \text{B} \), \( \text{A} = \alpha \text{A} \)
  - \( \text{C}' = A' + \gamma A' - \alpha B' \)
  - \( \gamma = 1 + \alpha - \gamma B' \)
  - 1 multiply to find \( \text{C} \), same equations for alpha and RGB
Clipping

Next Topic: Clipping
- we've been assuming that all primitives (lines, triangles, polygons) lie entirely within the viewport
- in general, this assumption will not hold:

Why Clip?
- bad idea to rasterize outside of framebuffer bounds
- also, don't waste time scan converting pixels outside window
  - could be billions of pixels for very close objects!

Clipping
- analytically calculating the portions of primitives within the viewport

Line Clipping
- 2D
  - determine portion of line inside an axis-aligned rectangle (screen or window)
- 3D
  - determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
  - simple extension to 2D algorithms
Clipping

- naïve approach to clipping lines:
  for each line segment
  for each edge of viewport
    find intersection point
    pick “nearest” point
    if anything is left, draw it
  what do we mean by “nearest”?  
  how can we optimize this?

Trivial Accepts

- big optimization: trivial accept/rejects
  - Q: how can we quickly determine whether a line segment is entirely inside the viewport?
  - A: test both endpoints

Trivial Rejects

- Q: how can we know a line is outside viewport?
  - A: if both endpoints on wrong side of same edge, can trivially reject line

Clipping Lines To Viewport

- combining trivial accepts/rejects
  - trivially accept lines with both endpoints inside all edges of the viewport
  - trivially reject lines with both endpoints outside the same edge of the viewport
  - otherwise, reduce to trivial cases by splitting into two segments

Cohen-Sutherland Line Clipping

- outcodes
  - 4 flags encoding position of a point relative to top, bottom, left, and right boundary
  - $OC(p1)=0010$
  - $OC(p2)=0000$
  - $OC(p3)=1001$
  - assign outcode to each vertex of line to test
  - line segment: $(p1,p2)$
  - trivial cases
    - $OC(p1)==0$ & $OC(p2)==0$
      - both points inside window, thus line segment completely visible (trivial accept)
    - $(OC(p1) & OC(p2))==0$
      - there is (at least) one boundary for which both points are outside (same flag set in both outcodes)
      - thus line segment completely outside window (trivial reject)
Cohen-Sutherland Line Clipping
- if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- pick an edge that the line crosses (how?)
- intersect line with edge (how?)
- discard portion on wrong side of edge and assign outcode to new vertex
- apply trivial accept/reject tests; repeat if necessary

Viewport Intersection Code
- \((x_1, y_1), (x_2, y_2)\) intersect vertical edge at \(x_{\text{right}}\)
  - \(y_{\text{intersect}} = y_1 + m(x_{\text{right}} - x_1)\)
  - \(m = (y_2 - y_1)/(x_2 - x_1)\)
- \((x_1, y_1), (x_2, y_2)\) intersect horiz edge at \(y_{\text{bottom}}\)
  - \(x_{\text{intersect}} = x_1 + (y_{\text{bottom}} - y_1)/m\)
  - \(m = (y_2 - y_1)/(x_2 - x_1)\)

Cohen-Sutherland Discussion
- use opcodes to quickly eliminate/include lines
- best algorithm when trivial accepts/rejects are common
- must compute viewport clipping of remaining lines
- non-trivial clipping cost
- redundant clipping of some lines
- more efficient algorithms exist
Line Clipping in 3D
- approach
- clip against parallelepiped in NDC
- after perspective transform
- means that clipping volume always the same
  - xmin=ymin= -1, xmax=ymax= 1 in OpenGL
- boundary lines become boundary planes
- but outcodes still work the same way
- additional front and back clipping plane
  - zmin = -1, zmax = 1 in OpenGL

Polygon Clipping
- objective
  - 2D: clip polygon against rectangular window
  - or general convex polygons
  - extensions for non-convex or general polygons
  - 3D: clip polygon against parallelepiped

Polygon Clipping
- not just clipping all boundary lines
- may have to introduce new line segments

Why Is Clipping Hard?
- what happens to a triangle during clipping?
- possible outcomes:
  - triangle ⊆ triangle
  - triangle ⊆ quad
  - triangle ⊆ 5-gon
- how many sides can a clipped triangle have?

How Many Sides?
- seven…

Why Is Clipping Hard?
- a really tough case:
Why Is Clipping Hard?

- a really tough case:

![concave polygon ⊖ multiple polygons]

Polygon Clipping

- classes of polygons
  - triangles
  - convex
  - concave
  - holes and self-intersection

Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped

![Sutherland-Hodgeman Clipping](image)

Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped

![Sutherland-Hodgeman Clipping](image)
Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped

Sutherland-Hodgeman Algorithm

- input/output for algorithm
  - input: list of polygon vertices in order
  - output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)
- basic routine
  - go around polygon one vertex at a time
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?
Clipping Against One Edge

- \( p[i] \) inside: 2 cases

\[
\begin{align*}
\text{inside} & \quad \text{outside} \\
p[i-1] & \quad \quad \quad \quad p[i] \\
p[i] & \quad \quad \quad \quad p[i-1]
\end{align*}
\]

Clipping Against One Edge

- \( p[i] \) outside: 2 cases

\[
\begin{align*}
\text{inside} & \quad \text{outside} \\
p[i-1] & \quad \quad \quad \quad p[i] \\
p[i] & \quad \quad \quad \quad p[i-1]
\end{align*}
\]

Clipping Against One Edge

- \( p[i] \) inside: 2 cases

\[
\begin{align*}
\text{inside} & \quad \text{outside} \\
p[i-1] & \quad \quad \quad \quad p[i] \\
p[i] & \quad \quad \quad \quad p[i-1]
\end{align*}
\]

Sutherland-Hodgeman Example

\[
\begin{align*}
p[0] & \quad p[1] \\
\end{align*}
\]

Sutherland-Hodgeman Discussion

- similar to Cohen/Sutherland line clipping
- inside/outside tests: outcodes
- intersection of line segment with edge: window-edge coordinates
- clipping against individual edges independent
- great for hardware (pipelining)
- all vertices required in memory at same time
- not so good, but unavoidable
- another reason for using triangles only in hardware rendering

Sutherland/Hodgeman Discussion

- for rendering pipeline:
  - re-triangulate resulting polygon
  (can be done for every individual clipping edge)
Curves

Parametric Curves
- parametric form for a line:
  \[ x = x_0 + (1-t)x_1 \]
  \[ y = y_0 + (1-t)y_1 \]
  \[ z = z_0 + (1-t)z_1 \]
- \( x, y \) and \( z \) are each given by an equation that involves:
  - parameter \( t \)
  - some user specified control points, \( x_0 \) and \( x_1 \)
  - this is an example of a parametric curve

Splines
- a spline is a parametric curve defined by control points
  - term “spline” dates from engineering drawing, where a spline was a piece of flexible wood used to draw smooth curves
  - control points are adjusted by the user to control shape of curve

Splines - History
- draftsman used ‘ducks’ and strips of wood (splines) to draw curves
- wood splines have second-order continuity, pass through the control points
- a duck (weight)
- ducks trace out curve

Hermite Spline
- hermite spline is curve for which user provides:
  - endpoints of curve
  - parametric derivatives of curve at endpoints
  - parametric derivatives are \( \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt} \)
- more derivatives would be required for higher order curves

Hermite Cubic Splines
- example of knot and continuity constraints
- \( \nabla p_1 \)
- \( r = 1 \)
- \( p_2 \)

Hermite Specification
Hermite Spline (2)

- say user provides \( x_0, x_1, x'_0, x'_1 \)
- cubic spline has degree 3, is of the form:
  \[ s = a t^3 + b t^2 + c t + d \]
  for some constants \( a, b, c \) and \( d \) derived from the control points, but how?
- we have constraints:
  - curve must pass through \( x_0 \) when \( t=0 \)
  - derivative must be \( x'_0 \) when \( t=0 \)
  - curve must pass through \( x_1 \) when \( t=1 \)
  - derivative must be \( x'_1 \) when \( t=1 \)

Hermite Spline (3)

- solving for the unknowns gives
- rearranging gives

\[
\begin{align*}
  a &= -2x_0 + 2x_1 + x'_0 + x'_1 \\
  b &= 3(x_1 - 3x_0 - x'_0 - 2x'_1) \\
  c &= x'_0 \\
  d &= x'_1
\end{align*}
\]

Basis Functions

- a point on a Hermite curve is obtained by multiplying each control point by some function and summing
- functions are called basis functions

Sample Hermite Curves

Splines in 2D and 3D

- so far, defined only 1D splines:
  \[ x = f(t; x_0, x_1, x'_0, x'_1) \]
- for higher dimensions, define control points in higher dimensions (that is, as vectors)

\[
\begin{bmatrix}
  x \\
  y \\
  z
\end{bmatrix} =
\begin{bmatrix}
  x_0 & x_1 & x'_0 & x'_1 \\
  y_0 & y_1 & y'_0 & y'_1 \\
  z_0 & z_1 & z'_0 & z'_1
\end{bmatrix}
\begin{bmatrix}
  -2 & 3 & 0 & 0 \\
  2 & -3 & 0 & 1 \\
  1 & -1 & 0 & 0 \\
  1 & -2 & 1 & 0 \\
\end{bmatrix} t^3
\]

Bézier Curves

- similar to Hermite, but more intuitive definition of endpoint derivatives
- four control points, two of which are knots
Bézier Curves

- derivative values of Bezier curve at knots dependent on adjacent points

\[ \nabla p_1 = 3(p_2 - p_1) \]
\[ \nabla p_3 = 3(p_4 - p_3) \]

Bézier vs. Hermite

- can write Bezier in terms of Hermite
  - note: just matrix form of previous

\[ \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \frac{dx}{dt} & \frac{dy}{dt} \\ \frac{d^2x}{dt^2} & \frac{d^2y}{dt^2} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \]

Bézier Basis, Geometry Matrices

- Now substitute this in for previous Hermite

\[ \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \\ c_1 & c_2 \\ d_1 & d_2 \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \]

- but why is \( M_{\text{Bezier}} \) a good basis matrix?

Bézier Blending Functions

- look at blending functions

- family of polynomials called order-3 Bernstein polynomials
  - \( C(3, k) t^k (1-t)^{3-k}; 0 \leq k \leq 3 \)
  - all positive in interval \([0,1]\)
  - sum is equal to 1

\[ p(t) = \begin{bmatrix} (1-t)^3 & \frac{3(1-t)^2}{2} & \frac{3(1-t)}{2} & t^3 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} \]

Bézier Blending Functions

- every point on curve is linear combination of control points
- weights of combination are all positive
- sum of weights is 1
- therefore, curve is a convex combination of the control points
Bézier Curves
- curve will always remain within convex hull (bounding region) defined by control points

Bézier Curves
- interpolate between first, last control points
- 1st point’s tangent along line joining 1st, 2nd pts
- 4th point’s tangent along line joining 3rd, 4th pts

Comparing Hermite and Bézier

Rendering Bezier Curves: Simple
- evaluate curve at fixed set of parameter values, join points with straight lines
- advantage: very simple
- disadvantages:
  - expensive to evaluate the curve at many points
  - no easy way of knowing how fine to sample points, and maybe sampling rate must be different along curve
  - no easy way to adapt: hard to measure deviation of line segment from exact curve

Rendering Bezsiers: Subdivision
- a cubic Bezier curve can be broken into two shorter cubic Bezier curves that exactly cover original curve
- suggests a rendering algorithm:
  - keep breaking curve into sub-curves
  - stop when control points of each sub-curve are nearly collinear
  - draw the control polygon: polygon formed by control points

Comparing Hermite and Bezier

demo: www.siggraph.org/education/materials/HyperGraph/modeling/adobe/demoprog/curve.html
Sub-Dividing Bezier Curves

- step 1: find the midpoints of the lines joining the original control vertices. Call them $M_{01}$, $M_{12}$, $M_{23}$.

- step 2: find the midpoints of the lines joining $M_{01}$, $M_{12}$ and $M_{12}$, $M_{23}$. Call them $M_{012}$, $M_{123}$.

- step 3: find the midpoint of the line joining $M_{012}$, $M_{123}$. Call it $M_{0123}$.

Curve $P_0, M_{01}, M_{012}, M_{0123}$ exactly follows original from $t=0$ to $t=0.5$.

Curve $M_{0123}, M_{123}, M_{23}, P_3$ exactly follows original from $t=0.5$ to $t=1$.

continue process to create smooth curve.

de Casteljau’s Algorithm

- can find the point on a Bezier curve for any parameter value $t$ with similar algorithm.
- for $t=0.25$, instead of taking midpoints take points 0.25 of the way.

demo: www.saltire.com/applets/advanced_geometry/spine/spline.htm
Longer Curves

- A single cubic Bezier or Hermite curve can only capture a small class of curves
- One solution is to raise the degree
- Control is not local, one control point influences entire curve
- Better solution is to join pieces of cubic curve together into piecewise cubic curves

Piecewise Bezier: Continuity Problems

demo: [www.cs.princeton.edu/~min/cs426/jar/bezier.html](http://www.cs.princeton.edu/~min/cs426/jar/bezier.html)

Continuity

- When two curves joined, typically want some degree of continuity across knot boundary
- C0, “C-zero”, point-wise continuous, curves share same point where they join
- C1, “C-one”, continuous derivatives
- C2, “C-two”, continuous second derivatives

Geometric Continuity

- Derivative continuity is important for animation
- If object moves along curve with constant parametric speed, should be no sudden jump at knots
- For other applications, tangent continuity suffices
- Requires that the tangents point in the same direction
- Referring to as $G^1$ geometric continuity
- Curves could be made $C^1$ with a re-parameterization
- $G^2$ geometric version of $C^2$ is based on curves having the same radius of curvature across the knot

Achieving Continuity

- Hermite curves
  - User specifies derivatives, so $C^1$ by sharing points and derivatives across knot
- Bezier curves
  - They interpolate endpoints, so $C^0$ by sharing control pts
  - Introduce additional constraints to get $C^1$
    - Parametric derivative is a constant multiple of vector joining first/last 2 control points
    - So $C^1$ achieved by setting $P_{i-2} = P_{i+1}$ and making $P_{i+1} - J$ and $P_{i-1} - J$ collinear, with $J \parallel P_{i-1} - P_{i+1}$
    - $C^2$ comes from further constraints on $P_{i} - P_{i-1}$ and $P_{i+1} - P_{i}$
  - Leads to...

B-Spline Curve

- Start with a sequence of control points
- Select four from middle of sequence $(p_{i-2}, p_{i-1}, p_{i}, p_{i+1})$
- Bezier and Hermite goes between $p_{i-2}$ and $p_{i+1}$
- B-Spline doesn’t interpolate (touch) any of them but approximates the going through $p_{i+1}$ and $p_{i}$
B-Spline

- by far the most popular spline used
- $C_0$, $C_1$, and $C_2$ continuous

[Image: www.siggraph.org/education/materials/HyperGraph/modeling/splines/demoprog/curve.html]

Project 3

- bumpy plane
  - vertex height varies randomly by 20% of face width
  - world coordinate light, camera coord light
  - regenerate terrain
  - toggle colors
  - six triangles around a vertex
  - [demo]

Project 3: Normals

- calculate once (per terrain)
  - per-face normals
  - then interpolate for per-vertex
  - use when drawing
    - specify interleaved with vertices
    - explicitly drawing normals
      - bristles at vertices
      - visual debugging

Project 3: Data Structures

- suggestion: 100x100x4 array for vertex coords
- colors?
- normals? per-face, per-vertex

Project 4

- create your own graphics game or tutorial
- required functionality
  - 3D, interactive, lighting/shading
  - texturing, picking, HUD
- advanced functionality pieces
  - two for 1-person team
  - four for 2-person team
  - six for 3-person team
P4: Advanced Functionality
- (new) navigation
- procedural modelling/textures
  - particle systems
- collision detection
- simulated dynamics
- level of detail control
- advanced rendering effects
- whatever else you want to do
  - proposal is a check with me

P4 Proposal
- due Wed 1 Jun 4pm
  - either electronic handin, or box handin for hardcopy
  - short (< 1 page) description
    - how game works
    - how it will fulfill required functionality
    - advanced functionality
    - must include at least one annotated screenshot mockup sketch
      - hand-drawn scanned or using computer tools

P4 Writeup
- what: a high level description of what you’ve created, including an explicit list of the advanced functionality items
- how: mid-level description of the algorithms and data structures that you’ve used
- howto: detailed instructions of the low-level mechanics of how to actually play (keyboard controls, etc)
- sources: sources of inspiration and ideas
  - especially any source code you looked at for inspiration on the Internet
- include screen shots with handin for HOF eligibility

P4 Grading
- final project due 11:59pm Fri Jun 17
  - face to face demos again
  - I will be grading
  - grading
    - 50% base: required functions, gameplay, etc
    - 50% advanced functionality
    - buckets, tentative mapping
      - zero = 0
      - minus = 40
      - check-minus = 60
      - check = 80
      - check-plus = 100
      - plus 105