Nandini Ramakrishnan

Jake's Notes

//**/
/
//******************** First Day - January 7th, 2019 ***********************//
//**//

- 12:18 - activity on the screen detected
  - On the screen: "CS 3451 - COMPUTER GRAPHICS - INSTRUCTOR GREG TURK"

- "Okay, so as you hopefully know, this class is CS 3451 - an intro course to computer graphics"
- Instructor: Professor Greg Turk!
  - "I work in the school of interactive computing, which most students don't know or care about"
  - Specifically, he works on computer graphics applications, so this is his own favorite child in the course curriculum :)

- Okay, some intro syllabus stuff:
  - This course was designed as a first class for computer graphics, so if you haven't taken a CG class before, great! Otherwise, this class might be a bit basic for the experienced folks among us
- Some things I expect you to already know:
  - BASIC linear algebra knowledge (vectors, matrices, matrix multiplication, inverses, etc.)
  - Familiarity with Java/a C-like programming languages
    - "We'll be using Processing in this course, which is a sort of wrapper-extension to Java"
  - Things I hope that you WILL get to know: output devices, rasterization, texturing, raytracing, color, lighting...etc.
- There'll be 5 medium-sized programming assignments (and 1 warm-up assignment) that make up 70% of the grade, along with a midterm and a final
  - These are all individual assignments - feel free to get help from me or the TAs, but I think you'll learn best by going it on your own. That means NO code sharing, using code from books/github, etc.
    - As per usual, honor code violations result in a 0, and it breaks his heart everytime
- We DO have a textbook for this course - and it's pretty good! "Fundamentals

of Computer Graphics", by Peter Shirley. Any edition from 2-4 is fine, and optional reading assignments from it will be pretty frequent.
    - You can probably get through this class without it, but I'd highly recommend you do get it

- How do you succeed in this course? Really, just 2 pieces of usual advice from other classes:
    - Start the projects EARLY - start them a day or two after it's released, not the night of. Give yourself that buffer if something goes wrong
    - Get help if you feel like you're struggling. Some of this stuff might be confusing the first time you hear it, and that's okay! We're here to help!

- Student question: "wat bout blender pr0gram?"
    - Professor Turk: Blender's a really nice open source CG modeling program that uses many of the techniques we'll discuss here, but that's an example of a 3D computer graphics application - we're going to be learning the algorithms and mathematics BEHIND how those graphics work and how applications like Blender are created, not using them directly

- ...okay, that's most of the administrivia and a broad outline of what we're doing this semester.
- With that said, let's talk about the first example programming assignment, and why it exists:
    - This project is just to give you guys an early heads-up on what the programming/difficulty in this course looks like, so you can decide right away if this course is appropriate for you. Get started on it right away; if it's easy for you, great! If it takes some work, consider talking to me about whether this course is right for you

- ...and with that, let's start talking about the first big topic in the course: RASTER IMAGES
    - As most of you know, computer screens are composed of PIXELS (short for "picture element" - "I don't know how the X got in there"), arranged in rows (known as SCANLINES) and each with its own assigned color
        - Typically, this color is given as an RGB (red-green-blue) value of 3 numbers in the range 0 to 255 - e.g., [255, 0, 0] is bright red, [0, 255, 255] would be cyan, etc.
        - There ARE a few less common ways of specifying color, but we'll talk about that in due time
    - These pixels are arranged in a rectangular grid, according to some

coordinate system

- In Processing, the coordinates are given as 2D (x,y) coordinates, with X on the horizontal axis and Y on the vertical
  - Annoyingly, instead of the origin being in the lower-left corner of the screen, it's in the UPPER-LEFT, with the "x" coordinate being the distance to the right and "y" being the distance down from the top
    - Apparently, there's some historical reason for this due to how early TV transmissions were displayed, but still: annoyance
- The size of the grid in Processing can be specified with the "size" command, e.g:

```
size(400, 400); // opens a 400x400 pixel window
```

- EVERY Processing program needs 2 functions:
  - void setup() {...}, which handles any initialization
  - void draw() {...}, which is repeatedly called every frame, usually to redraw the window

- A few other useful Processing commands:

```
rect(100, 200, width, height); // width-by-height rectangle with the upper-left corner at (100, 200)
ellipse(200, 200, width, height); // creates an ellipse

background(r, g, b); // fills the window w/ the given background color
stroke(r, g, b); // sets the outline color
fill(r, g, b); // sets the fill color
noStroke(); //turns off outlines
noFill(); //turns off the fill color
```

- Other Processing things:
  - "width" and "height" are predefined variables that refer to the width/height (in pixels) of the window
  - "mouseX" / "mouseY" are ditto for the mouse position coordinates

- Okay, but what if we really didn't like Processing's coordinate system?
  - Let's say, for instance, we want to specify coordinates using the regular origin-in-the-middle graph system from middle school, w/ the X-range from -1 to 1 - how would we do this?

- Well, the current Processing X-coordinates go from 0 to screenWidth, so we need to map from [-1, 1] to [0, screenWidth]
    - Since we want to go from positive/negative numbers to all positive, let's just shift our coordinates to all positive by adding 1; i.e,

        [-1, 1] -> [0, 2]

    - From there, we can just multiply by screenWidth/2 to get the screen's coordinates!
        - "Some of you might be comfortable doing this in 1 step with linear algebra, but I like to split it up in my head"
- Similarly, to go from the screen X coordinates to our [-1, 1] system, we'd subtract "screenWidth/2" from the coordinates to shift the "center" of the range, then multiply it by (1 / (screenWidth/2)) to finish it off

- "Now, remember the unit circle for sines and cosines in Trig? Remember, for a point on that unit circle, the X-coordinate is cos(theta), and the Y-coordinate is sin(theta)"
    - Importantly, Processing uses radians for sine/cosine angles; don't use degrees, or things'll get real weird fast

- ...and with that, you should have all the Processing and math you need to finish the intro project - until then, Salud!

//**/
/
//********** Basic 2D Transformations - January 9th, 2019 **********//
//**//

- Well, I've been evicted from my original seat (kids these days have no respect for 1-day spontaneous claims of ownership)
    - ...annnnnd it looks like I'll have to figure out how to type matrices pretty quickly
- In other news, Professor Turk's office hours'll be on Wednesday/ Thursday, 1:10-2:10, somewhere in Clough
    - The TA office hours are still being sorted out

- Okay, today we're going to start getting into some VERY basic linear algebra

review
- "I know we've all taken a class in this, but if you're like me, you need some refreshers on this stuff from time to time"

- So, let's start with matrix multiplication
  - Let's say we've got two 2x2 matrices A and B; how do we multiply them?
    - Well, we multiply the respective row from the 1st matrix with the column from the 2nd! For instance,

```
[3 -1] X [1 2] = [(3*1 + -1*2) (3*2 + -1*0)] = [1 6]
[2  1]   [2 0]   [(2*1 + 1*2)   (2*2 + 1*0)]   [4 4]
```

  - Great! But what if we switch the order, so it's BxA?
    - Well, it'll be different! AB does NOT equal BA in linear algebra; matrix multiplication isn't commutative!
      - "SOME special matrices, like the identity matrix, do commute - but in general, don't count on it"

- What about vectors?
  - Well, let's start with the dot product between 2 vectors "v1" and "v2":

```
[2 1] * |2| = 2*2 + 1*3 = 4 + 3 = 7
        |3|
```

    - "I'll often write the two vectors this way, as a row and column vector, since it's closer to how we do matrix multiplication"
  - Now, as you might recall, a dot product of 0 means the two vectors are orthogonal/perpendicular to each other, e.g.

```
[2 1] * |-2| = 0
        | 4|
```

    - "The dot product comes up a TON in computer graphics = it's used to determine how much light falls on a surface, how rotations work, etc."

- Now, why do we care about these? Well, matrix operations are at the heart of 3 VERY common transformations we'll do on images:
  - Translations
  - Rotations
  - Scaling

- "The textbook readings you should do for this section are in the Matrices/Transformations chapter, by the way"

- So, first up is TRANSLATION: simply changing the position of an object and nothing else
  - Typically in CG, we'll have a set of points in a coordinate system that we'll use to create what we want
  - For instance, let's say we have 3 points defining a triangle at (1,1), (2,3), and (3,1) - how do we move this triangle to the right?
    - More specifically, let's say we want to move the triangle 3 units to the right and 1 unit up
    - To do this, it's simple, right? We just add 3 to all the X coordinates and 1 to all the y coordinates!

$$x' = x + dx$$
$$y' = y + dy$$

    - So, we end up with the new coordinates (4,2), (5,4), (6,2)
      - In general, assuming a bottom-left origin at (0,0), adding to X moves things right, subtracting X moves things left, adding Y moves things up and subtracting moves things down
- Now, let's write this same thing in terms of vectors
  - We have 2 coordinates, so we'll have a vector 2 units long
  - Let's have 2 vectors: P, for the original point, and T for the displacement amount
    - Using this, the new position "P'" will be:

$$P' = P + T = \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} dx \\ dy \end{vmatrix}$$

- Now, SCALING means that we're changing the size of the objects
  - Let's say we have the same triangle and we want to double it in size - what'll we do?
    - Well, we'll just multiply the coordinates! (specifically, this is "uniform scaling", since we're scaling all the dimensions by the same amount)

$$x' = sx*x = 2*x$$
$$y' = sy*y = 2*y$$

- However, this'll scale relative to the ORIGIN at 0,0, so be aware of that - the triangle will also be scooted over away from the origin a bit
    - How would we fix that? Well, we could move the triangle over to the origin before we scale it...in other words, we could TRANSLATE it...
        - Keep that thought, of combining operations, in the back of your mind
- So, expressing this in terms of vectors and matrices:
    - We'll have a 2x2 matrix "S" that has our scaling factors, AND we'd have the original point in the "P" vector
    - With that, our new point "P'" is:

$$P' = SP = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} * \begin{vmatrix} x \\ y \end{vmatrix} = \begin{vmatrix} sx*x \\ sy*y \end{vmatrix}$$

- And finally, ROTATION means that we're...well, rotating the point about the origin
    - "Rotating things isn't hard, it just seems a bit more quirky than the others since we're dealing with sines and cosines"
        - Also, REMEMBER: by default, we'll be rotating things counter-clockwise (because of how the trig functions work)
    - So, if we rotate a point by some angle "theta" about the origin, we'll get:

$$x' = x*\cos(\theta) - y*\sin(\theta)$$
$$y' = x*\sin(\theta) + y*\cos(\theta)$$

    - "Remember: sines and cosines are just positions on the unit circle"

- More succinctly, we can rewrite these equations as a matrix:
    - R, the rotation matrix, is:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

    - Plugging in our point vector, then, we end up with:

$$P' = RP = \begin{bmatrix} \cos & -\sin \\ \sin & \cos \end{bmatrix} * \begin{vmatrix} x \\ y \end{vmatrix} = \begin{vmatrix} x*\cos - x*\sin \\ x*\sin + y*\cos \end{vmatrix}$$

- Again, maybe there'll be times you don't want to rotate about the origin...and for those times, we'll be shifting our object so that the origin is where we want to rotate around

- So, rotation and scaling are both done by multiplying an NxN matrix by an N-order vector, but translation is done by adding 2 vectors - it's the odd one out!
  - How do we fix this and get them all to play nice together? We'll use something called HOMOGENOUS COORDINATES:
    - Instead of writing our 2D points as 2D vectors, let's add a 1 to the bottom, like this:

      P(x,y) = [x, y, 1]

    - Now, to scale our vector, we can write our matrix S like this:

      S = [Sx]
      [Sy]
      [1]

    - Similarly, we can do the same thing for our rotation matrix...
    - ...BUT, importantly, we can NOW express our translation operation as a matrix like this:

      T = [1 0 dx]
      [0 1 dy]
      [0 0 1]

      - "2x2 matrices weren't enough to get us there (since then our translation amount would've been scaled by the x/y coordinates), but by adding that extra coordinate, we can express our translation matrix nicely!"
        - What we'll find is that, even though matrix multiplication isn't commutative, it IS associative - and by having ALL of our operations as matrix operations, we can combine ALL of our operations into a single matrix
          - The reason why this is important should get a little clearer on Friday
      - Also, note that if dx and dy are 0, we end up with the identity matrix - which is what we want!

        - In fact, ALL the matrix operations have a way of "doing nothing" by plugging in values that turn them into the identity matrix
- "Get to know these 3 matrices - we'll use them time and time again during this semester"

- Other than the big 3, there are a few less common transformations it'd be good to know about:
  - SHEAR is an operation that "bends"/shifts an object's coordinates, but only in one direction - "Imagine you have a square made of toothpicks or something, and it starts tipping over"
    - "Practically, this doesn't come up very often - probably the most common time it comes up is when you're calculating the view plane of a camera"
    - Mathematically, it looks something like this for an 'x-shear':

      $x' = x + ay$
      $y' = y$

    - For a "y-shear":

      $x' = x$
      $y' = y + by$

  - REFLECTION is when we're flipping a set of points about some line
    - As it turns out, though, this is just a special case of scaling - it works out to be the same thing as scaling by a negative value!
      - e.g., reflecting about the Y-axis (i.e. reflecting the x-coordinates) would be:

        $x' = -x$
        $y' = y$

- And with that said, we'll keep diving into 2D transformations on Friday - see you then!

//**********/
/
//********** Composing 2D Transformations- January 11th, 2019 **********//
//**********//

- The 4th row in a class...it is a thing

- Okay, so on Wednesday we talked about the 3 major types of transformations: translation, scaling, and rotation
  - Translation, sadly, worked out to adding 2 vectors instead of multiplying matrices...but then we found out that we COULD express it as a matrix multiplication using "homogenous coordinates!"
    - "This is how graphic libraries handle transformations internally, as matrix multiplications"

- Now, let's suppose we have the following problem: we have a little picture of a house (represented by Mr. Pentagon), and we want to rotate our home by 90 degrees
  - If we just apply our rotation operation, we'd rotate about the origin
- But what if we want to rotate IN PLACE, so that we're rotating around the center of our object?
  - As it turns out, OpenGL has NO idea what we mean by the "center" of our object - so how do we do this?
  - Well, if WE know where our house's center is, we can do it by moving our house to the origin, rotating it, and then moving it back
    - i.e., we're TRANSLATING it!
- So, the process would look like:
  1) Translating our house so the center is at the origin (T1)
  2) Rotating our house however we want (R)
  3) Translating the house back to its original position (T2)
     - So, in matrix form, rotating our house around a point (a,b) by 90 degrees would look like:

translate(-a,-b) -> rotate(pi/2) -> translate(a,b)

$$T1 = \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad T2 = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix}$$

- However, REMEMBER that matrix multiplications happen right-to-left - so, we have to put the first operations at the END of the matrix multiplication
  - "Think of these as functions, and the idea of function composition - the innermost function will get called first!"

- Therefore, to rotate our point P to its new home:

  P' = T2(R(T1(P))) = (T2*R*T1)*P

  - "DON'T FORGET THIS; you should always read the matrix operations right-to-left, from the vector backwards"

- Notice here, though, that our matrix multiplications are associative - and, therefore, we can combine ALL of our operations into a single matrix just by multiplying them together!
  - "So, if we have a polygon with 5 million points or something, we don't have to do 3 separate intense calculations to rotate it - we can just do 3 quick matrix operations to combine our transformations, then do it once!"
- So, the key points to remember:
  1) Operation order matters
     - If we rotated before translating, we'd end up with a completely different result!
  2) We can combine/compose multiple operations into one

- So, going back to our triangle from yesteryear, how would we translate it 2 units to the left, and then double it in size? Well, it's a translation and a scaling - SO, writing out the handy-dandy matrices for these:

  T = [1 0 -2] S = [2 0 0]
  [0 1 0] [0 2 0]
  [0 0 1] [0 0 1]

  - Combining them (don't forget to reverse the order, so translate happens first!),

  ST = [2 0 0] * [1 0 -2] = [2 0 -4]
  [0 2 0] [0 1 0] [0 2 0]
  [0 0 1] [0 0 1] [0 0 1]

  - Great! Now, let's apply our combined translation-scaling operation to each point in the tri-angled figure:

  [2 0 -4] * | 1| = | -2|
  [0 2 0] | 1| | 2|
  [0 0 1] |{1}| |{1}|

(...do for the other 2 points...)

- *had to add the 3rd coordinate to the vector to work w/ homogenous coordinates

- Doing this in the OpenGL API would look something like this:

```
glScale(2.0, 2.0);
glTranslate(-2.0, 0.0);
draw_triangle()
```

- "This looks backwards, because we have to call our functions in the SAME order as the matrix multiplications!" (i.e., the last-called operation is done first!)
  - Why doesn't the library just handle this re-ordering for us? We'll see an example on Monday where having control over this is useful

- Okay, we've established that in general, operations aren't commutative - but are ANY operations commutative with each other?
  - Well, translations DO commute with other translations (it's the same thing as commuting additions)
    - ...but translations don't commute with rotations, uniform scaling, or non-uniform scaling
      - all of these apply vice-versa as well (e.g. since translations don't commute with rotations, rotations obviously also don't commute with translations!)
  - Rotations commute with one another in 2D, and with uniform scaling!
    - ...but not with non-uniform scaling, and 3D rotations DON'T commute with one another!
      - Why doesn't it work in 3D? Imagine rotating 90 degrees around X, then Y, and vice versa - it just doesn't get you the same answer
  - Uniform scaling is really just multiplication, so it's commutative with itself AND non-uniform scaling
    - ...but not with the other stuff

- On Monday, we'll start exploring the "matrix stack" and getting a little bit into OpenGL - until then, have a great weekend!

//**/
/
//***************** Matrix Stack Basics - January 14th, 2019 ***************//
//**//

- Paper handouts? In this digital age?! MADNESS!!!
- On the intro project: ...well, it looks like luckily I did it correctly, so yay?

- Alright, first things first, let's talk about OpenGL - "the library that Processing, and many other apps, use to draw things"
  - You'll be creating your own basic implementation of this (and the matrix stack) for project 1 - so pay attention!

- Let's say we want to draw a simple line - how'll we do this? Well, we'll start off doing this:

```
glBeginShape(GL_LINES);

// place the endpoints of the line
glVertex(100.0, 400.0);
glVertex(100.0, 100.0);

glEnd(); // more important for filling the shape later on
```

  - We can use this same strategy to draw more complicated shapes; for instance, to draw two CONNECTED lines:

```
glBeginShape(GL_LINES);
glVertex(100.0, 400.0);
glVertex(100.0, 100.0);
//if we add more lines, because of the GL_LINES param, it'll draw a
DIFFERENT line between the 3rd/4th vertex, 5th/6th, etc., so we need to
put another vertex on top of the previous one to connect them
glVertex(100.0, 400.0);
glVertex(400.0, 400.0);
glEnd();
```

  - And here's how we would draw a circle (basically: little lines):

```
void unitCircle() {
   glBeginShape(GL_LINES);
   xOld = 1, yOld = 0;
   int numVertices = 20;
   for (int i = 1, i <= numVertices, i++) {
      theta = 2*PI* i / ((float) numVertices);
      int x = cos(theta), y = sin(theta);

      gtVertex(x, y);
      gtVertex(xOld, yOld);

      xOld = x;
      yOld = y;
   }
   glEnd();
}
```

- "But this unit circle would be 2 pixels across - that's tiny! You can't even see it!"
  - So, to fix this problem, we'll use something called the matrix stack

- The MATRIX STACK, basically, lets us perform different transformations on things and work with them in a translated state
  - It has a few different components:
    - The "current transformation matrix" - CTM - represents the current topmost transformation that's applied to what we're drawing
    - glPushMatrix() copies the CTM and pushes the copy onto the top of the stack
    - glPopMatrix() will pop off the top of the matrix stack - we just throw the removed matrix away
      - "You should always have 1 push for each pop - no more, no less. They're like parentheses that way; they have to be balanced."
    - glTranslate(x, y, z) - translates the component
    - glTranslate(x, y, z) - scales the component
    - glRotate(?) - ...for now, just pretend this rotates by some angle
      - "Each of these 3 commands creates a transformation matrix AND multiplies it on the right-hand side of the CTM"
    - glVertex(x, y) - position multiplied by the CTM before drawing

- e.g. to translate our line combination, we'd say:

```
glPushMatrix();
glTranslate(200, 0);
glBeginShape();
(...do our vertex placing...)
glEnd();
glPopMatrix();
```

- "Why didn't we just add to the X and Y coordinates?" In this example, we could've done that - but for more complicated operations, we want the matrix stack to handle all that for us
  - For instance:

```
void circleImage() {           // 1
   glPushMatrix();             // 2
   glTranslate(0.5, 0.5);      // 3
   glScale(0.5, 0.5);          // 4
   circle();
   glPopMatrix();

   glPushMatrix(); // Now, we're altering our ORIGINAL matrix
   stack, since popping reset the stack
   glTranslate(0.5, 0.25);
   glScale(0.5, 0.5);
   circle();
   glPopMatrix();
}
```

  - What does the stack actually look like after first calling "pushMatrix" and Translate/Scale, then?

1)
```
          -----------------
CTM       |Identity Matrix|     =>
```

2)
```
          -----------------
CTM       |Identity Matrix|
          -----------------
          |Identity Matrix|     =>
```

3)
```
CTM      -----------------
         |I * Translate  |
         -----------------
         |Identity Matrix|     =>
```

4)
```
CTM      -----------------
         |I*T * Scale    |
         -----------------
         |Identity Matrix|
```

- "So, push says that we're getting ready to transform things; it creates a fresh copy of the current state for us to work with, so that when we pop, the stack goes back to the original state!"
  - A quick side-note: Professor Turk likes giving matrix stack traces/drawing the matrix stack as exam problems, so be aware!

- So, there are 3 big uses for a matrix stack:
  1) Changing the coordinate system we're using
  2) Instantiation (object re-use, basically)
  3) Hierarchy creation (i.e. objects composed of sub-objects)
  - ...this is what our next example illustrates:
  - Let's say we want to draw a stick figure with OpenGL, with a torso, two arms, a hand for each arm, and two fingers for each hand
    - You can view this as a dependency tree, where our overall "stick figure" object is the parent of the torso, the torso is the parent of 2 arms, each of which is the parent of a hand, each of which is the parent of 2 fingers
  - "The source code for this is given in the handout/on Canvas, so make sure you look at that!"
    - Here, we have a "square" command that just draws a box centered at the origin - but by using the matrix stack, we can translate, scale, and rotate this square to draw all of our stick figure's components!
      - And since the matrix stack is persistent, we can then push, say, a rotation on the stack, then call a "drawHand()" method, and all of its drawing methods will be rotated accordingly!
        - So, we can rotate the whole component thanks to the MS without having to change our original methods!
    - e.g.:

```
    void drawHand() {
      pushMatrix();
      rotate(PI/4.0);
      scale(1.0, 0.25);
      translate(1.0, 0.0);
      drawBox();
      popMatrix();
    }
```

- Okay, we'll finish this example (and keep pushing on) on Wednesday.

```
//*************************************************************************/
/
//************** Matrix Stack (cont.) - January 16th, 2019 ******************//
//*************************************************************************//
```

- Project 1, where art thou? Whence hast thy form fled from the sullen coves of earth?

- Alright - last class, in our Frankensteinien hubris, we were creating a person - a tiny homunculus within our digital light-flashy boxes
  - As discussed, we can use the matrix stack to persistently transform the stuff we're drawing - but we only got around to creating a lonely hand
  - Today, let's continue right along with our arm-drawing method

```
    void drawArm() {
      push()
      translate(6, 0); //position the hand at the end of the arm
      hand()
      pop(); //"Fuh-git-abowt-it"
      armExtent(); //draw the actual arm
    }
    void armExtent() {
      push()
      scale(3, 5);
      square();
      pop();
    }
```

- Now for the torso, and the rest of our headless horseless man!

```
void person() {
    push(); //copy a 2nd "I" onto the top of the stack
    scale(4, 5);
    square(); //draws the torso
    pop();

    push();
    translate(4, 4.5); //move the arm from the origin to the
    top-right of the torso
    arm();
    pop();

    push();
    scale(-1, 1); //flip the x-direction, to draw the arm the other
    way
    translate(-4, 4.5);
    arm();
    pop();
}
```

- Notice that, as we were writing this, we didn't have to think too hard about what was inside the arm() method - we could just let the matrix stack and our previous transformations handle that for us!
    - Each "matrix push" is like going level down the dependency tree for these objects, and each pop is like going a level back
    - As we go downwards, we keep "accumulating" more transformations on our matrix stack, adding each transformation to the right of our equation (i.e. most recently added is the first one to be performed)
        - Then, when we draw an object, ALL the transformations on the top of the matrix stack are applied to it!
- This is the reason why the matrix stack behaves the way it does: it handles this hierarchy of commands for us as a linear set of stuff, which is great!
    - It seems unnecessarily complicated at first, but in the long run, this matrix stack saves us a ton of work

- "If you can understand this example code I gave you, draw the matrix stack for it, and understand why it works the way it does, you should be in excellent

shape with this"
  - ...and again, Professor Turk loves giving matrix stack traces on exams

- So, that's the matrix stack, but we're still stuck in Flatland. Let's try to break into the world of three dimensions
  - Let's add our 'z' coordinate, coming out of the page toward us, with the y axis still going up and the x coordinates going right
    - This is a RIGHT-HANDED COORDINATE system, probably coming from the right-hand rule, with the thumb on the x-axis and your index finger on the y-axis
      - It's perfectly okay to have the z-coordinate going into the page, of course - but that's a left-handed system
- Now, for our 3D vectors, we'll need to actually add a 1 to the bottom to make it homogenous, just like we did in the 2D case. For instance:

    P = [x, y, z, 1]

  - Similarly, to make our transformation matrices homogenous, they have to be 4x4:

    [1 0 0 dx]
    [0 1 0 dy]
    [0 0 1 dz]
    [0 0 0 1]

- The final bit of weirdness comes from rotating in 3D. Trying to rotate just around one of the axes is normal - for instance, to rotate counter-clockwise around the Z-axis (looking from the direction toward the origin):

    Rz = [cos -sin 0 0]
         [sin cos 0 0]
         [0 0 1 0]
         [0 0 0 1]

  - Similarly, for the other axes,

    Rx = [1 0 0 0]
         [0 cos -sin 0]
         [0 sin cos 0]
         [0 0 0 1]

Ry = [cos 0 sin 0]
[0 1 0 0]
[-sin 0 cos 0]
[0 0 0 1]

- "Ry probably looks weird, but the reason is because its 2x2 rotation columns are SWAPPED compared with the other matrices - think of it that way, and it makes more sense"
- "Remember, you're going to be implementing these in project 1 - so if you've got questions, fire away"

- Trying to rotate around an arbitrary axis, on the other hand, is a little weirder...but we'll get to that

- So, that's 3D, but we have a problem: all of our display devices are flat 2D panels! So, we need to somehow squash our 3D objects into a sensible 2D representation
  - To do this, we need to "project" our objects onto the screen, and there are 2 common ways of doing it:
    - PARALLEL PROJECTION (or "ORTHOGRAPHIC" projection) is where we pretend we have a rectangular "view plane" grid of pixels, representing the screen, and each of the pixels sends out a parallel "projector line" straight ahead - and then it draws whatever it hits first
      - This is nice and convenient - but it's NOT how our eyes work, sadly
    - PERSPECTIVE PROJECTION means that instead of shooting all the rays straight ahead, we instead shoot them out at an angle, emanating out from some point called the "center of projection"
      - Realistically, what this means is that objects that are farther away seem smaller - which is great!

- On Friday, we'll dig into the math behind how we can do these projections - stay well till then!

//**********/
/
//********** Projection Basics - January 18th, 2019 **********//
//**********//

- Okay, part A of project 1 has been released on Canvas - go forth, young

minions!
  - Professor Turk also apologizes for his Canvas noobiness, which is nice
  - "This first part is pretty straightforward, but it never hurts to get started bright and early"
- In the textbook right now, you should be in the Vision/Projection sections if you want to read along

---

- So, we started talking about "projections" at the end of last class, but what does that even mean?
  - Well, in geometry, projection is when we move a point/points onto some subspace (e.g. a 2D line onto the 1D X-axis)
- In PARALLEL projection, all of the "projector" lines are parallel to one another, and we draw whatever the line hits
  - ...whereas in PERSPECTIVE projection, all those lines converge to a point, more accurately mimicking how our eyes work
    - "Perspective is much more common for special effects, video games, and so on, but parallel projection is useful for stuff like drafting, where you're more concerned with how things line up"
    - These are NOT the only 2 projections in the world - there're actually quite a few - but most of the other ones are pretty special purpose, and much, much less common

- Projection is SUPER important, since it's how we get our 3D scenes out into our 2D monitors and eyeballs, but how does it actually work?
  - Let's start with parallel projection; think of it as putting a flat sheet of glass into the scene, and then us peering in through that window
  - Pretend that this VIEW PLANE of ours is on the X-Y axes, so that one flat side is facing +Z and the other's facing -Z, and its bottom axis is at Z=0
    - Now, all we have to do is project ALL of our points onto this sheet i.e. onto the Z axis), so that all the points are mapped like this:

      P(x, y, z) ----> P(x, y, 0)

    - "Are we done?" Well, not quite - we've mapped ALL the points to our view plane, but our monitor is only so big - so, we have to do a VIEWING TRANSFORMATION to figure out which points should actually be on-screen
      - Basically, we should have a certain max width/height for our view plane, and then a maximum range we can see, forming a cube/"viewing

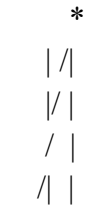
volume" that contains all the points we want to actually draw
- So, pretending the origin is at (0,0) in the bottom-left, we want to map from the points on our view plane within a certain left/right or top/bottom boundary to our actual screen's WINDOW with a width/height
- This is just mapping from one range to another! e.g.

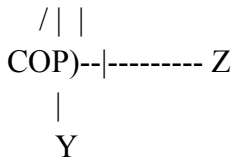
X: [left, right] ---------> [0, w]
x' = (x - left) * width/(right - left)
Y: [bottom, top] ---------> [0, height]
y' = (y - bottom) * height/(top - bottom)
Z: ...well, this is thrown away in our screen's coordinates

- You might have to deal with clipping for points that are too far away, but this is often handled at the hardware level nowadays

- "Now, I love our textbook, but it wants us to do EVERYTHING in matrices...but I'll just tell you now, parallel projection can be done with this ONE LINE of code to map x/y. It's much easier than dealing with matrix multiplication."
- Of course, matrix multiplication is MUCH more efficient, but this is significantly easier to understand

- Perspective projection, on the other hand, is all about triangles and angles and all that fun stuff - "remember it from high school geometry 80 years ago?"
- So, we'll have our virtual eyeball - the CENTER OF PROJECTION - at some point in the grid, which we'll say is at (0, 0, -1), pointed straight down the Z axis, with our view plane still at Z=0 (i.e. a bit ahead of our eye)
- Now, if we want to draw some point (x,y,z), how do we get it onto the view plane?
- Well, since our view plane is at Z=0, we know the point is p(z) away from the view plane
- So, there's a BIG triangle from our COP to the point/Z-axis, and a little triangle from the COP to the view plane





- These are SIMILAR triangles, so they'll have the same angles!
- Therefore, for the point p(x,y,z):

$y'/y = 1/|z|$

$\Rightarrow y' = y/|z|$

-Similarly,

$x' = x/|z|$

- "So, this division by Z tells us that things that are farther away will seem smaller - great!"
  - Note that this absolute value for Z would mean that anything behind us would appear in front of us - so we'll need to deal with that in the "real world"

- One thing we need to also decide for this camera is how wide our field-of-view is (FOV) - this'll decide what parts of our view plane are actually visible!
  - "You can have separate horizontal/vertical FOVs, but for simplicity's sake assume that we have a square screen right now, so they'll be the same"
  - Basically, this is the angle theta between the two extreme angles of our view plane (i.e. the total angle of our view)
    - So, our triangle would have an angle (at the COP corner) of theta/2
    - ...which means, by definition, the farthest up/down the Y axis we can see for a Z-distance of 1 is:

      $\text{maxY (or minY)} = \tan(\text{theta}/2)$

    - So, to map this from the view plane to the screen, we're doing:

      $[-\text{maxY}, \text{maxY}] \dashrightarrow [0, 2*\text{maxY}] \dashrightarrow [0, \text{height}]$

    - i.e,

y" = (y' + maxY) * height/(2*maxY)

- Adding maxY to y' so that we get rid of negatives

- "So, perspective projection is a little more complicated than parallel, but it basically boils down to 3 equations"
- In the meantime, enjoy your weekend!

//**********/
/
//********** Arbitrary Rotations - January 23rd, 2019 **********//
//**********//

- "Greg recommends using matrices to implement projection and mapping to the screen - true or false?"
  - The fools. The answer is obviously C, conveniently hidden on the backside of the paper
  - "...but really, don't use matrices. They're great for most stuff, but noooooo, they're just too annoying to deal with for a first-time projection assignment"

- So, the goal for today is to move away from just doing "axis-aligned" rotations, and instead to learn how to rotate around any vector we want
  - "The math here is actually pretty elegant, and will come in handy when we need to deal with 3D cameras"

- First off, some quick pieces of math we'll need to recover from the past:
  - VECTOR NORMALIZATION - to make a vector unit-length, we just take the vector and divide all of its components by its magnitude/length!
    - Remember, the magnitude/length of a vector is just sqrt(x^2 + y^2 + z^2 + ...)
  - Now, the dot product between 2 identical unit-length vectors is always 1
    - On the other hand, the dot product between any 2 perpendicular vectors is 0
  - ...and additionally, there's this operation you might remember called the CROSS-PRODUCT, where we multiply 2 vectors and get a 3rd vector that's orthogonal to both of the original ones
    - I won't derive it here, but to remind you, the cross-product equation looks like this:

V1 x V2 = (y1*z2 - y2*z1, z1*x2 - z2*x1, x1*y2 - x2*y1)

- Hopefully that was just review for all of you, but what're we aiming towards?
  - Well, let's suppose we're back in 2D land, and there's a unit-length vector A pointing off in the first quadrant. Here's a puzzle: if we want to rotate A to line up with the X axis, how can we do that?
    - Sure, we could do something with a rotation matrix and arctangents and so forth, but there's a really neat solution that's simpler:

1) Create another unit vector "B" perpendicular to A, such that:

B = (-ay, ax)

- Is this always perpendicular? Well, yes!

A*B = a1*b1 + a2*b2 = ax*-ay + ay*ax = 0

- In 2D, there's also another perpendicular vector that would also work - this would give us the same answer (although it might possibly flip something?)

2) Create a rotation matrix as the following:

R = [ax ay 0] = [ax ay 0]
[bx by 0] [-ay ax 0]
[0 0 1] [0 0 1]

- "For the positive and negatives, think of ax replacing cosine and ay replacing sine"
  - Why is the negative sign in the wrong place, then?
  Because this time we're rotating CLOCKWISE onto the x-axis, instead of counter-clockwise like we're used to
- So, what does this get us when we multiply A by this matrix?

R*A = [ax ay 0] |ax|
[-ax ay 0] * |ay|
[0 0 1] | 1|

= |ax^2 + ay^2 = 1| |1|

|-ay*ax+ax*ay = 0| = |0|
|1 | |1|

- So, we get a unit vector pointing along the x-axis - which is what we wanted!
- But here's a question: if we multiply B by this matrix, what would we get?
- Well, it turns out that we'd rotate B onto the Y axis!
- So, that's all we had to do - it's almost magically simple!

- "That's all well and good, Professor Turk, but how does it help us with rotations?" Well, let's try looking at this in 3D!
- Let's say that A is now a 3D vector |ax, ay, az|, pointing off in space somewhere, and we want to rotate a point around A - how can we do it?
- Well, we're not going to - INSTEAD, we're going to break this down into 3 simpler problems:
  1) We're going to rotate A onto the X-axis
  2) We're going to rotate about the X-axis (which we know how to do)
  3) We're going to rotate A back to its original positions
- So, how do we do step 1?
  - Well, very similarly to what we did in the 2D case, we need to find 2 perpendicular vectors to A; to do this, we'll need:
    - A itself
    - N (some other vector that's not parallel to A - pretend we have an easy way to find this)
    - B: the CROSS PRODUCT of AxN that's been normalized, and therefore perpendicular to both of them!
    - and C, the cross-product of AxB (which should be unit-length if we've normalized B)
  - So, we now have 3 orthogonal vectors A, B, and C - which we can now use to create the following rotation matrix:

R = [ax ay az 0]
[bx by bz 0]
[cx cy cz 0]
[0 0 0 1]

- Will this really rotate A onto the X axis? Well, yes!
  - The x-coordinate of R*A will be one (dot product with itself)
  - The z/y coordinates of R*A will be 0 (dot product of A with

perpendicular vectors)

- For step 2, then, we'll just rotate about the X axis, which we already know how to do!
- ...but then, for step 3, we need to undo our first rotation. How do we do that?
  - Well, we need the inverse of our first matrix! "But it's a 4x4 matrix
  - wouldn't that be annoying to solve?"
    - Normally, yes, it would be - but in the very special case of an ORTHONORMAL matrix like we've got here, it turns out that the matrix's transpose IS its inverse!

    R^-1 = Rt = [ax bx cx 0]
    [ay by cy 0]
    [az bz cz 0]
    [0 0 0 1]

- SO, our final rotation matrix around some arbitrary vector A looks just like this:

    Ra = Rt * Rx * R

- A quick side-note: an easy way to find N, given our starting vector A, is the following:

```
getN(vector A):
    if (ax == 0):
        N = (1, 0, 0)
    else
        N = (0, 1, 0)
```

- "Every method I've seen for finding this vector N has some sort of if-statement, implicit or otherwise. This confused me for awhile, but then it dawned on me: it's just a special case of the hairy ball theorem from topology (which basically says a vector field on a ball has to have a pole somewhere)...which I don't expect ANY of you to know for tests or anything, but that was the epiphany I had."

- Here's a problem that seems unrelated, but is actually very tied to this rotation stuff: how do we put a camera in our 3D scene at any position, pointed

in any direction?
    - We've learned how to deal with projections and camera stuff when we're at the origin, pointed down a single axis - but sometimes life gets boring, and we want to see the world from other angles
    - As it turns out, specifying the camera's position really does just take 3 numbers: x, y, and z. It's just a point in space
    - ...but how do we specify which way we're pointing? Well, the most obvious thing would be to have another 3D vector for which way we're facing...
        - ...but that's actually more than we need! Remember a globe? We can specify ANY direction on that globe using just latitude and longitude, since we don't need to care about the vector's length!
            - For the same reason, a unit vector technically carries more information than we need - if we wanted to, we could figure out where it was with just 2 things
    - For a camera, we'll call these values the AZIMUTH and ELEVATION
- We also need a 6th number: the "orientation" number, that basically just tells us which direction/axis is "up" in our world
    - But many graphics libraries require us to give up to NINE numbers! What gives? Well, skipping some of the details, it's for convenience sake so that we don't have to calculate the azimuth/elevation when passing them to the functions

- On Friday, we'll dig a bit deeper into HOW we're going to implement this - in the meantime, read the textbook for the camera sections, and I'll see you then!

//**********/
/
//********** Viewing Transformation - January 25th, 2019 **********//
//**********//

- Okay, people have some questions about project 1A, which is due tonight:
    - "After I do a rotation, one of my coordinates is 8.97*10^-9?" - that's close enough to 0 for me!
    - (...everything else is pretty straightforward)
        - "Most common matrix multiplication error I've seen is accidentally altering one of the matrices as you multiply - you want to COPY the values, otherwise you'll end up affecting your results"

- Alright, remember how we did some arbitrary rotation stuff on Wednesday?

Where we rotated the coordinates, rotated the stuff, then rotated the camera back? That'll be important for doing our viewing transformations

- Now, Processing (for the viewing transformation function, called "camera()") will ask us for 9 numbers:
  - 3 specifying the camera position
  - 3 specifying the point we're looking at (?)
  - 3 specifying the "up" vector
- OpenGL does basically the same thing, but with a different function (called "glnLookUp")

- Most of what we'll talk about today will assume the perspective projection, since we're specifying the EXACT location of our camera
  - "Parallel projection is a bit more loose, since we have a viewing box instead of any specific point"

- In Processing, if we were to invoke the command:

```
camera(0,0,8, 0,0,0, 0,1,0)
```

  - This means the camera will be positioned at (0,0,8), should look at the point (0,0,0), all with the up direction being straight-up in the Y direction
    - In this case, all it does is move the camera to the origin for us (?)
  - This command does the EXACT SAME THING as:

```
translate(0,0,-8)
```

  - However, if we had a camera initially at the origin and pointed down the -Z direction, then

```
camera(0,0,0, 1,0,0, 0,1,0)
```

  - Is equivalent to just saying:

```
rotate(90, 0,1,0)   // rotate on the y-axis
```

- Mathematically, how does this work, though?
  - Well, let's say we know 3 vectors:
    - The position of our eye/camera "e"

- The "gaze direction" of our camera "g"
- The up vector for our camera, "t"

- We want to pick up our camera, set it down at the origin, and then point it so it's facing the -Z direction (but everything else is shifted, so that's okay)
- To do this, we need to do a translation, and THEN a rotation:

Mview = R*T

- The translation to the origin is pretty straightforward:

T = Translate(-ex, -ey, -ez)

- ...but the rotation is a little more complicated:

w = -g/|g| - this is the opposite of our view direction
u = (t x w)/|t x w| - this is pointing to the LEFT of our eye
v = w x u - this is pointing to the right

- These vertices leave us with 3 perpendicular UNIT vectors, so we now have a basis we can use to rotate, using our arbitrary rotation trick from last time!

- This leads to a final rotation matrix of:

R = [ux uy uz 0]
[vx vy vz 0]
[wx wy wz 0]
[0 0 0 1]

R*g = [0, 0, -1, 1] - yup, it's now pointing to -Z!

- "...okay, that might not be the clearest explanation I've given, but here're the important things this camera transformation is actually doing that you should remember:"
1) It's moving a virtual camera position from SOMEWHERE in 3D space to the origin
2) It's rotating the camera's gaze direction to be parallel to the -Z axis

- ...Okay, so now, somewhere in the middle of his many buried notes, Professor

Turk is going to tell us about output devices
    - Specifically, we're going to talk about LCD screens and E-Ink (at a VERY high level)
- First up, though, how does the CPU deal with output devices?
    - Well, if you stretch your minds back to CS 2200, the CPU is putting stuff onto the output bus, which then stores the image we're putting on the screen into the FRAME BUFFER
        - This buffer usually has several megabytes of memory, which used to be absolutely GINORMOUS by 1970s standards but is considered pretty trivial today
    - This buffer then sends the stored image to the "video controller," which then sends it out to whatever monitor is connected
        - "In graphics, we do something known as DOUBLE-BUFFERING: one buffer holds the image currently being displayed, while the other holds the next image that's ready-to-go"
            - Why can't we get away with just using 1 piece of memory? Well, imagine the screen is currently drawing your beautiful image. Suddenly, while it's 1/3 of the way through, you start overwriting that with a new image, and the screen starts drawing your NEW image the rest of the way! You've torn the image in two!
                - For that reason, this is called IMAGE TEARING
                - Occasionally, it can also result in a partially blank screen, if the buffer is flushed before you write to it
        - So, what actually happens in double buffering is that the frame buffer holds whatever's the latest and greatest image from the CPU, and the video controller will load that image AFTER it's done drawing the current one

- "I'm going to put on my special glasses now, because we're talking about Liquid Crystal Displays" *Professor Turk puts on cardboard glasses, looks only slightly insane*
    - What're liquid crystals? They're these wonderful molecules that chemists came up with some decades ago that change their conformation in response to pressure, temperature, electric fields, etc.
    - Liquid crystal DISPLAYS use a specific type of liquid crystal that twists and untwists in response to voltage being put through them
        - In the monitor, a polarizing filter twists all the light so it's pointed in the exact same direction
            - If no voltage is being put through them, the crystals don't turn, and allow light to go through them

        - If there IS voltage applied, the crystals turn, blocking the polarized light for that pixel!
- "...okay, let me explain the glasses now"
    - The glasses have 2 polarized lenses in them: 1 that's turned 90 degrees, and another turned the opposite way
    - Up front, if he holds them up to an LCD screen, 1 of the lenses is black (no light gets through), and the other is clear! If he rotates them, that reverses!
        - "These glasses were the old way of doing 3D films, where one eye would see the left image and the other would see the right. Nowadays, they use something called circular polarization, which is much weirder"

- And with that, the day is done! You are free!...until Monday!

//**********/
/
//********** Displays and Lines - January 28th, 2019 **********//
//**********//

- There are no pre-class announcements today. There is nothing to write down.
    - You can stop reading this now. There is nothing to read.
        - Stop it.
            - Stop it now.
- There is homework due next Monday. This is less than noteworthy.

- On Friday, we began to learn about LCD displays - specifically, the kind that use "twisted nematic liquid crystals" (the most common type in LCD monitors)
    - So, light passes through when there's no voltage, and is blocked when there is voltage
    - Intermediate amounts, of course, can just partially twist the crystals, letting us control the brightness
        - To turn a pixel on or off, a full LCD screen uses "thin film transistors" - transistors so thin, they're actually translucent enough to put on the glass itself, letting us activate a given column/row to activate the pixel we want
- But how do we get colors with our LCD screen, though? Well, in each pixel there're 3 sub-pixels arranged next to each other corresponding to red, green, and blue - each one is its own light, and by mixing them we can control the color of the pixel itself

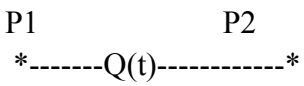
- So, there're actually 3 separate lights we have to control per pixel - "I'm sorry, I was fibbing a little last time!"

- Now, let's consider a very different kind of display - E-INK (or, sometimes, "e-paper")
  - This was a kind of display technology invented wayyyyyyyy back in the day by Xerox, and it's a display that ONLY reflects light, rather than producing it - just like traditional paper and canvas!
    - To do this, the display contains pigment particles - some of them light, some of them dark - in micro-capsules right beneath the screen (a single pixel might contain dozens of microcapsules).
      - The black particles are positively charged, while the white ones are negatively charged
      - This means when the electric field is negatively charged on the bottom of the display, the black particles move towards the bottom and only the white particles become visible, and vice-versa
        - Zooming in on e-ink displays looks kind of weird - you can see the microcapsules making up each pixel, like pebbles or bubbles
  - What pros and cons do we get from this sort of display? Why not just use an LCD for everything?
    - Well, on the pro side:
      - Being a reflective display means we can read this in bright sunlight, no issues at all
      - It takes very little energy to run, compared to the LCD needing to power a strong light!
    - Unfortunately, though, the refresh rate is considerably slower than other display techniques. It's getting better, but these sorts of displays are still impractical for video
      - Color e-ink is still in its infancy, too - they've started to appear in watches, but for awhile they've stayed in the lab, and color e-ink displays tend to still be much less vibrant than traditional displays

- Computer display technologies have come and gone through the years, of course - cathode ray tube monitors were the hot stuff for years and have since gone the way of the dodo, and E-Ink and LCDs are hardly the only kinds of technologies out these days. So what're some candidate display technologies of the future on our horizon?
  - TMI (?) has a patent on "mirror display technology," where each pixel is

a mirror that reflects light - this is common in projectors and virtual reality devices
    - Currently, most VR devices and head-mounted displays are just LCDs that use special lenses
- Various labs have been looking into holography and digital fluorescent crystal techniques to try and get true 3D holographic displays
    - This is one of the holy grails of display technology, but we haven't cracked the code for it yet. You won't find this stuff outside of semi-functional prototypes in the lab
        - Professor Turk saw a demonstration that tried to heat up air molecules with a laser, creating sparks in midair, and it was interesting - and apparently very loud
    - Slightly more pragmatic approaches are partially-transparent LCD layers that try to have different layers of displays, letting you display "slices" of a 3D scene, but these haven't really caught on
    - You can also do the classic two-image 3D fakery, but this is "weak 3D" - only one person can see the effect
- Researchers have looked into putting images into people's eyes directly, drawing the image on the back of your retina with a laser beam ("which is kind of a scary way to put it")
- A common, end-of-the-road sci-fi possibility is hooking up some electronic gizmo directly to your visual cortex to skip your eyes entirely
- but this is wayyyyyy out there stuff (and Professor Turk has declared he probably won't be volunteering as a test subject)

- So, those are some of the display technologies we may have to worry about one day - but back in the present, how do we draw simple things? How do we draw the MOST simple thing: a line?
    - There are two common equations for defining a line: the PARAMETRIC and IMPLICIT ways
    - For parametric:



    - This is where we have some parameter "t" that defines how far we've "slid" along a given line between points p1 and p2
        - To do this, we'll have an equation for the X-coordinate on the line, and another for the Y-coordinate:

x(t) = x1 + t*(x2-x1) = x1 + t*dx
y(t) = y1 + t*(y2-y1) = y1 + t*dy

- We'll then say the point on the line "Q(t)" is:

Q(t) = (x(t), y(t)) = P1 + t*(P2-P1)

- Here, we're multiplying the vector "P2-P1" by a scalar
  - "There's another way of writing this that's useful for curves, but that's for another day"
- What happens if t is greater than 1 or less than 0, though?
  - Well, if it's greater than one, we'll start extrapolating past the end of P2 - and if it's less than 0, it will extrapolate backwards behind P1!
    - "So, we haven't just defined a line segment, but a whole line! For convenience, the line segment is just the part between t=0 and t=1!"
- For implicit lines:
  - "At first, this seems a bit more unusual than the parametric way, but hopefully I can give a little sense to it"
  - Implicit lines are kind of like a guessing game: we're given a point (x,y), and then we say that point is "on the line" if f(x,y) = 0, and off the line for any other value
    - So, we need a function that equals 0 for all the points on the line, and ONLY for those points!
  - How can we come up with such a function? Well, let's think of the format this equation would have to take:

f(x,y) = a*x + b*y + c

- We just need to figure out the coefficients a, b, and c that'll give us the line we want!
  - Notice that if we multiply the equation by a scalar constant, all the values would be scaled by the same amount, so the points on the line wouldn't change
- As an example, if (A=1, B=-1, C=0), we'd have the line:

f(x,y) = x - y

- This is the equation for a 45-degree angle line going through

the origin!
- Sure enough, f(0,0) = 0, f(1,1) = 0, f(-2,-2)=0, but f(-2,1) = -3; NOT 0!
- Anything below the line will be positive, anything above it will be negative, but ONLY the stuff right on the line will exactly be 0!
- Furthermore, any points on a line parallel to the one defined by our equation will have the same value for f(x,y) - "just to give you some intuition for this"

- As it turns out, you've seen this equation before in a different costume. Remember the line equation from grade school?

y = m*x + b

- This was an implicit line equation all along!

- We'll finish going over these line equations on Wednesday, and then we'll see how we can use them to start drawing the lines themselves on our screens - tune in then!

//***/
/
//************* Intro to Rasterization - January 30th, 2019 *****************//
//***//

- So, waaaaaaaaaaaaay back before the non-snowday (the *shudder* before times), we were talking about how to write out equations of line, implicitly and parametrically

- Now, let's see some code for actually drawing a line:

```
void line(x0, y0, x1, y1):
    dx = x1 - x0
    dy = y1 - y0
    // this is the maximum number of pixel "steps" we have to take to get to
    // the end, not the actual line's length
    length = max(float_abs(dx), float_abs(dy))
    xinc = (float)dx / length
    yinc = (float)dy / length
```

```
x = x0
y = y0
for (i = 0; i < length; i++):
    //since we can't put a pixel in a fractional location, just round
    // to the nearest pixel
    gtWritePixel(round(x), round(y), someColor)
    x += xinc
    y += yinc
```

- So, if we run this code on (say) a 6x6 grid (the smallest monitor in the world), and we try to draw a line from (1,3) to (5,5):
  - We end up with the following initial values:
    - dx = 5 - 1 = 4
    - dy = 5 - 3 = 2
    - length = 4
    - xinc = 4/4 = 1
    - yinc = 2/4 = 0.5
  - This leads to a sort of stair-stepping line
    - "If we only have black and white pixels to deal with, this line will look pretty jaggedy, or ALIASED - fortunately, almost all displays these days let us have a variety of shades, which means we can soften this line to look smoother using ANTI-ALIASING techniques"
      - We might not get to this in class, but it's in the textbook if you're interested
- So, what kind of line equation is this function using? It's parametric!
  - ...but where's the "t" value here? Well, it's "i"! It's a bit hidden, but we're basically using our incremented i value in place of t
  - There are more complicated line drawing algorithms if you want to use them, but we won't really bother with them in this class

- Now, here's some context for where we're going in this class:
  - On the screen right now is a "shutterbug" image of 3 different orthographic views of a wireframe scene (lines only) - it looks pretty primitive, and is similar to what we're trying to do in class right now
  - If we jump to perspective projection, it'll look a bit more jumbled (since it's only lines, and we can see through them!), but the perspective is clearly more lifelike now
    - If we jump to the next image, we see that there are now "hidden surfaces" in the rendering - objects that are behind another object are

no longer drawn!
- Then, the next image has surfaces on the objects - there's COLOR on the objects, but there's no shading, so the image looks pretty flat
- In the next image, there's some basic shading, and now it looks like the objects have depths!
    - Still further on, there are multiple light sources in the scene, and then objects casting shadows and having reflections, and then objects having textures and images!

- "So, that's where we're headed in the not-too-distant future for this class - but for now, let's get back to the present and keep working towards this stuff!"

- So, let's talk about our next step: POLYGON RASTERIZATION!
    - Let's start off with the humble rectangle: how do we fill in a rectangle with a solid color on our screen?
        - Let's assume we already know where the corners of the rectangle are going to be drawn on our screen; (xmin, ymin) is the lower-left corner, (xmax, ymax) is 1 pixel BEFORE the top-right row/column
            - Why stop 1 pixel early? Because if 2 objects were right next to each other, they would overlap and "fight" over who fills in the column/row of pixels where they're touching; if we do it this way, then who "owns" which row is now well-defined (it's just taken by the left/bottom row of the object to the right/top)
                - This isn't a big problem now, but it becomes a much bigger issue when dealing with transparency, or techniques like "XOR drawing"
        - For the time being, let's also assume our rectangle isn't rotated, and is lying flat against the screen
    - So, with that knowledge, filling in the rectangle on our grid would look something like this:

```
for (y = ymin; y < ymax; y++):
    for (x = xmin; x < xmax; x++):
        gtWritePixel(x, y, color)
```

- So, doing this with rectangles isn't too difficult - but what about for general polygons?
    - As I'm sure you remember, a POLYGON is just some region of space that's enclosed by a set of straight lines
        - These can be CONVEX (i.e. any infinitely-long line can intersect the shape at at most 2 points) or CONCAVE (there's a "cave"/inset in the

object where they can be hit multiple times)
    - "Concave polygons tend to be the more ugl-...well, I shouldn't say ugly. They're, um, different."
  - Some graphics libraries will let you draw polygons with "features," i.e. holes (like a triangle with a star-shaped hole in the middle) - these are tricky to deal with if you have to write the drawing functions from scratch
    - This isn't really a focus of this course, but it's something to be aware of out there in the wild
- To rasterize ANY polygon, we're going to fill in one "scanline" (i.e. row of pixels) of the polygon at a time, from bottom to top, maxing sure to fill between INTERSECTIONS
  - What intersections? We'll get there in a second, but it's basically where the lines of the polygon intersect the edges of our pixels in the grid
  - As for bottom-to-top, it's just an arbitrary convention that's become pretty standard
- So, if we had some rotated triangle overlaid on our tiny monitor grid again:
  - We find the points in the row where the lines are intersecting with the pixel grid
  - Fill in all the pixels in the row that're in-between those points
  - Move up to the next row
- In pseudocode:

```
for (y = ymin; y < ymax; y++):
    find x intersections with polygon edges
    sort intersections on x-values
    fill between pairs of intersections (from left-to-right)
```

- ...of course, there're some unanswered questions here, including a pretty big one: how do we find these intersection points quickly?
  - Fortunately, it's pretty easy once we have the first intersection point
    - For now, suppose we know where the leftmost intersection point in the row is
    - We know where the closest point of the POLYGON (not the intersection) to our left is, and the next point of the polygon to the right; knowing this, we can make a right triangle (3rd point is just (x1, y0)

- If we then scale that triangle so its height is 1 pixel,
we'll have a similar triangle whose base's length is how much
we need to step to the right for every scanrow (i.e. 1 pixel)
we go up to stay on the polygon's edge!
- So, if we know where 1 point is, we can figure out where the
edges of the polygon are really easily/quickly! Go efficiency!

- We'll keep talking about rasterization on Friday - 'til then, keep working on your projects, etc.

//**********/
/
//****************** Hidden Surfaces - February 1st, 2019 ******************//
//**********//

- We continue on our epic journey of drawing triangles. Repeatedly.

- Alright, the hope for today is to wrap up polygon rasterization - and to capture your feeble attentions, here's a possible test question Professor Turk likes:
  - "What algorithm is used to draw in your graphics card?"
    - Rasterization by z-buffer
    - Ray tracing
      - "Ray tracing is a beautiful, wonderful way of doing things, but nope, it's barely used for real-time stuff right now - polygon rasterization is more efficient, so it's still the way of the world!"

- So, let's keep going on our *sketchy* (HAHAHAHA -_-) journey of drawing triangles to the screen
  - What information do we want to know for this, to draw the edges? Well...
    - The minimum/maximum y-value (i.e. lowest point)
    - xleft/xright (the left/right intercepts for the two edges above the lowest point)
    - dxleft/dxright (how much are we "jumping" to the left/right for a given edge for every pixel we go up?)
  - With that, the pseudocode for drawing our triangle looks something like this:

```
find ymin, ymax
find xleft, xright, dxleft, dxright //this sets up the two non-top edges
for (y = ceil(ymin); y < ymax; y++) {
   for (x = ceil(xleft); x < xright; x++) {
      writePixel(x, y, color)
   }
   // possibly swap edges if needed
   xleft += dxleft
   xright += dxRight
}
```

- So, a few unfilled details aside, that sounds pretty good - but we have a problem. Right now, ANY polygon in our camera will be drawn, even if it's behind another object! How do we deal with this? How do we determine which surfaces are hidden and which are visible?
  - This is known as the HIDDEN SURFACES problem, and it was a big deal back when computer graphics was getting started in the 1970s
    - Nowadays, though, it's more-or-less a solved problem
  - How do we begin approaching this? There're a few well-known techniques we'll talk about:
    - Painter's algorithm
    - Z-buffering
    - (MUCH later) Raytracing
      - "We used to also talk about something called BSP trees, but really, it's just a more complicated variant of the Painter's algorithm, so we'll ignore it"

- First up: the Painter's algorithm!
  - "A quick note: very few people use this algorithm anymore, since there're more effective alternatives we've figured out since"
  - Let's say we have a cube we want to draw, with both front AND back surfaces, and we only want to draw the ones that should be visible - what'll we do?
    - According to the painter's algorithm, we do the following:

      1) Sort the polygons in order of increasing Z/depth
         - We'll do this by just taking the centroid (i.e. average of points) of each polygon, and ordering based on the centroid's Z value
      2) Draw polygons in back-to-front order

    - This'll mean that we draw the closest polygons overtop the ones in the back, so they won't be visible!
  - "But Greg, if the back surfaces weren't going to be drawn, isn't that wasted computation? Why didn't we just skip to drawing the visible surface?" And you're right!
    - In SOME cases, we can figure out which surfaces aren't visible, and skip over them
- HOWEVER, there's a serious flaw with this algorithm: it doesn't work for every case
  - For instance, what if we have a BIG polygon whose center is farther forward than a smaller polygon in front of it? Then that small polygon will be hidden incorrectly!
    - The BSP tree algorithm addresses this flaw
  - What if two polygons are intersecting, so that they're both behind AND in front of each other? Then only one polygon will be drawn; that's not what we want!
- Because this algorithm is so simple, it was used pretty frequently in the old days - but in the end, the incorrect results it gave led most people to abandon it

- So, what's the new hotness that most people use today? It's a now-common technique called Z-BUFFERING
  - Back when this was first proposed, people though that it used far too much memory to ever be practical - but of course, RAM is cheap as chips nowadays, and it's become the de-facto hidden surface algorithm for 99% of the globe
    - The guy who originally wrote the paper for this is now one of the higher-ups at Pixar Animation
  - How does it actually work, though? Well, here's some pseudocode:

```
//Setup, just drawing the background color and pretending each pixel is
//an infinite distance away from us
for every pixel (x,y):
    writePixel(x, y, background_color)
    writeZ(x, y, very_far_away_large_constant_value)

// Now, the real workhorse loop
for every polygon:
    Figure out the pixels polygon covers using our rasterization
```

```
techniques
for every pixel(x,y) in polygon:
   pz = Z-value of polygon at pixel (x,y)
   if pz >= ReadZ(x,y):
      // pixel is the new closest pixel!
      writeZ(x,y, pz)
      writePixel(x, y, poly_color)
```

- So, we just go through every PIXEL we're going to draw, figure out what the closest object *in that pixel* is, and draw that!
  - Memory-wise, we have to hold an integer value for each pixel in the monitor, which was viewed as insane back-in-the day - but, nowadays, is perfectly reasonable

- So, that's the gist of hidden surfaces, and one more step forward on our thousand-mile journey to Draw All The Best Triangles.

//**********/
/
//********** Shading Basics - February 4th, 2019 **********//
//**********//

- "Reading: Lines/Hidden Surfaces (last week); Shading (today)"
  - This will be useful; my line-reading skills have been regressing lately, and I've always had some issues with seeing hidden surfaces
- "Technically on the schedule, we should be doing radiometry right now - but for concision's sake, we're going to skip it for now and jump straight into shading. We might come back to it later on in the course, but if you're interested, the textbook is always open!"
  - In other news, Professor Turk figured out how to disable automatic white balancing on the camera, so we can now read the paper he's writing on! Woohoo!

- QUICK MATH REVIEW FACT:
  - If we have 2 unit vectors "u" and "v", then the dot product is:

    u*v = v*u = cos(theta)

  - This lets us check what the angle between two vectors is, which is VERY

useful
- We can also calculate the SURFACE NORMAL of a polygon (i.e. a vector pointing perpendicularly "out" from the surface)
  - Say we've got a triangle ABC (where each one is a 3D point/vector) - in that case, we can do THIS:

    - vector1 = B - A
    - vector2 = C - A
    - Normal = crossProduct(vector1, vector2) = v1 X v2

- Now, let's switch gears to the topic of surface shading
  - When we look at a surface, it isn't just a single color; instead, it's darker or lighter based off of how it's interacting with the light
  - In particular, there are two BIG factors that affect how bright a surface is:
    - Where the light sources shining on the object are
    - Properties of the surface itself
      - There're a LOT of these, but for now, we'll only care about two of them: if the surface reflects light (and how much), and if it absorbs light (and how much)
  - If an object is DIFFUSE (or 'matte' or 'Lambertian'), it means that the surface reflects light equally in all directions
    - A good example of this is chalk, or paper (from most angles)
    - (cue several students excitedly looking at pieces of paper from various angles - thank you, Fresnel effect!)
- Now, imagine that there's a light source (like the sun) where all the photons are coming from a single direction
  - If we hold the paper flat up against the light, a set amount of photons are hitting it in a 2D-area 'a'
  - If we tilt that paper, then the effective surface area where the same amount of photons hits has increased to 'a/cos(theta)' - i.e., the same amount of light has to cover a larger area, so the light will seem dimmer!

- So, to figure out how much light we should draw on the surface, we need to know how much light is hitting the surface, and how much of that light is reaching our eyes
  - Since a diffuse surface reflects light equally in all directions, the position of our eye won't matter when drawing it - but it WILL matter for reflective surfaces
  - So, for a given point on our surface, let's say we know a) the

unit-length normal vector N at that point, and b) the unit vector L pointing TOWARDS the light source we want to worry about

- We can get a vector pointing towards the light pretty easily, by saying "L = lightPos - surfacePos"
- Using our dot product, we can calculate the angle between the two vectors as:

N*L = cos(theta)

- If theta is small, it means the light is striking the surface almost head-on, so the surface will be bright
- If theta is large, it means the light is just glancing the surface, so the surface will be dim

- So, this quantity represents approximately how much light is hitting our surface!

- If we then also know the color "Cs" of our surface, and the color "Cl" of our light, we can multiply it by our brightness factor to get what the color the surface should be! Putting this together, we get an equation:

final surface color = Cs * Cl * (N*L)

- ...where Cs is the surface color, Cl is the light color, N*L is how much light is hitting the surface, and 'b' is a scaling factor
- Since we have to deal with RGB color, we'll do this equation separately for the red, blue, and green channels; so REALLY, we have a matrix that does this 3 separate times:

C = (Cr, Cg, Cb) = [Csr * Clr * (N*L)]
[Csg * Clg * (N*L)]
[Csb * Clb * (N*L)]

- However, what if the angle between the vectors (i.e. the dot product) is negative? Well, it means that the light is BEHIND the object, so no light should be hitting the surface! Therefore, our equation should actually be:

C = Cs * Cl * max(0, N*L)

- However, is it realistic for objects in shadow to be COMPLETELY pitch black? No! In the real world, light bounces around and can provide "indirect illumination," even to objects the light source can't directly "see"

- Calculating all those bounces, however, is pretty expensive - so instead, we'll often just fake this indirect light by adding some constant ambient light color 'Ca' to our objects so shadows aren't just pure black:

C = Cs * (Ca + Cl*max(0, N*L))

- For a number of years, calculating indirect illumination at all, in fact, was pretty difficult...but today, there are several (computation intensive) well-understood methods for doing it correctly, including:
  - Photon mapping
  - Metropolis light transport
  - Radiosity

- So, those're the basics of Lambertian diffuse shading - but what should we do for SPECULAR (i.e. glossy) surfaces, like plastics and metals?
  - Well, there are a number of different ways of doing this, but one of the simplest and most common is the PHONG-BLINN ILLUMINATION model
    - Again, let's say we've got a surface normal N and our light-pointing vector L again
    - But now, let's ALSO suppose we have a vector "R" that's the mirrored version of L (flipped about the normal vector), and a vector "E" for which way our eye/camera is pointed
      - If the eye is pointed towards R (i.e. angle between E and R is small), then it should see the full glare of the reflection
      - If the eye is pointed perpendicularly to R, it shouldn't see any glare at all
      - In-between, it'll drop off by some pretty high amount
    - So, one way of representing the reflected light is like this:

cos(angle between E and R)^(specular power)
= (E*R)^P

    - ...which'd give us a final PHONG ILLUMINATION color of:

C = Cl * max(0, (E*R)^P)

  - This should give us the SPECULAR HIGHLIGHTS (i.e. bright spots) we see on shiny objects like glossy plastics

- So, using these two techniques (and some trial and error), we can simulate a

wide variety of materials - cool! However, there's still plenty we can do to make this look better (early 3D animations had a reputation for everything looking plastic, for instance), so we'll keep pushing forward in the next lecture.

//**/
/
//********** Basic Shading (cont.) - February 6th, 2019 **********//
//**//

- Lastly, we learned light
  - Thusly, we'll learn...well...um...
- Anyway, we're doing stuff. Probably.
- Also, project 2A is up on canvas, where we're making some object/character that we're planning on animating in part 2B
  - You are NOT allowed to make a snowman or the Android logo, because SO many people made them last year - be creative!

- Yup. Last time we talked about diffuse surfaces, where if the surface is "looking" at the reflected ray: brightness! Otherwise: darkness!
- Specular surfaces (read: shiny) look shiny because they have these bright spots we see (known as HIGHLIGHTS), which occur when we're looking at the reflected light head-on, per the Phong illumination equation:

$$C = Cl * \max(0, E*R)^p$$

- How do we calculate this R vector, the reflected light ray?
  - Well, one of the simplest ways is to project the L vector onto the normal vector, doubling its scale like so:

$$x = 2 * N(N*L)$$

  - ...and THEN subtracting from the L vector to reflect it (that's why we doubled its length above)

$$R = L - x$$

  - All together, then:

R = L - 2N(N*L)

- Now, that's the Phong model, and it's all well and good, but it's not the most physically accurate model in the world. ANOTHER model for specular surfaces is the BLINN HALF-ANGLE model, and it goes something like this:
  - The "halfway vector" is some unit vector that points halfway between the eye and the light, like so:

    H = L+E / |L+E|

  - A researcher named Blinn noticed that if the angle "b" between H and the normal vector is small, then the eye is exactly mirroring the light's position - otherwise, the eye isn't at the right location to see a glint!
  - So, we can model this by saying the strength of the light is:

    (H*N)^p = (cos(b))^p

    - Which gets us a final color of:

      C = Cl * (H*N)^p

- "This is a little more realistic than the Phong model, but really, most people don't care all that much"

- So, we've got an equation for diffuse surfaces and some equations for specular surfaces - let's start putting them together!
  - Here's our Big Scary Hairy Final Color Equation:

    C = Cr*(Ca + Cl * max(0, N*L)) + Cl*Cp(H*N)^p

    - What does all this mean? Well,
      - Cr is the diffuse color of the surface
      - Ca is the ambient light color
      - Cl is the color of the light
      - Cp is the color of the specular highlight
        - "Plastics tend to have sharp white highlights, regardless of the diffuse color, while metals tend to have somewhat fuzzy highlights that are closer to the color of the metal"
          - Metals tend to have somewhat-rough surfaces that reflect light, but still have a diffuse component that absorbs some

colors, giving them that colored highlight
- Plastics, on the other hand, are actually transparent materials with pigments inside of them - the transparent outer part reflects the photons with little change, while the ones that make it through to the pigment are colored instead
- p is the specular exponent, controlling how shiny the surface is (larger values are more shiny, smaller values are rougher/less shiny)
- All of these colors will be scalars (for a given RGB color channel), while N,L, and H will be vectors whose dot products are used

- Now, where do we actually apply this shading equation? In the middle of each polygon? Per-pixel? Well, there're 3 common options:
  - Per-polygon basis
    - This is known as "flat shading"
  - Per-vertex (aka Goraud interpolation)
  - Per-pixel (aka Phong interpolation)
- To go over each one in more detail:
  - Per-polygon shading is where we use just 1 normal for the whole polygon (usually a triangle), and apply the same color to the whole polygon
    - This is fine for flat, boxy surfaces, but when we use it for curves surfaces, it doesn't really work - you can see abrupt shading changes as the curve goes around
  - Per vertex shading, then, is what we start looking at when we need to render these more curved surfaces, by doing the following:
    - We calculate the surface normal at each vertex
    - We apply the shading equation for each vertex to get its color
    - For each pixel, we then interpolate the color at each point on the polygon based on the color of its 3 vertices
      - So, the key is that we do color on a PER-VERTEX basis, and then just interpolate between them, weighting the colors by the point's distance from each vertex
        - e.g. On the vertical edge of a triangle with color C1 at the bottom and C2 at the top, we'd use:

          Color = C1 + (y-y1)/(y2-y1) * (C2 - C1)

        - This is just linear interpolation, weighted by distance - nothing too fancy

    - This is still pretty fast to calculate, but it gives us a much smoother look - neat!

- Alright, we'll finish going over this on Friday. That's all.

//***/
/
//****************** Color Perception - February 8th, 2019 ******************//
//***//

- Two hours of trying to read up on seismic displacement theory later, I can officially confirm I do not like reading math from classes I haven't taken yet (PDEs literally look like actual Greek to me)

- Alright, we didn't *quite* wrap up Gouraud interpolation on Wednesday, so let's do that now!
  - We calculate the normal vector for each vertex, then apply our color equation to figure out the color at that vertex
    - The normal is sometimes calculated by averaging the normal vectors of the polygons touching the vertex - which technically isn't correct, but hey, the mathematicians aren't looking
  - Then, for a given point on the polygon we're drawing, we decide its color by interpolating between all the points, like this:
    - If the point is on the polygon's edge between vertices 1 and 2, the color will just be linearly interpolated:

        C = C1 + ((y - y1)/(y2 - y1)) * (C2 - C1)

      - "Y", here, is the distance from vertex 1
    - For a point somewhere in the middle of the polygon, then, we'll first draw a horizontal line through the point (i.e. either side of the scanline), then take the colors at the polygon edges Ca and Cb; we'll then linearly interpolate BETWEEN those two points:

        C = Ca + (y - ya)/(yb - ya) * (Cb - Ca)

  - All this interpolation tends to soften the edges and make them look pretty blurry, which is often what we want, but not always

- Finally, for PHONG INTERPOLATION, we're going to do per-pixel shading
    - Here, instead of interpolating the colors at each pixel, we're going to interpolate the SURFACE NORMALS across the polygon for each pixel and then calculate the color separately for each pixel using that interpolated normal vector!
        - Here, we calculate the normal vector for each vertex again (doing what we did for Gouraud shading), and then linearly interpolate those normals on the edges (just like we did for Gouraud shading), and THEN use the edge normals and our position in the scanline to calculate the normal for that pixel!
        - Then, at last, we calculate the color for the pixel itself
- Why do all this extra work? Because it leads to smoother shading than Gouraud interpolation, especially for surfaces with not enough polygons and ESPECIALLY for specular highlights
    - If we're supposed to have a highlight just in the middle of a polygon, for instance, and we use Gouraud interpolation, it'll just average the color at the corners - it'll completely miss that highlight, or at least make it look less sharp than it should
    - And it IS extra work - we need to do a shading calculation for each pixel AND take a square root for each pixel
        - Why the square root? Because we need to normalize each normal vector to unit length, which requires calculating the magnitude of each

- Now, us humans can only see "visible light" (i.e. wavelengths between ~380nm-700nm, which is similar to the most abundant radiation released by the sun)
    - Violet is near the 300nm side of the spectrum, red is near the 700nm, and blue-cyan-green-yellow-orange fill in the gaps between
    - With shorter wavelengths, we've got high energy ultraviolet waves, and then even more dangerous x-rays and gamma rays
    - With longer wavelengths, we've got infrared and radio waves, which are a bit tamer

- "Now, this here is a drawing of an eyeball - this is the back, which we never see...because it's embedded in our skull"
    - The primary purpose of the eye, as we all know, is to be squishy and painful when pressed
    - The secondary purpose of it is to be stared into romantically on choice occasions
        - (please don't write the above two on a test unless you're feeling

exceptionally awesome)
- The tertiary purpose is (rather unintuitively) to let us see - and particularly, to see color!
    - In the front of the eye, we've got the LENS (which focuses light into the back our eyes), the CORNEA (the thick covering in the front) and the IRIS (which controls how much light gets in)
    - In the back of the eye, there's the RETINA, which has a bunch of photoreceptors that take this visible radiation and turn it into sweet, delicious electronically encoded images
        - There are two kinds of photoreceptors: RODS (which are only sensitive to brightness, work better in the dark), and CONES (for color vision, of which there are three types: short, medium, and long)
            - "It's tempting to say these are red, green, and blue cones, but I don't think that's quite right"
            - Why are they called rods and cones? Well, because cones actually are a little pointed, and rods aren't!
        - In the center of our retina is the FOVEA, a small area where the density of the photoreceptors is much higher, and where most of our color-sensitive cones live
            - Because our optic nerve needs to go somewhere, there's a little spot in the retina where our nerves come through and there aren't any rods or cones - our "blind spot," where our brain just kind of fills in what's supposed to go there
    - What're the three types of cones? Let's break it down:
        - The SHORT cones are mostly sensitive to light in the ~400nm range (sensitivity drops off like a normal distribution), which corresponds to blue light!
        - The MEDIUM cones are sensitive in the green/~500nm range; these are the most abundant cones in our eye
        - The LONG cones are sensitive in the red/~600nm range
            - There is some overlap between all of these, but the predominant sensitivity range of each is different

- Luckily, our eyes seem to mostly be sensitive to the three colors red, green, and blue (and their combinations). It could've been something else entirely, but luckily, it's not - and that means we can get away with still using 3D matrices for color!
    - Some animals, for instance, can only see two types of colors, while others (like the 11-dimensional color seeing Mantis Shrimp) can see far more

    - Not everyone has the same color vision, even in humans; color blindness is a very real thing (and fairly common in men), and there are also "tetrachromatic" people who can see a slightly wider range of colors than normal
- What colors can we see? That's usually given by the "CIE Chromoticity diagram," which tries to diagram the types of light most people can see
    - Now, there's no such thing as a "magenta photon" or a "magenta cone" - many of the colors we can see are mixes of the three fundamental colors, and don't correspond exactly to a particular wavelength

- As a reward for your patience, here's a picture of a Mantis Shrimp (please provide your own picture of a Mantis Shrimp)

- ...annnnnnnd that's all we wrote!

//**********/
/
//********** Color Spaces - February 11th, 2019 **********//
//**********//

- I might have finished a project...early? What world have I trespassed into?
- Apparently 5 minutes ago, Professor Turk has also decided that project 2A will be due this Thursday (St. Valentine's National Singles Remembrance Day), and part 2B will be due next week on Feb. 23rd

---

- We wrapped up last week by talking about the cones in the human eye, and how we perceive colors. A few important takeaways:
    - There are significantly less blue cone receptors, which means we don't focus on it as well as green/red; this is why many people recommend avoiding blue text
    - There IS some overlap between each of these cones frequency
- We also started talking about the "CIE" chromaticity diagram that shows all the colors visible to the human eye
    - This is actually a 3D diagram, where 2D slices of it are shown at a time; the X-axis is, the Y-axis is , and the Z-axis as brightness
    - Everything on the curved portion of the diagram are "visible chromaticity values," meaning they correspond to visible wavelengths - i.e., we can find a photon for that color!
        - On the flat part, though, are colors we see that don't match up with

an exact wavelengths (mostly shades of magenta, i.e. mixes of purple/red) - we only see them as mixes of different colors of light! There are NO photons with this color, we just perceive mixes of light as such!
- These are known as NON-SPECTRAL colors (because they're SPOOOOOOOOKYYYYYYYY...and, well, not part of the visible spectrum)

- Another thing to note for this diagram is the notion of COMPLEMENTARY COLORS, i.e. colors that are opposite from each other on the diagram (blue-yellow, green-magenta, etc.)

- For a given display device, we say the GAMUT of the device is the range of colors that it can display
- But can't my smartphone display ANY color? NO! Most devices can only display a medium-ish-sized triangle of color inside the CIE diagram - there are vivid colors that your monitor simply can't display
- Why a triangle? Because most displays are just RGB, with only 3 colors - we could add more types of colors to add more corners, but there's still (at least for right now) limitations on how extreme the colors can get
- What about making the corners encompass the whole CIE diagram? Well, to get the triangle's corners that far out, you'd need a monitor that could produce X-rays, which are expensive (and mildly dangerous)

- How do we display colors that are outside our device's range, then? There are 2 main schools of thought:
- PROJECTING the color to the nearest point on the triangle
- CLIPPING any of the color's extreme R/G/B values to the maximum possible with the monitor

- So, most monitors have the RGB color space with ADDITIVE colors, where we produce light of a given wavelength for the color we want
- You can think of the colors we make as a 3-circle Venn diagram, with a circle each for red, green, and blue
- Red + Green = Yellow
- Blue + Green = Cyan
- Red + Blue = Magenta
- Magenta photons do NOT exist; you won't find them on the color spectrum by themselves
- Red + Green + Blue = White light
- Similarly, there is no such thing as a pure-white photon

- You might also think of this scheme as an RGB "color cube" with RGB coordinates from (0,0,0) to (1,1,1), with red, blue, and green all in opposite corners, yellow/cyan/magenta at the in-between corners, black at (0,0,0) and white at (1,1,1), and the mixes of colors all in-between
    - This'd mean that gray is in the very middle of the cube
    - In a computer, of course, we represent these colors as discrete values between 0 and 255, instead of a continuous [0, 1] range, since it's convenient to represent each color channel with a single byte
        - This is starting to change, however - most digital cameras nowadays have more than 8 bits of colors, and high-end monitors have higher color ranges as well
- An alternative to this is SUBTRACTIVE color methods, like with printers and inks
    - Excepting really weird cases, paints and inks don't produce photons - they just absorb particular wavelengths of light, and reflect the remaining ones
    - In this scheme, the primary colors change to CYAN, MAGENTA, and YELLOW
        - "But Greg, what about red, blue, and yellow? What happened to them? Well, while that was still a subtractive color scheme, your teachers didn't want to confuse you by teaching about these weird cyan colors!"
        - Here, in the "CMY" color space, the colors are:
            - Cyan + Magenta = Blue
                - Cyan by itself absorbs red light, and magenta by itself absorbs green - so when we mix them, only blue remains unabsorbed!
            - Magenta + Yellow = Red
            - Cyan + Yellow = Green
            - Cyan + Magenta + Yellow = Black
        - ...and if all colors are left out, you get white, since none of the colors are absorbed!
            - In the "real world," black is often included as its own pigment because it's so common, and because mixing cyan/magenta/yellow perfectly into black is more difficult to get right than just having it by itself. This is known as the "CMYK" color space ("K" instead of "B" so we don't confuse black and blue)
    - We can think of this sort of color scheme as subtracting RGB colors from white light, like so:

        |1| |R|
        |1| - |G| = final color

|1| |B|

- So, we talked about non-spectral colors already, like magenta and white - but for some colors, there are more than one way to get them
  - These colors are known as METAMERS: "two spectral distributions that look the same to humans"
    - Cyan light, for instance, can either be gotten directly as a photon w/ a wavelength between blue and green, OR we can get it by having blue AND green light of the same intensity together!
  - Colors we can get just on the spectrum directly, without having to mix two different colors of light, are known as "spectrally pure"
    - Colors like magenta, on the other hand, can only be gotten by mixing two colors together - the fact we can see them at all is down mostly to human biology interpolating the colors

- On Wednesday, we'll start talking about some other color spaces - not for human perception, or for display devices, but for it to be easier for human designers to get the color mix they want. It's hard to tell what makes off-pink color in RGB, for instance - but a clever color scale can help us out!

- So, that's what's on the menu - see you for dinner (in the most platonic of senses) on Wednesday!

//**/
/
//***** Color Space(cont.) / Intro to Raytracing - February 13th, 2019 ******//
//**//

- Alright, today our goal is to wrap up color and start moving on towards raytracing - let's do it!

- First off, let's talk about two more color spaces that are sometimes used today: HSV, and HLS
  - HSV, or HUE-VALUE-SATURATION, is a color space designed to make it easier to choose the right color you need; it's easier to think about defining colors this way for most people, instead of thinking abstractly in terms of "mixing" red, green, and blue
    - Hue is the actual color (green, magenta, orange, blue, etc.) that you want to have a shade of

- Practically, there are 6 different hues we can pick from: red, yellow, green, cyan, blue, and magenta
- Value is the same thing as brightness, intensity, lightness, etc. - it's how bright/dark your colors are, between 0 (black) and 1 (full brightness)
- Saturation is how vibrant the color itself is vs. the value, between 0 (white/gray) and 1 (the fully vibrant color)
- We can think of this color space as a cone with a hexagonal base, and each of the hues on one of the corners (and white in the middle, and the different shades of color in-between)
- The hue is our rotation around the cone (i.e. which color we're closest to), the value is our color's height within the cone (0 is black near the tip of it, 1 is full brightness near the base), and saturation is how far away we are from the center of the cone (1 is on the edges (i.e. fully vibrant), 0 is right in the center)
- If you think about it, this cone is actually just a distorted version of the RGB cube, with the black corner stretched out and the white point flattened to be in line with the other colors!
- So, this HSV cone is great for picking out specific colors we want: it's easy to think about how make a particular color brighter, or more yellow, etc.
- It's NOT as good, however, for seeing complementary colors, or for working with additive color displays that are based on RGB
- HSL (Hue-Saturation-Lightness) is a variant of this where white, instead of being flat with the other colors, is instead raised up from them to form a double cone; some people thought that white deserved to be given special treatment, and HSL was the result
- ...there are a LOT more special-case color spaces, but really, the ones we talked about'll cover 99.9% of what you need in the real world

- Now, let's change gears here and look at some pictures of spheres and surfaces. These are different from what we've seen so far: the reflections are accurate, the light in glass is distorted, there are soft shadows...it looks pretty realistic! What's going on here? The power of RAYTRACING!
- Roughly speaking, there are two main techniques for drawing 3D graphics on computers these days:
- Rasterization is what we've been doing so far. It's FAST, can be hardware-accelerated on the GPU pretty easily, and is well supported.

But to get it to look nice, we have to use a lot of tricks.
    - The vast majority of video games still use rasterization and Z-buffer
- Ray Tracing is much slower than rasterization, and it's difficult to compute on GPUs. But compared to rasterization, it's a LOT easier to get realistic-looking images, and almost necessary for some effects.
    - Nvidia threw a bit of a wrench in this by claiming their newest GPUs have started to support raytracing, but historically, GPUs haven't been able to support it
    - Because of the extra realism, most movies and special effects use raytracing these days; for these applications, it doesn't matter how long it takes a frame to render!

- "That's great, Professor Turk, but what IS raytracing?"
  - Well, think back to our viewing plane concept, where our camera had a grid of cells that each corresponded to a pixel needing to be rendered
  - Now, in raytracing, we shoot out a line (or RAY) from our camera's eye through each pixel in the grid, and see what object it hits - and the pixel color is determined by that collision!
    - In high-level pseudocode terms:

```
for each pixel(xs, ys):
    create ray R from eye through (xs, ys)
    for each object Oi in scene:
        if R intersects Oi and is closest hit by R so far:
            save the intersection point
    shade pixel based on nearest intersection we found
    # might have to do extra work here for shading/transparency/etc.
```

    - This seems pretty simple, but it does a LOT: the "closest so far" part handles hidden surfaces for us implicitly, for instance, and shooting the ray from the eye gets us perspective projection for free!
        - Isn't this how Z-buffering basically worked, though? Well, Z-buffering only worried about the pixels that we knew had a chance of appearing on screen, so it worked out to being far less computation-intensive on average
  - But why is raytracing slow? Not because we have to go through every pixel (we have to do that no matter what), but because we have to check intersections against EVERY object in the scene. That's fine if we've only got a few cubes, but what about modern movie scenes, which can have tens of

millions of polygons?
- There are some clever techniques for speeding up this loop, of course, but we'll get into those later

- We can describe the rays themselves PARAMETRICALLY, like this:

x(t) = x0 + t*(x1 - x0) = x + t*dx
y(t) = y0 + " = y + t*dy
z(t) = z0 + " = z + t*dz

- We can rewrite this as a 3D vector equation for the position on the line 'r':

r(t) = o + t*d

- Where 'o' is the origin of the ray (x0, y0, z0), and 'd' is the direction of the ray (dx, dy, dz)

- Okay, that's how we define our rays - but what about our object surfaces? How do we define them so we know when our rays have collided with them?
- Well, we usually describe our objects with IMPLICIT equations instead; for instance, a unit sphere centered at the origin would be defined as:

x^2 + y^2 + z^2 = 1

- Or, more generally, the places where the ray intersects this sphere are given by:

(x0 + t*dx)^2 + (y0 + t*dy)^2 + (z0 + t*dz)^2 = 1

- Where "t" is the only unknown, which we can solve to find!

- We'll explain how this works a bit more on Friday - hopefully project 2A won't consume all your free time between now and then.
- Today's farewell: "do widzenia!"

//**********/
/
//********** Raytracing Intersections - February 15th, 2019 **********//
//**********//

- This is the lull before the wave of work...is that a scientifically correct description?
    - *One Google search later* Ah, this is truly the trough before the crest (unless we're talking about my emotional state, in which case reverse the terms, reader)
    - Wherefore bringeth such labor much strain and stress upon us mortals? Soft, what cause moves us to make so much of passing moments, and to heap our souls on idols of the world?
        - (...yeah.)
- THE SCREEN HAS CHANGED. INITIATE PROFESSOR TURK ACTIVATION SEQUENCE.
    - PERFORMING QUESTION ANSWERING SEQUENCE IN 3...2...1...
        - Okay, for homework 2b:
            - Object instancing just means creating multiple copies of an object, using the matrix stack to move/position them in different ways

---

- Alright, yesterday we started talking about ray tracing, and how it's the favored technique for animators - "but, like, who?"
    - Blue Sky Animation, who created the Ice Age movies, started using Ray Tracing allllll the way back to the first Ice Age movie
    - Pixar actually was late to the ray tracing game - they didn't start predominantly using Ray tracing until "Monsters University" in 2013!
        - They did use it for small bits in previous films (reflections in "Cars," etc.), but not predominantly by any means
    - Literally EVERY special effects company since ~2005 uses ray tracing for the vast majority of their effects

- We also started talking about how SURFACES are defined in ray-tracing, and how they're usually defined using implicit equations to help us tell when a ray collides with them
    - For a unit sphere, for instance, we usually define it with the implicit equation:

        $x^2 + y^2 + z^2 = 1$

        - If we want to calculate where the ray pointing in direction (dx, dy, dz) and of length "t" actually intersects the sphere, then we can change this equation to:

(x0 + t*dx)^2 + (y0 + t*dy)^2 + (z0 + t*dz)^2 - 1 = 0

- Simplifying a bit, we can rewrite this equation as:

t^2*(dx^2 + dy^2 + dz^2) + 2t*(x0dx + y0dy + z0dz) + x0^2+y0^2+z0^2 - 1 = 0

- This is a quadratic equation, since x0/y0/z0 are all constants! So, we can solve for t using the quadratic formula!

t = (-b +/- sqrt(b^2 - 4ac)) / 2a

- This +/- means that there are TWO points where the ray intersects the sphere's surface: one through the front of the sphere, and the other out the back!
  - It's like a severe bullet wound - it goes in, but it must also come out!
    - ...there are probably alternative analogies
  - (Professor Turk's microphone died; he promptly began rummaging through his backpack saying "Here battery, battery, battery...")

- ...and finally, for a general sphere of radius "r", centered at (xc, yc, zc), we'll define it as:

(x-xc)^2 + (y-yc)^2 + (z-zc)^2 = r^2

- Now, graphics programmers don't like calling square roots, since they're MUCH more computationally intensive then multiplication! So, we usually just compare the result against r^2 instead of r if we can get away with it

- That's great for a sphere and everything, but how do we figure out where a ray intersects with any polygon in general?
  - Well, a polygon is a 2D shape, right? So maybe we can simplify this problem by projecting our ray onto the 2D plane the polygon lies in, and then figure out the 2D problem of if the point lies within that polygon
  - So, ONE way of figuring this out is to:

1) Calculate the ray's intersection with the infinite 2D plane the polygon lies in
2) Project the triangle/polygon and the point of intersection onto that 2D plane
3) Figure out if, in 2D, the intersection point is inside the polygon (the "point-in-polygon" test)

- So, that sounds like a plan!...but how do we calculate the intersection of a ray with a 3D plane, anyway?
  - Sure enough, we can define a plane implicitly in 3D with the following equation:

    $ax + by + cz + d = 0$

    - Basically, what's going on here is that d is a vector that defines the position/center point of the plane, and the vector (a,b,c) is a vector that's normal to the plane
  - To calculate where the ray INTERSECTS, we'll again factor the ray into this equation:

    $a(x0 + t*dx) + b(y0 + t*dy) + c(z0 + t*dz) + d = 0 \Rightarrow$
    $t*(a*dx + b*dy + c*dz) + ax0 + by0 + cz0 + d = 0 \Rightarrow$
    $t = -(ax0 + by0 + cz0 + d) / (a*dx + b*dy + c*dz)$

    - So, we can solve for t - great! HOWEVER, what if the denominator of this equation ends up being 0? That's not allowed, so how do we prevent it?
      - Well, since (a,b,c) is the normal vector to the plane, and (dx,dy,dz) is the direction of the ray, the denominator is actually the dot product of those two vectors - and if it's zero, that means the the ray is perpendicular to the normal and PARALLEL to the plane, so we can just assume it doesn't hit!

- Alright, that's the intersection figured out, but how do we handle the 2D projection part of this?
  - So, we know where the intersection point is on our 3D plane (missed the rest)

- And then, finally, we have to figure out if our point is inside the triangle
- easy right?

- Well, it's pretty easy for us to just look at a point and say "that's inside something" or "that's outside something" - but when we try to think about HOW we do this, it isn't obvious! So how do we tell our computer to do this?
    - One bad way is the CROSSING test, where you shoot a ray out from the point and count how many times it intersects a line in the polygon
        - The idea is that if the point is inside the polygon, it will only cross the polygon's lines ONCE, but if it's outside it'll either cross twice (entering one side and leaving the other) or not at all
        - But this misses an important edge case: if the point's ray is tangent to the polygon, this method will falsely think it's inside!
    - A slightly better way is the WINDING METHOD: we draw 3 rays out from the corners of the triangle (or 4 for a square, etc.), and count up the angles the rays make with each other while going ONLY IN ONE DIRECTION (clockwise or counter-clockwise)
        - If the point is inside the polygon, we can draw all ray-angle pairs by just moving in a circle, so the angles won't cancel out and'll be some non-zero value (almost always the sum of the polygon's internal angles)
            - HOWEVER, if the point is outside of the polygon, the last angle/ray pair will have to go backwards in the opposite direction, which'll cancel out and result in a total angle sum of 0
        - This method has problems too, though: what if the point is on a line? Is it inside or outside? That now becomes inconsistent! And it's pretty cumbersome, too
    - The method Professor Turk recommends, then, is the HALF-PLANE test
        - Here, we draw a line (our "plane") for EACH side of the triangle, and say that the point is "positive" if it's to the left of the line (i.e. towards the inside of the triangle) and "negative" if it's to the right (i.e. outside the triangle)
            - If the point is negative for ANY of the lines, then it can't be inside the polygon
            - Otherwise, if it's positive for all of them, it must be inside the triangle!

- On Monday, we'll give the actual equation for doing the half-plane test - keep on until then!

//**/
/
//*************** Raytracing (cont.) - February 18th, 2019 ****************//
//**//

- Midterm is NEXT WEEK - details will begin to emerge fairly soon, but keep an eye on it!

- So, last week we ended by talking about the HALF-PLANE TEST for figuring our if a point is inside a triangle, where we make sure the point is on the right side of all the lines making up the triangle's edges - but how do we actually calculate this mathematically? I'm glad you asked!

- First off, we know the equation for a given line is:

$$f(x,y) = ax + by + c = 0$$

- If we have two points p1 and p0, then, how do we calculate a, b, and c?
- Well, we can get the vector pointing along the line pretty easily:

$$V = P1 - P0 / |P1 - P0|$$

- We can also get a vector "W" that's perpendicular to this vector VERY easily:

$$W = (-vy, vx)$$

  - Essentially, this is the same thing as rotating the V vector 90 degrees (try multiplying it by a rotation matrix to make sure)
- SO, as it turns out, this already gets us a and b - they're the components of W!

$$w = (a, b)$$

  - But what's 'c', then? Well, it's the distance of the line to the origin - and since we know a and b already, we can solve for it by just plugging in one of our known points for x/y and then solving for c!
- That's well and good, but if we have 3 points, how would we get the equation for a plane?

- As we know, the equation for a plane is:

  $f(x,y,z) = ax + by + cz + d = 0$

  - If we have 3 points, we can get 2 vectors pointing along the edges, and then calculate the normal vector for the plane based on that!

    $E1 = P1-P0/|P1-P0|$
    $E2 = P2-P0/|P2-P0|$

    $N = E1 \times E2 / |E1 \times E2|$

  - This normal vector will give us the coordinates for a, b, and c - great!

    $N = (a, b, c)$

  - ...and, now that we know those coefficients, we can just plug in one of our original points for x/y/z and then solve for 'd'

- So, MATH (yay, I know). But why do we care? How does this help us figure out out our half-plane test?
  - (...Professor chooses to ignores this question in favor of a ray-tracing tangent)
  - Well, I suppose I'll have to put on my thinking cap and deduce the logic myself!
    - For a 2D triangle, we can compute the line equation for each pair of the triangle's vertices, giving us 3 lines (or "half-planes")
      - For each line, the points on the line will equal 0 when plugged into the equation; there'll then be one side of the line that's positive for plugged-in points, and another that's negative
        - We just have to make sure the point is on the correct side of each line, and bam! We know if it's inside our triangle or not!
  - Here's some example code, courtesy of Stack Overflow:

```
float sign (fPoint pt, fPoint v1, fPoint v2)
{
  return (pt.x - v2.x) * (v1.y - v2.y) - (v1.x - v2.x) * (pt.y -
  v2.y);
}
```

```
bool PointInTriangle (fPoint pt, fPoint v1, fPoint v2, fPoint v3)
{
   float d1, d2, d3;
   bool has_neg, has_pos;

   d1 = sign(pt, v1, v2);
   d2 = sign(pt, v2, v3);
   d3 = sign(pt, v3, v1);

   has_neg = (d1 < 0) || (d2 < 0) || (d3 < 0);
   has_pos = (d1 > 0) || (d2 > 0) || (d3 > 0);

   return !(has_neg && has_pos);
}
```

- Let's say we're in a 3D scene with orthonormal coordinate axes (u, v, w), and a "focal length" of distance 'd' away from our camera, which is positioned at some point 'e' in the scene
  - At the focal-length distance d away from our camera, we'll have our view plane, representing the pixels on our monitor. If we assume our camera is pointed in the -U direction, and our monitor has pixel dimensions (w, h) and the view plane is from [-1, 1] in the UV plane, how do we go from one coordinate to the other?
    - Well, pretty simply, as it turns out; for a given pixel coordinate on the screen (i, j), we can translate it to the view plane like so:

      u = -1 + 2i/w
      v = -1 + 2j/h

  - With that done, how do we calculate our "eye rays" shooting out from the camera for a perspective projection?
    - Well, ALL of the rays will have their origin at the eye's position, 'e'
    - The ray direction vector will then be based on scaling the different orthonormal vectors:

      dir = a*w + b*u + c*v

    - Where a = d (our focal length), b = u (the U-coordinate of the

pixel in our VIEW PLANE), and c = v (ditto for the V-coordinate of the pixel in the view plane)

- Specifically, it would be "-dw + Uu + Vv" in this case, since

- Since the view plane goes from [-1, 1], and the distance from the camera to the plane is 'd', we can calculate our field of view for the camera 'theta' (the total angle from the eye to the top/bottom of the view plane) using the equation:

tan(theta/2) = 1/d =>
d = 1/tan(theta/2)

- "We can do orthographic projections with ray tracing too, of course, but it's not too different from what we did here, and it's very seldom done in practice with ray tracing"

- Now, there's on more thing that's ESSENTIAL for proper ray tracing that you might not think about at first: recursion!
  - Why is recursion important? Well, think of reflections: if we're looking at a point in the mirror, how do we calculate reflections using ray tracing? By bouncing a second ray from where our original 'eye ray' hit!
    - So, the origin of this ray'll just be where we hit on the object, but how do we calculate the reflection direction for this ray? It's actually the VERY similar to the math that we did for Phong reflection! Look here:

R = E - 2(N*E)N

- Where R is the reflected ray we want, N is the normal vector, and E is the original eye vector that hit the surface (N/E should both be unit vectors)

  - When we're calculating the color for any given pixel in ray tracing, we usually consider it as the combination of a few different things:

color = ambient + diffuse + specular + k_refl*c_refl

- Where 'k_refl' is the reflection coefficient (how close is the surface to a perfectly reflective mirror?), and 'c_refl' is the color of the reflected ray
  - Sometimes, we might also want to include color/coefficient terms

for the 'TRANSMITTED' or REFRACTION ray (we'll talk about this in more detail later, when we talk about transparent surfaces)

- This color equation is where the recursion for ray tracing comes in: in order to calculate the 'c_refl' color, we need to call the color function again, shooting off a new ray - which might then lead to another ray, which might then require ANOTHER ray, and...
  - So, we need to decide how long this ray-bouncing recursion can go on before we stop it, and there are two main schools of thought on when we should stop:
    - When the "depth" of the ray-tracing recursion hits some maximum value (i.e. we just say '6 reflections are good enough', and stop recursing past this point)
    - When the contribution to the 1st ray becomes too small to care about (i.e. when the overall change to k_refl becomes small)
      - Practically, most people use the first one, but you can go either way

- So, as we've described this algorithm, ray tracing actually seems BACKWARDS from how the real world works: instead of photons bouncing into our eyes, we're shooting rays out of our eyes like a superhero!
  - This REALLY bothers some people, and there are alternative algorithms that actually do involve bouncing light rays into the camera - but for the most part, people view this eye-ray-shooting as a good-enough compromise

- So, this is great for diffuse surfaces. It's great for shiny surfaces. But what should we do about TRANSPARENT surfaces?
  - Let's say we have some transparent object we'd like to render, like an ice cube; how does light behave when it hits this object?
    - Well, per refraction, the light will BEND: a small part of the light will be reflected (depending on the angle), but the rest of the light will go through the object and shoot out the other side - and because light travels at different speeds through the material, the light will bend based on how slowly light travels through the material:

      Index-of-Refraction = speed of light in vacuum / speed of light in medium

      - Some common IORs:
        - Vacuum: 1.0
        - Air: 1.0003

        - Water: 1.33
        - Ice: 1.309
        - Glass: ~1.5
        - Diamond: 2.417
    - So, this index controls how much the light bends in the material: the higher the index, the more the light bends!

- We'll dive into the details of how we calculate this next class - in the meantime, adieu!

//**/
/
//********* Ray Tracing Shadows & Effects - February 20th, 2019 **********//
//**//

- Alright, the midterm is going to be THIS MONDAY, February 25th
  - Weirdly, for the first time ever, we'll be taking the test online on Canvas, so BRING YOUR LAPTOP! It'll be open-note and open-book!
    - The test will be based on the topics we've covered in lecture, up to the stuff on Friday's lecture
    - Most of the questions will be multiple-choice/short-answer, with a few true/false questions ("what does this code snippet do?", etc.)
      - Most of the questions will be conceptual, but there might be a few along the lines of "do these matrix transformations," "what does the final matrix stack look like after these function calls?", etc.
      - Mercifully, Professor Turk has said he doesn't know how to add long text boxes to the online quizzes, so he won't ask any coding questions

---

- Alright, we left off yesterday by trying to figure out how raytracing can deal with transparent surfaces, like glass or water (or a glass of water)
  - Part of the ray is reflected, like with specular surfaces, but the other part of it is TRANSMITTED through the surfaces - and due to the speed of light being different in different materials, it'll bend by a certain amount based on its index of refraction (IOR)
  - How much will it bend by, though? That's based on something you might remember from physics called SNELL'S LAW:

sin(theta_incident) / sin(theta_trans) = IOR_trans / IOR_incident

- Where "theta_incident" is the angle that the ray makes w/ the normal before it enters the material, and "theta_trans" is the angle the transmitted ray will make with the surface normal as it passes through
    - In general, a ray passing into a material with a higher IOR will bend towards the normal, and vice-versa
- Another feature of these materials, though, is a thing called "total internal reflection:" if a ray is traveling through a material and is about to enter another one, but the angle is above some "critical angle," it'll actually bounce off instead and stay in the material!
    - This is critical to how fiber optic cables work, for example

- So, with all this stuff considered, what should the color of our eye ray be now?
    - Here's our updated equation (I think? DOUBLE CHECK THIS):

color = ambient + diffuse + specular + k_refl*c_refl + c_trans

- Alright, so we've spent a lot of time talking about light in this class - but what about its opposite? What about SHADOWS? Well, let's shine a "light" on the topic now (HAHAHAHAHA -_-)
    - We'll start off with "hard shadows," with clearly defined, non-blurry edges
        - Let's suppose we have some point light source in our scene, and there's an object in front of another object that should be casting a shadow - what can we do?
        - Basically, when we first shoot an eye ray onto a surface, we shoot a "shadow ray" from the impact point to each light source in the scene - if we can reach the light's pont without hitting anything, hooray! The point is lit up!
            - If our shadow hits another object BEFORE it reaches the light, though, that means that we can't "see" that light source from that point - so we don't add its contribution to the lighting at that point
    - Cool! But more commonly in the real world, we see SOFT SHADOWS that are darker near the center of the shadow, then fade out to light towards the edges
        - This occurs when we have light sources that aren't just infinitesimal points, but have a non-zero area <insert outdated thine-mother joke

here>, like a fluorescent tube
- For these shadows, there'll be an area of darkness where can't see ANY of that light source's parts, known as the UMBRA (total shadow)
    - As we move out from behind the object's shadow, though, part of the light source's area will become "visible," and the surface will only be partly in shadow - this is the shadow's PENUMBRA (partial shadow)

- Alright, these are the two types of shadows, so how do we calculate what a light's contribution to a point is NOW? Well, like THIS:

    Final Color = Cl * Cr * (N*L) * Visible(P,L)

- Where:
    - Cl is the color of the light
    - Cr is the surface color
    - N is the surface normal, L is the vector to the light
    - Visible(P,L) is 0 if the light 'L' is blocked and point 'P', 1 if the light is unblocked
        - We COULD write this as an if statement, but being able to have fractional values between 0 and 1 for this'll be useful
        - Determining visibility is easy in the hard shadow case (just shoot a ray from P to L, 0 if blocked, 1 if unblocked), but how do we determine what percentage of the light is visible for an area light?
            - Well, for an area light, we'll shoot "n" shadow rays to equally-spaced points along the light source, and record if the light is visible or not for each ray
            - Then, we'll average their visibilities like so:

                Visible(P,L) = 1/n * sum{CLi * Visible(P, Li)}

            - This is one form of "distribution ray tracing," which is also done for other effects like motion blur, blurred reflections, depth-of-field blur, etc.
                - If we shoot too many rays, it'll really slow stuff down; if we don't shoot enough rays, the penumbra will look grainy and discretized ("it just looks like multiple point lights are lighting it")
                - "Why would we want motion blur, though? It makes my real-world photos look blurry, Greg!" Well, because it

makes the image look more realistic - perfectly in-focus images look fake to our object, ESPECIALLY in the VFX world where you have to merge CGI with real footage

- So, which type of shadow is faster to render? HARD SHADOWS, since we only have to shoot one ray to each light source!
  - Soft shadows, though, tend to look much more realistic; we get the effect of shadows getting blurrier with distance for free, and in general it just matches real life a lot better

- Now, shadows are certainly very important, but how can we use distribution ray tracing to do other effects?
  - Let's think about glossy reflections, for instance, where we don't have perfectly smooth reflections
  - The "traditional" way of getting the surface color for a reflective surface is something like this:

    C = ambient + diffuse + specular + k_refl*Cr

    - Where 'Cr' is the reflect color (based on the reflected ray), and 'k_refl' is the coefficient for how much of that reflection should come through
  - To do it the distributed way, we do ALMOST the same thing, but replace the single "k_refl*Cr" with the average of multiple rays, to simulate the scattering of the reflected light:

    C = (...) + 1/n * sum{ k_refl * Cri}

- Cool! Now, what about motion blur?
  - To do this, we need to distribute our rays not in space, but in TIME; our ray might hit the object at T=0 but miss it at T=1, for instance, and if we want that to show up in our frame as motion blur we need to average the two colors together
  - So, we'll go from our "traditional" color:

    C = Cl*Cr*(N*L)

  - To this:

C = 1/n * sum(Cl*Cr_t * (N_t*L_t))

- As we advance the object in time, we need to average the color of the pixel across these "n" different times, which'll change if the object is moving
  - Usually, you'll need at least n ~= 20 so the blur effect looks smooth, instead of like a weird ghost-like object hovering over your scene

- That's all the time we've got today, so FLEE!

//**********/
/
//********** Raytracing Optimization - February 22nd, 2019 **********//
//**********//

- So, we have the gift of a midterm bestowed on us this Monday, the 25th - it's open note/open-book, calculators are allowed (though you probably won't need it), and it'll be online, so BRING YOUR LAPTOP!!!
  - The test will start promptly at 12:20, and will be cut off at 1:10
  - Basically the only restriction is that, while you can use notes on your computer, you can't look up resources online during the test
    - What're the test topics?
      - Look at the reading list; we won't talk about radiometry or curves, but everything else on there is something we've discussed
        - For reference, here's the said list:
          - Pixels / display technologies (LCD, E-ink, etc.)
          - Matrix / vector basics
          - Transformations and the matrix stack
          - Viewing and projection transformations
          - Rotation in 2D / 3D
          - Lines and rasterization
          - Hidden surfaces
          - Color, lighting and shading
          - Ray tracing
      - Content from the previous lecture, and today's lecture, won't be on the midterm; everything else is fair game

---

- So, we've been talking about raytracing, but let's say we want to add a brand

new primitive/shape; what do we need to do to render that shape?
- First, we need to intersect a ray with the object's surface
- From there, we need to return the intersection point and the surface normal of where we hit
- Then, at last, we return the bounding box for the object
  - Why is this important? We'll explain that in due time (i.e. 20 minutes)

- Some objects are easy to raytrace: spheres, triangles/polygons, and cylinders are pretty quick to render, boxes and ellipsoids aren't too bad...but there are others where the story isn't as nice
- Toruses take MUCH longer than normal; instead of having to solve a quadratic equation to find the intersection point, we have to solve a quartic one, which makes it far slower to render than, say, a sphere
  - Other objects that take awhile to raytrace include:
    - Subdivided surfaces
    - Cubic paths/curves
    - Fractals
    - "Blobby spheres" (or "metaballs")
    - Collections of polygons
      - "What? Aren't polygons supposed to be fast?" Well, each individual polygon might be quick, but anything in large groups can take awhile

- So, if we want to make industrial-strength raytracer, how do we deal with these slow-to-raytrace objects efficiently? We can't just ban them from existence!
- Let's remember our raytracing loop:

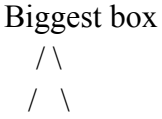
```
for each pixel:
    create ray through pixel
    for each object in scene:
        intersect ray with object
```

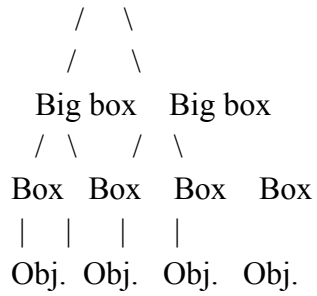
- This loop is the source of all our slowness; Z-buffering only checks the intersection for the current object for all the pixels that might be rasterized, but with ray-tracing, we have to check ALL of the objects for each pixel!
- So, how can we check all our objects more efficiently? Fortunately, this question has been around for awhile, and there're a number of techniques we

can try

- First off, we can surround each of our objects in an invisible BOUNDING BOX; this way, instead of trying to solve a complicated object's equation just to figure out if our ray's near it or not, we can just check against all the easy-to-calculate bounding boxes FIRST
    - If our ray doesn't hit an object's bounding box, then we know it won't hit the object itself, so we can skip it!
        - Otherwise, if it does hit the bounding box, we'll just calculate the object's intersection like normal
    - More generally, we don't HAVE to use a box; any simplified "bounding volume," like a cylinder or sphere or ellipsoid, can be used
        - Ideally, we want to use the fast primitive closest in shape to our object to avoid "false positives," where the ray hits an object's bounding box but not the object itself (e.g. ray goes through the hole of a donut)

- So, bounding volumes let us speed up raytracing by skipping ONE object - but what if we have multiple objects, all of them complex to raytrace?
    - Well, if we have a group of complex objects (e.g. a bicycle chain,) we can surround all of their bounding volumes in ANOTHER big bounding volume; that way, if the box isn't hit, we can skip ALL of them!
        - Of course, if that volume is hit, we need to open it back up and check each object's bounding box like normal
    - What if we put this box inside another, larger box? We can keep doing that, too!
    - We'll say this larger box is the root of a BOUNDING HIERARCHY, and is the parent of some smaller boxes "B" and "C" inside it, which are themselves parents of other boxes, which are themselves...well, you get the idea. We do this all the way until we get down to the individual bounding volumes themselves, which contain the "real" object
        - So, if we had a chain made up of 4 toruses (tori?), each of them might have their own bounding boxes
            - Then there might be 2 meta-boxes that each hold 2 toruses, and then an even bigger box that holds both of those, and acts as the parent, making a dependency tree like so:





- Even in this small case, we're now only checking our ray against at MOST 3 bounding boxes and 1 torus, which is MUCH faster than checking all 4 toruses!

- How do we actually do this, though? Let's take a peek at some pseudocode:

```
hierarchy_traverse(ray r, node n):
    if r intersects n's bounding volume:
        if n is a leaf
            // We've reached the object itself
            intersect r with n's actual object
        else:
            for each child node c of n:
                hierarchy_traverse(r, c)
```

- Notice that if we don't intersect anything, we just stop traversing the tree - we're done, and can ignore everything else!

- So, once we've got this bounding hierarchy tree, we can get MASSIVE speedups in rendering - but how do we construct the tree itself?
  - There's a LOT of different opinions, but most of them fall into one of two camps:
    - The "bottom-up" strategy
      - Let's say we have a scene with thousands and thousands of spheres, all scattered around; for each sphere, we ask "where's the nearest un-grouped sphere?" and put them both in a bigger box
        - Once we run out of pairs, we then do this for pairs of pairs (i.e. the next level up on the hierarchy), then pair THOSE boxes together, and so on, until we've eventually got everything contained under one giant box
    - The "top-down" strategy
      - Here, we start off by having one big bounding volume, covering the whole scene, and ask "how many objects are in this box?"

- If there's some arbitrary "n" or less objects, we say that our cube is sufficiently divided
- Otherwise, if there's still too many objects inside our box, we'll divide it up into evenly-sized sub-cubes (usually 8), each parented to our original "big box," and recursively do the same thing to all those sub-cubes until all of them are sufficiently small
- What if objects are on the edges of these cubes? There are a few different ways to handle this,
- This is probably the more common strategy today
- While sometimes one strategy is better than another, both of these techniques tend to give pretty similar results; creating an "optimal" tree is an NP-hard problem, but getting a good-enough tree is very doable

- These are the most prominent methods for speeding up raytracing, but certainly not the only one:
- The GRIDS method, for instance, involves "spatially partitioning" (i.e. chopping up) our scene into a 3D grid of small, evenly spaced cells, each of which has a list of the objects inside it
- So, for each cell our ray passes through, we ask "are there any objects in this cell?" If there aren't, we just skip it!
- If there are objects in it, we check if we intersect each of the objects in that cell, and stop once we have our first "real" intersection
- This might seem like a lot of wasted work (all those empty cells, right?), but it lets us skip a ton of intersection-checking, which is great!
- We end up moving from cell-to-cell in a way very similar to our 2D line rasterization methods
- This method is GREAT when we have a lot of similarly-sized objects in a scene to deal with, but it doesn't work as well for scenes where the objects are at wildly different scales; large objects have to be checked entirely, while small objects all get grouped in the same cell
- K-D trees is another technique for splitting up the spatial search space, but we won't get into the details of them just yet

- Raytracing is still slower today than rasterization, but thanks to these and other techniques it's become fast enough to be practical for many applications - and today, it dominates the special effects industry.

- So, your homework is due on Saturday, and the midterm is coming up on Monday
- good luck to you on both of them, and study hard!

//**********/
/
//********** Rasterization Effects - February 27th, 2019 **********//
//**********//

- Okay, project 3A has been posted on Canvas; this is a MUCH more difficult project than the previous two, so PLEASE start early; I don't want anyone to start working on this the night it's due and get overwhelmed
    - Note from the future: it wasn't.

- With the midterm past, we're actually all done with raytracing! We spent quite a few lectures on it - so now, let's turn back to rasterization

- Raytracing can handle cool effects like shadows, reflections, etc. pretty easily, but without raytracing, how can we do those things? How, for instance, do most games create shadows?
    - Well, there are two main techniques for creating raster-style shadows today: "shadow volumes," and shadow mapping
        - Shadow volumes are the more confusing technique for most people, but both methods are about as equally effective
    - As we noted with raytracing, shadows are all about visibility; if the light can't "see" a surface, then that surface is in shadow
        - ...so, what do we have in our bag of rasterization tricks to deal with visibility, if something is 'hidden'? The Z-Buffer!
    - Not ALL shadows deal with visibility issues; there are ATTACHED shadows, where the backside part of an object doesn't face the light (e.g. the dark side of the moon)
        - Luckily, these are already handled by our shading equation
    - CAST shadows, though, are when a different object blocks light from reaching a surface behind it, and definitely do require visibility checks
        - If an object is concave, it can create a cast shadow on itself; if it's convex, though, we only need to worry about attached shadows when considering a single object

- How do we actually create these shadows? Let's look at the algorithm for SHADOW MAPPING (specifically, the Lance Williams-created 'Two-Pass Z-Buffer'

method)

- The basic idea here is to first render the scene from the point of view of the LIGHT SOURCE
  - Then, render the scene normally from the eye/camera...
  - ...and determine what pixels should be in shadow for our camera, based on which pixels were hidden from our light source
- So, we're doing a "light space" render and an "eye space" render, but how do we map the hidden pixels in our light space to the shadows in our final render?
  - Think about how this mapping is working: there's a 3D world space where the point we actually care about lives. For the first render, that pixel is getting mapped onto "light space" with the mapping 'L', while in our 2nd render it's getting rendered to our "camera space" with a mapping 'V'
  - So, what we'll do is the following:
    - Get the pixel we're rendering in our camera view, and use its ACTUAL position in the world space
    - Then, test that point's Z-position against the light's Z-buffer value; if it's less than the Z-buffer, it isn't visible to the light, so it should be in shadow!
      - (Me, confused: wouldn't this change with direction, though? It might not always be the Z-position, right? (although it should be a pretty simple distance check))
      - So, in short, with a given pixel 'P,' we map it to the world space, then to the light space, with the following:

        $$P' = L*V^{-1}*P$$

        - (not sure I understand the details of how you get these mappings/reverse them?)
  - How do we figure out which way to "point" the light's camera? In the worst-case scenario, we might have to create 6 images (a CUBEMAP) to have information for every direction; if we know some directions aren't needed, though, we can fiddle with this

- Now, there's a problem here: if we have to render the scene again for EACH light source, that means every light source we add slows us down!
  - Basically, shadows require visibility checks, and visibility checks are slow; there's no free lunch for us shadow-drawers

- With that, let's shift gears to another question: how do we put textures on objects?
  - This might not seem related, but we have to do a similar thing as with shadows: we have to map the texture image's coordinates onto a given "texture space" (with the color defined at different "TEXELS"), and then render that onto the screen
    - So, we look at the texture, see it's a white or blue texel or what have you, and then render it
  - At a high level, the process looks a little like this:
    1) Map and interpolate the texture's S,T values (basically, the image's coordinates) across the polygon during rasterization
      - These S,T coordinates are in the range from [0, 1], and are sometimes called "UV" coordinates instead
    2) Find the color from the texture at that point ("texture lookup")
    3) Color the pixel based on the texture color of the object at that point (plus all our usual lighting+shading doohickiness)

- Let's try to go through the first sub-stage
  - Suppose we have a simple triangle polygon, with texture coordinates (s1,t1), (s2,t2), (s3,t3); VERY similarly to Goraud interpolation, we then (for a given point during rasterization) interpolate these S-T texture coordinates across the polygon, in preparation for looking up the actual color in the next step
- HOWEVER, there's a hidden gotcha here that has to do with perspective projection...but we can talk about THAT more on Friday.

//**********/
/
//********** Texture Mapping - March 1st, 2019 **********//
//**********//

- February officially went cold on the operating table, so now it's March's turn! Give it a hand!

- Alright, we started talking about texture mapping on Wednesday - let's take a look at the gory details!
  - As mentioned, there are 3 steps to this: interpolating the S/T values across the polygon, looking up the color at that point, then rendering it

- Let's begin at the beginning with interpolating the S-T values
  - Now, sadly, perspective projection makes this a little more complicated than straight-up Goraud interpolation; we instead do the following steps:

    1) Divide S/T by |Z| to get s' and t'
    2) Interpolate s' and t' linearly during rasterization, just like Goraud!
    3) Divide by Z' per-pixel
    4) Look up the color of the textured surface

  - If we don't do this, we get weird effects where the texture looks like it's sliding around from different perspectives, etc.; we can't just interpolate

- Texture lookup should be easy though, right? We just take color of our texture at the given S/T value, right?
  - NOPE! If we just did that and took the nearest color in the texture to our coordinates, we'd get a blocky/pixelated effect, which usually isn't desirable
  - To fix this, we'll instead use BILINEAR INTERPOLATION to take the weighted average of these colors
    - For instance, let's suppose we have a tiny 4x4 texture image - "remember, we're using S and T coordinates, not x and y"
    - If we have an S-T coordinate of, say, (1.7, 1.2), then instead of just taking the color of pixel (2, 1) and calling it a day, we'll instead interpolate between the colors of the 4 nearest pixels (i.e. (2,1), (1,1), (1,2), (2,2))
      - Here, we'll have a ratio "a" for our percentage distance to the right pixel (e.g. in this case, 1.7 - 1 / (2 - 1) = 0.7), and "b" for the percentage towards the top pixel (i.e. 1.2 - 1 / (2 - 1) = 0.2)
      - With this, the bottom horizontal color "r0" and the top horizontal color "r1" will be found as weighted averages for the top/bottom pairs, e.g.:

        r0 = r00 + a*(r10 - r00)
        r1 = r01 + a*(r11 - r01)

      - Then, the final color will be the interpolated average of THOSE averages using the vertical ratio:

r = r0 + b*(r1 - r0)

- This interpolation is ESPECIALLY important when we're close to an object, where the blockiness of our textures becomes really obvious (known as "texture magnification")
  - The opposite of this is texture MINIFICATION, where a texture is so far away that only a few pixels of it are showing on-screen
    - The problems with this minification problem aren't as obvious, but they're definitely there!
      - Imagine you're on a spaceship that's blasted off from earth, and you're watching the earth shrink smaller and smaller in the rear-view mirror (...of your spaceship...yes...) - we'd imagine that the earth should be predominantly blue and green and white, right?
      - But if we don't properly shrink the texture as you get farther away, the colors from the texture will get chosen at random, and it'll "sparkle" as we get really far away and appear noisy
        - In extreme cases, they can result in Moire patterns for textures with parallel lines (which is fun, but NOT what we want)
  - How do we deal with this? Basically, we'll create multiple textures of different sizes, known as "MIP-MAPPING" or an "image pyramid"
    - Mip-mapping, specifically, is a technique created by Lance Williams, where we successively scale down our texture while averaging its pixels
      - For instance, if we start out with a 128x128 pixel texture, we would create a smaller 64x64 version by making each new pixel's color the average of its corresponding 2x2 group of 4 pixels in the original
      - Then, we'd create a 32x32 image from the 64x64 one, then a 16x6 one...etc.
    - The image pyramid technique comes from a different way of viewing this as "stacking" these different sized images together
      - ...and if we ALSO use our different image-size "layers" to calculate the color of our texture (i.e. we interpolate between the next-largest layer's color, using a distance-based factor as our ratio), we end up with TRILINEAR interpolation, which helps prevent the "sparkling effect" event more
  - How do we choose which size texture to use at a given time? Well, to be hand-wavey about it, it's based on a function of your distance from the

texture
    - To give some more detail, what we're trying to mimic with all this work is - for a given pixel on our monitor - to "project" that square pixel from screen space onto the surface where our texture is, then pretending that stretched/distorted pixel is on our texture itself (i.e. in "texture space") and finding the average color it should be
    - To approximate this, we'll pretend that projected pixel is an ellipsoid, calculate how "stretched" the ellipse is on its two axes, and say the distance factor "d" is ~= sqrt(ds/dx^2 + dt/dy^2)
- Furthermore, though, how do we STORE all these extra textures efficiently? Lance Williams came up with a decent way of doing it:
    - Each image has 3 color channels, right (RGB)? Well, we'll make a new image and divide it into fourths, then put our original image's RGB values SEPARATELY into a box for each color, with the fourth square left empty; we'll then repeat this process for the smaller texture, putting it in the empty square, then repeat...

- Some of you might end up taking classes latter that talk about using "low-pass filters" (e.g. the Gaussian filter) to blur these image colors in a more sophisticated way than the bilinear interpolation we're using

- Now, mipmaps are great and efficient and supported by most graphics cards today at a hardware level, but they're not the only game in town by a longshot
- there are other, more elaborate methods that try to have better quality
    - Similarly, there're LOADS more color interpolation techniques than just bilinear vs trilinear that try to be more performant, or better-looking, etc.

- Switching gears a bit, in raytracing, reflections were GREAT - we got them almost for free! - but they're harder to do for real in raster rendering. Instead, we'll try to cheat and fake our reflections with "cubemaps"
- We'll talk about that - and more! - on Monday

//**********/
/
//********** Reflections and Environment Maps - March 4th, 2019 **********//
//**********//

- ALL THE WORK AGGGHHHHHHHHHHHHHHH!
- Alright, some things to note for the raytracing homework:

    - When you do the ray-sphere intersection, sometimes it's useful to return a "hit" data structure containing the point of intersection, the parameter "t" along the line where the intersection was, the normal of the surface we hit, and a reference to the object's material
        - "But Greg, I want to know the color of the sphere!"...but that's usually better done later on in the process, so you can separate intersection and shading
        - So, we might have a "shade_intersect(hit)" function later on
            - "You don't have to do your raytracer this way, but it'll make a few things easier for part 2"
    - A few questions:
        - Our eye is looking in the -W direction, V is pointing "up," and U is pointing to the side
            - So, the direction of the eye ray is "dir = -dW + uU + vV"
        - How do you debug this thing? You DON'T print out all the coordinates where the ray hit, since there'll be waaaaay too many; instead, make some global flag for a particular pixel and ONLY print out the ray/hit information for that particular ray; that way, you can calculate what should be happening by hand
        - For part A, just do N*L for each light source and don't worry about visibility; in part B, though, you'll need to start worrying about shadows
    - If you have any questions later, please visit the TAs - they're ready-to-go in office hours, and they want to help!
- As a side-note: PLEASE start the raytracing homework tonight if you haven't started yet; it's a challenging homework, and I don't want any of you to get overwhelmed because you started it Friday night!

- If you remember back to Friday ("I barely can"), we were talking about texture lookup, where we were doing bilinear interpolation to get the color values for our texture
    - In that case, we were interpolating the colors of our texture, but here's a thought: what if, instead, we did the same thing for heights?
        - Imagine if the image were a heightmap, with each of the color values at the pixel representing a height (like a bunch of bars sticking out a certain amount)
        - In that case, bilinear interpolation would get us a curved surface that connects all of those sticking-out rods
- As it turns out, bilinear interpolation is used EVERYWHERE in computer

graphics, not just to interpolate colors!
- So, having a good idea of the shape of the bilinear surface is really useful to have
- Keep this thought in the back of your skulls...

- Now, we also talked on Friday about how we wished we could "fake" reflections in our raster scenes - let's talk about that more now, using ENVIRONMENT MAPPING
- As we mentioned in raytracing, reflection is when we have a ray "R" that hits a surface and bounces off
- Environment mapping is where we instead imagine that we take a bunch of pictures of the environment around a point, and paste those photographs onto the walls of a giant sphere
    - After that, we can take away all the actual objects and just leave the wallpapered-on sphere, and if we're standing in the center of the sphere, we can't tell the difference!
- As it turns out, this technique is enough to fake reflections without having to do any raytracing!
    - Here, we'll take the view vector from our eye to a given point on the reflective surface, and then calculate its "bounce direction" using the normal vector to get a vector 'r'
    - Once we've got that bounce direction, we'll use it to look up the color from the environment map we want - this'll be our "reflected" color for the surface!
- Using an actual sphere for the environment map, though, is difficult, since it's hard to "unfold" a sphere to put textures on without distortion (think of all the weird projections mapmakers use!)
    - Instead, most implementations actually use a cube map instead of a sphere, "unfolding" the 6 faces of the cube and rendering the scene 6 times, in the 6 different directions for the cube
- The reason this is "fake," of course, is that we're pretending all of the reflected objects in the scene are infinitely far away and flat, instead of adjusting our reflection based on the actual positions and depth of the objects - we couldn't use this for dynamic reflections, since everything is "stuck" in the position we rendered it at (although you can try to re-render it on the fly)
- Because of this, environment mapping works great for scenes where everything's far away, but looks a little bit off if we're supposed to be close to the object
- Still, this was a good enough solution that many films used it for early CGI effects, and most games still use it today

- So, that's reflections, making an object look shiny and smooth - but what if we wanted to make an object look bumpy? How could we do that?
  - The "most correct" answer is to make a bunch of new, tiny polygons - but that's expensive!
    - WHY do those extra polygons make a difference to how our object looks, though? Because they give us different surface normals, right?
  - So, in 1978, the same Blinn who came up with our lighting equation decided to try approximating these little changes in the normals WITHOUT creating new polygons. Instead, the way the normals should change are defined in an image called a BUMP MAP
    - The idea behind this is that we can have an image that shows the "heights" of our object's surface; we can then find the slope/derivative of that imaginary surface, calculate a fake normal from that, and then add those normals to our object's real normal for the given pixel!
    - This let's us change the shading of a surface JUST using an image - that's great!

- What does the math behind this actually look like, though? That's something we'll look at a bit more closely on Wednesday; come hear all about it then!

//**********/
/
//********** Bump Maps and GPUs - March 6th, 2019 **********//
//**********//

- On Monday, we were talking about bump mapping: using an image to make a surface look bumpy without adding any polygons, by using the image to "wiggle" our normals
  - Using bilinear interpolation, we'll have a function "f(u, v)" that gets us the slope of the "surface" the image is representing; we'll then be able to calculate a normal using that slope
    - If the function is 1D, we can do this based on subtracting it from a straight-up default normal vector N' and having a tangent vector T = (1, 0):

      $$N = N' - f(x)*T$$

    - The bump map image itself will be a grayscale image, where dark colors mean that pixel is lowered and light colors mean that pixel is

raised
- For the 2D case, we'll have two different derivatives, one in the "up" or "v" direction Bv, and another in the "right" or "u" direction Bu
    - We'll say that "B" is the height of our surface at a given point, meaning the slopes are:

        Bv = dB/dv
        Bu = dB/du

    - To calculate our normals now, we'll say our normal vector is based on subtracting from the straight-up "default normal" N' vector, like so:

        N = N' - (Bu*Ps + Bv*Pt)

        - Where Ps is the tangent vector in the u/s direction, and Pt is the tangent vector in the v/t direction (actually calculating these vectors will vary, based on the surface)
            - How do we calculate the derivatives "Bu" and "Bv" in the first place? We can do something called a "finite difference calculation," where we literally just take the slope between one pixel and the next one
            - We're leaving a few details out here, so be aware this isn't a be-all end-all discussion of bump mapping
- Now, because we're not changing the actual polygons of the object, the silhouette of our object will NOT change; the appearance of the surface will, but the outline will just look the same

- Now, the BIG advantage of raster graphics over raytracing right now is that they can be computed on the GPU much more easily - but how does the GPU work? What IS a "graphic processing unit"?
    - Well, when the host CPU of our computer wants to draw something, it'll send the primitives it wants to draw on-screen to the GPU
        - When that information gets there, the GPU will first feed them through a VERTEX PROCESSOR, which handles any 3D transformations, projections from 3D to 2D, etc.
            - Project 1 in this class was similar to the jobs a vertex processor handles
        - That output of screen-space primitives then gets fed into the RASTERIZER, which turns the primitives into FRAGMENTS
            - Fragments are "almost-pixel" versions of these images, but not

quite - they're "per-pixel data" instead (e.g. color data, texture coordinates, depth, normals, etc.)
- Pixels themselves are just the final output color, while fragments hold the information we need to calculate the color for that pixel
- Rasterizers are the least accessible part of the GPU; most GPU drivers let us write programs for the vertex processors and fragment processors, but rasterizers are pretty off limits
- Finally, those fragments all go into a FRAGMENT PROCESSOR, which turns those fragments into actual output pixels and handles any shading, texture mapping, etc.
- Where does all that texture information come from? It comes from a piece of "texture mapping" memory in the GPU, which holds that data

- Now, the GPU doesn't have a single monolithic "vertex processor," "fragment processor," etc. that everything has to go through at once; instead, GPUs really shine by allowing us to do processing in PARALLEL, so they'll have multiple of each of these components
- As an example, let's look at the older NVIDIA GeForce 6800 graphics card from 2004:
- This card had 6 vertex processors that could handle 3D transformations, per-vertex shading (i.e. Goraud shading), 4 multiplications/additions for each of its 4 "vector units", and do basic mathematical operations like square root, sine, cosine, etc.
- Each of these vertex processors operated completely independently from one another, known as "Multi-Instruction Multiple-Data" processing (MIMD)
- It had 16 fragment processors, each of which could do per-fragment shading, texture mapping, and had 2 vector units (each of which could do 4 multiplications/additions)
- Importantly, however, all of these processors acted in LOCK STEP; each one would handle a different fragment, but they still did the same step at the same time
- This is known as "Single-Instruction Multiple-Data" (SIMD) processing
- Why were these operations locked together? Basically, it took time to get separate instructions for each individual processor, so to cut down on this the instruction was just fetched ONCE and then fed to all the processors

- Finally, it had a single rasterizer, which handled interpolation across the polygon (colors, surface normals, etc.) and generated fragments for each polygon (per-pixel data)
    - Modern cards usually do have multiple rasterizers, but not as many of them as we do for the other components
- More modern GPU architectures, however, noticed that many of the operations on the vertex/fragment shaders were quite similar (e.g. multiplying, adding, etc.), and tried to combine them in a "Unified Graphics Architecture"
    - In these sorts of cards, there's NO distinction between vertex and fragment processors (although the rasterizer remained separate)
    - The first card to really do this was the GeForce 8800 (circa 2008)
        - It had 128 "streaming processor cores" (often just called "cores"), which mostly do adds/multiplies, floating point operations, etc.
            - Here, instead of locking all the cores together in SIMD operation, they were grouped into 16 sets of "streaming multiprocessors" (each containing 8 SP cores and 2 "special function units") that could each operate independently, with different instructions
                - The "special instruction" units handled the less common operations that used to be on the vertex processors, e.g. sine, cosine, logarithms, etc.
    - In NVIDIA's 2018 "Turing" architecture (circa 2018), there were 72 SMP units like this, with several separate "tensor cores" for optimizing AI processing and a special block for handling raytracing
        - What part of raytracing, particularly? It optimizes the "spatial partitioning" step, which is what lets us build those bounding hierarchies we talked about efficiently

- Alright, that's a LOT of information dumped on you all at once, but here're the big takeaways:
    - GPUs really emphasize parallel processing; the individual cores don't do a whole lot, but there's a LOT of them, letting us do a lot at once
    - (...that's really the biggest one)

//**********/
/
//********** Introduction to GLSL/Shaders - March 8th, 2019 **********//
//**********//

- "If you're coming into the room right now, yes, it's unbearably hot - the AC's broken"
  - They're trying desperately to prop open the the doors to get some air flowing in...
- "If you're curious about if you have raytracing in your GPU, just ask yourself the following question: 'Did I pay more than $1000 for this piece of hardware?'"

- Alright, we talked a bit about the architecture of current graphics cards yesterday - now, we're going to switch gears a little bit and talk about the software we can write for these cards
  - In particular, there're "shading languages" we can use to write programs for these cards, like the GLSL language (for OpenGL), HLSL (for DirectX), etc.
    - In this class, we'll mostly be focusing on the GLSL language
  - Here's a basic overview of how information "flows through" the programs we'd write in GLSL:
    - We start out by getting some OpenGL input variables from our graphics program, like the screen-space vertex positions, color, normals, and texture coordinates
    - A VERTEX PROGRAM we write in GLSL then takes that information for each vertex and spits out the 2D vertex position and the colors for the front/back faces of the polygons, and passes along the texture coordinates (usually unaltered)
      - Realistically, separate front/back colors for a polygon aren't used very often
    - From there, the output is passed along to the rasterizer, which interpolates those color/texture values across the polygon and turns them into fragments, containing the color and texture information for each pixel (amongst other things)
    - Finally, those fragments go into a FRAGMENT PROGRAM, which operates on them and changes a given fragment however we'd like before it goes to the screen
  - At a lower level, of course, each of these stages might have multiple sub-stages, physical components, different nuances, etc., but this is the general "pipeline" we can think about

- That's the "process" for how a GLSL program is used, but what IS GLSL anyway?

- Well, it's a C-like programming language that's used to define these graphic operations we care about
  - It has some standard datatypes like int, float, bool, etc., but also some graphic specific ones:
    - Vec2 (e.g. texture coordinates)
    - Vec3 (3D positions, normal vectors, RGB colors, etc.)
    - Vec4 (homogenous coordinates, RGBA colors w/ transparency, etc.)
    - mat3/4 (3x3 matrix and 4x4 matrix datatype, respectively, used for transformations)
    - uniform sampler2D (e.g. texture maps)
      - "This is a LOUSY name for a datatype, if you ask me; it should've just been called a texture map, since that's what it's almost always used for"
        - The 'uniform' keyword means we're going to use the same value for every fragment/pixel, which makes sense for things like textures; 'varying' means the information will get passed through the rasterizer and be interpolated for EACH pixel separately (e.g. normals will be interpolated on a per-pixel basis, etc.)
      - Also, yes, this is two separate words
  - Similarly, it's got your standard operators like +, -, *, %, =, etc., BUT some of the operators are overloaded (for instance, * will multiply two matrices by default, scale vector by a scalar, etc.)
    - There are some standard functions you can use without importing anything, like sin/cos, abs, floor/ceil, min/max, pow, sqrt, etc.
    - There are also plenty of vector operators, like length, dot, cross, and normalize
    - There are also some WEIRD ones, like "inversesqrt" (which is typically used to get)
      - Another weird one is "texture2D," which takes in a sampler and a texture map

- Let's look at an example GLSL program for implementing a diffuse shader - "most hardware nowadays supports this natively anyway, but it's a good teaching example"
  - Typically, when you write a vertex shader, you'll also write a fragment shader - you don't typically get one without the other
  - Here's what the vertex program would look like:

```
varying vec3 normal; // varying means this'll be different for each
```

```
pixel
varying vec3 vertex_to_light;

// Note that variables like "gl_Normal" are predefined and
implicitly passed to the shader without us needing to explicitly
specify them as inputs/outputs

void main() {
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   //transforms our polygon's vertices into 2D; this matrix is
   pre-defined in the language
   normal = gl_NormalMatrix * gl_Normal; //converts the normals to
   2D
   vec4 vertex_modelview = gl_ModelViewMatrix * gl_Vertex;
   //converts it to a 3D position, but does NOT project it to 2D
   yet - we'll use this to get a vector to our light source
   vertex_to_light = vec3(gl_LightSources[0].position -
   vertex_modelview)
   // we won't bother normalizing these variables, since the
   rasterizer output isn't unit-length anyway
}
```

- Here's the corresponding fragment program:

```
# PROCESSING_COLOR_SHADER //required in Processing-flavored GLSL;
says we're making a shader instead of a texture mapper

//import our vertex shader's outputs explicitly
varying vec3 normal;
varying vec3 vertex_to_light;

void main() {
   const vec4 diffuse_color = vec4(1.0, 0.0, 0.0, 1.0); //red,
   opaque
   vec3 n_normal = normalize(normal);
   vec3 n_vertex_to_light = normalize(vertex_to_light);

   // Calculate red * (N*L) lighting equation
   float diffuse = clamp(dot(n_normalize, n_vertex_to_light), 0.0,
   1.0); //'clamp' restricts result to [0, 1] range
```

```
    gl_FragColor = diffuse * diffuseColor;
}
```

- Notice that the output here is just assigning to the predefined variable "gl_FragColor"

- Okay, keep working on your raytracers, and we'll chug along on Monday!

```
//*************************************************************************/
/
//******************* More GLSL Shaders - March 11th, 2019 ******************//
//*************************************************************************//
```

- *refuses to use lavatory before class, urea build-up intensifies*

- Okay, we were just starting to get into shaders last week, and today we're continuing our tour through this magical land!
  - Remember: the basic idea is that we get the vertices of the polygon as input, then put it through our vertex shader, then the rasterizer, and then our fragment shader

- In GLSL, there's a nice piece of syntactic sugar called "Swizzling," which lets us address the components of a vector using dot notation
  - Specifically, we can access it using the syntax ".xyzw," or ".rgba," or .stpq
  - As an example, let's say we have the following vectors:

```
vec4 v1 = vec4(4.0, -2.0, 5.0, 3.0);
```

- From there, we can use swizzling to access these parameters, either in groups or alone:

```
vec2 v2 = v1.yz;        // v2 = (-2.0, 5.0)
float scalar = v1.w;    // scalar = 3.0
```

- So, that's a nice convenience to know about

- One thing we should emphasize is that we can use vertex shaders to actually modify the 2D projection of the 3D vertices we get as input - let's try it!

- Let's suppose we have a bunch of triangles in a triforce-like shape passed as input, on a 1024x1024 screen w/ coordinates (-512, +512)
- We'll then write the following "twisting" vertex program:

```
const twistAmt = ?;

void main() {
   vec4 pos = gl_ModelViewProjectMatrix*gl_vertex;
   vec4 center = vec4(0, 0, 0, 0);
   vec2 diff = vec2(pos.x - center.x, pos.y - center.y);

   // rotate the point counter-clockwise, based on its distance
   from the center of the screen
   float angle = twistAmt*length(diff);
   float c = cos(angle);
   float s = sin(angle);
   gl_Position.x = c*pos.x - s*pos.y;
   gl_Position.y = s*pos.x + c*pos.y;

   gl_Position.z = pos.z;
   gl_Position.w = pos.w;

   gl_FrontColor = gl_BackColor = gl_Color;
}
```

- This'll turn our triangle into a shuriken-like, wavy shape - cool!
- Why don't we just draw the geometry like this in the first place, though? Why bother with this vertex shader as a middleman?
  - Well, for one thing, it lets us handle these vertex manipulations on the GPU, which is typically much faster than dealing with it on the CPU
  - For that reason, this is commonly done for parts of the geometry that need to be interactively modified for appearance's sake, e.g. letting the user draw waves on a surface

- Let's also see how we can use a vertex program and a fragment program together to draw a texture map on an object:

```
//=============================================
// Vertex Shader
#define PROCESSING_TEXTURE_SHADER
```

```
varying vec2 texture_coord;
void main() {
   gl_Position = gl_ModelViewProjectMatrix * gl_Vertex;
   texture_coord = vec2(gl_MultiTexCoord0); //hands first texture
   coordinate to the rasterizer; since it's varying, it'll be
   interpolated correctly across the surface
}

//=============================================
// Fragment Shader

// how does this variable get passed between the vertex/fragment
shaders? Occasionally, some "glue code" is needed to handle that, but
we can worry about that later
varying vec2 texture_coord;
uniform sampler2D my_texture;
void main() {
   gl_FragColor = texture2D(my_texture, texture_coord); //gets the
   colors of the texture at the given coordinates, handling
   bilinear/trilinear interpolation for us
}
```

  - So, the vertex shader sets up the texture coordinates, which are then interpolated across the polygon by the rasterizer, and the fragment shader then colors each pixel appropriately - cool!
- That's the simplest POSSIBLE texture program we could make, though; what if we wanted to do something a tad more sophisticated? Let's try to make a program that combines two different, shifted versions of a texture
  - Admittedly this isn't a particularly common use case, but roll with me here

```
//=============================================
// Vertex Shader

varying vec2 left_coord;
varying vec2 right_coord;
void main() {
   gl_Position = gl_ModelViewProjectMatrix * gl_Vertex;
   vec2 texture_coord = vec2(gl_MultiTexCoord0);
```

```
    left_coord = texture_coord + vec2(-0.2, 0.0);
    right_coord = texture_coord + vec2(0.2, 0.0);
}

//=============================================
// Fragment Shader
#define PROCESSING_TEXTURE_SHADER

varying vec2 left_coord;
varying vec2 right_coord;
uniform sampler2D my_texture;
void main() {
    vec4 left_color = texture2D(my_texture, left_coord);
    vec4 right_color = texture2D(my_texture, right_coord);
    gl_FragColor = 0.5*(left_color + right_color);
}
```

- So, that's generally how you combine vertex and fragment shaders together
- the vertex shader
  - In this particular example, we COULD have done this whole thing just using the fragment shader, but the main point here was to show how you can use these 2 shaders together

- Alright, we've seen we can program the vertex and fragment shaders, the vertex shader does its thing, the rasterizer does its interpolatory magic, and then the fragment shader handles the last coloring step
  - You'll be doing stuff VERY similar to this in project 4, right after spring break

- That brings us to an interesting point in the class: the break between
  - Up until now, we've been assuming we have this interesting geometry already there on the screen, and are then trying to render them
  - But where does this geometry actually come from? How DO we make a polygonal cone, or sphere, or spaceship? That's what we're going to start talking about now!

- First up, a bit of terminology (CHECK THIS):
  - A POLYHEDRON is a surface composed of polygons
    - What's a polygon? It's any shape made up of vertices and straight-line edges

- We'll often use many polygons in a polyhedron to approximate "smooth" surfaces
- A MANIFOLD is a part of the surface that looks like a (possibly bent) plane, i.e. a point could travel smoothly to all points in the manifold
    - Think of a pizza; a pizza without any slices missing would be a manifold, since an ant could start in the center and walk in a spiral without issue!
        - If the pizza had 2 slices missing, though, so that one of the slices only touched the rest of the pizza at its tip, it's not a manifold; our ant would "fall off" the pizza now if it tried to reach the "island" piece
            - Similarly, 2 cones that are only touching at their points wouldn't count as a manifold
            - Many algorithms will fail when dealing with non-manifold things
    - A "manifold edge" is one edge that's entirely inside the manifold, i.e. it's part of the plane (e.g. the diagonal of a square)
        - A BOUNDARY EDGE is an edge on the manifold (e.g. the side of a flat square); it's not as nice as a manifold edge, but most algorithms can still deal with them
        - A NON-MANIFOLD EDGE is one that's "sticking out" from the plane of the manifold, like 2 intersecting planes; these are nasty to deal with
    - To make this a little more concrete, let's think of the example of a regular, 6-sided, perfectly societally-adjusted cube:
        - All the edges are manifold edges
        - All the vertices are manifold - great!
            - If we removed one of the faces of the cube, though, then those 4 edges where the face of the polygon are will now be boundary edges!

- How do we create these polyhedra? How should we represent them? How do we make them as manifold as possible? We'll keep asking all of these questions on Wednesday!

//**********/
/
//********** Polyhedra Operations - March 13th, 2019 **********//
//**********//

- This week has a very special day coming up tomorrow - PI DAY!
  - (also Albert Einstein's birthday, but PIIIIIIII!!!)
  - Professor Turk's favorite person related to Pi is William Shanks, who in 1873 published 770 digits of Pi he'd calculated over 15 years - but (oh no!) only the first 573 were correct! He'd made an error!
    - He later corrected the error, but the PUBLISHER of his paper made a typo and printed the wrong number; sometimes, you just can't get a break
  - The memorization record for digits of Pi is (as of the last check) 67,890 digits - because, y'know, why not?
    - The highest number of digits computed for Pi is on the order of 12.1 trillion digits - it took about 86 terabytes of disk space, and 96 days of computation, but why should WE care?
      - Well, remember when we talked about partial shadows, and how we dealt with them by casting multiple rays from the light source? With circular light sources, we can do something called "rejection sampling," where we sample points on a square grid and just reject them if they're not inside the circle
        - This is surprisingly fast, actually; the number of rejections tends to be fairly low (about 1/4 of the samples), since the area of the square is 4*r^2 and the area of the circle is pi*r^2
          - If we sample enough points, we can actually use this to approximate Pi! If we sample a BUNCH of points, and count the number of points we reject, then we can re-arrange the equation to get:

            pi = 4 * (# samples in) / (total # samples)

          - This is a really simple example of Monte Carlo estimation

- Now, project 3B, implementing the rest of your raytracer, is going to be due when you all get back from spring break - let's talk about some pointers for it
  - There're 3 big pieces to the raytracer: a "ray intersection" part, a "color the ray" part, and "shade a hit" part
    - We'll first get the intersection, then return the hit and try to color it; that color function will then try to shade its particular hit, which'll occasionally check intersections for shadows/reflections and recursively need to get the "ray color" for reflected rays
  - Now, you also need to code up the intersection equation for a cone; the simple version looks like this:

$x^2 + z^2 = (ky)^2$

- Where the radius of the cone is "k*y"; by default, this equation'll give us a double cone
- We want to position this cone at some point in space, though, so we need to do this:

$(x - xb)^2 + (z - zb^2) = k((y - yb))^2$

- Where "x/y/z" are the ray part, where (for instance) x = (x0 + t*xDir)
  - You solve for a, b, and c, then, algebraically - even if it gets messy, I trust you've all got the algebra chops to handle that!
- To get the surface normal for the rounded part of this cone, it'll be tilted based on our value of k somehow, but, well...how?
  - To get this normal for a given hit point "P", we'll get the tangent vector T1 to the cone "B - P" (where B is the base point of the cone), and the "radius" R pointing straight out from the cone to the point:

    $R = (B + (0, P.y, 0)) - P$

    - If we rotate this "radius" vector 90 degrees, then we've got 2 orthonormal vectors! So, the actual "T2" vector will be:

      T2 = (-dx, 0,) (missed this, but just rotating "R" 90 degrees so it's pointing straight down)
  - From there, we can finally use those 2 basis vectors to get the normal (try flipping the order if it doesn't work):

    $N = T1 \times T2 / |T1 \times T2|$

- "Again, PLEASE start this homework early on in case you get stuck!"

- Alright, we were talking a little about polygons yesterday; now, let's talk about some operations we can do on these polygons, and 3 operations in particular:
  - Laplacian Smoothing

- Face Subdivision
- Triangulation

- LAPLACIAN SMOOTHING is where we can "smooth" a given polygonal surface by taking a vertex in the middle of other vertices and moving it to the center; for instance, if we have a vertex 'V' that's surrounded by 5 other vertices, then we'll move it instead to the average position of those vertices:

V' = 1/5 * (v1 + v2 + v3 + v4 + v5)

- We apply that equation to ALL the vertices, and that really will smooth all the points on the surface! It means we can take a rough surface (especially from scan data) and make it more regular
  - The one downside is that this'll shrink the polygon, but there are ways around that by alternately "inflating"/"deflating" the object
  - Usually, this is coded so that when we apply this equation to all the vertices we store the "current vertex" positions, and use that to calculate all the new positions. This way, the result is independent of the order we smooth the vertices in

- FACE SUBDIVISION is where we have a given polygon and want to chop it up into a bunch of different faces (for various reasons)
  - For instance, if we have regular triangle, we might change it into a triforce-like version made up of 4 smaller polygons
  - This is useful for turning "rough" surfaces into smooth ones (think of the geodesic dome at Epcot) - or, alternatively, as a way of getting more polygons for us to play with so we can add texture to a surface
    - We'll revisit this in a few weeks, when we'll start talking about subdivided surfaces and specific algorithms like Catmull-Clark, etc.

- Finally, TRIANGULATION lets us turns a given polygon into a version made up entirely of triangles
  - Why would we do that? Well, triangles are NICE to play with! By nature their vertices will always lie on the same plane, it means we only need to have one constant-size data structure for triangles instead of a general "n-vertex" structure, etc.
  - Generally, we do this by adding diagonals to the polygon

- In order to use ANY of these algorithms, though (triangulation possibly excepted), we need to know how our polygons are connected to one another - and

to do that, we need to store connectivity information!

- How do we store this connectivity data, then? There're 4 general approaches we can take:
  - POLYGON SOUP is when we DON't store any connectivity information at all; we just "throw them all in the pot," and our faces don't store anything about what they're adjacent to:

```
class Face {
   vertices[3];
}
```

    - In this scheme, polyhedra are just lists of faces making up the polyhedra - that's all!
      - Because this stores a minimal amount of information, it's sometimes used as a file format, but it's pretty limited for production use
  - SHARED VERTICES (or "Indexed Face Sets") are where we store the list of vertices in a face AND the other faces that reference those same vertices
    - This is also pretty common as a file format, since it's a good mix of flexibility AND (due to how we reuse vertex IDs) actually can save us memory for large objects
    - To actually use this, we'll go through and assign a number/ID to each vertex; then, for each face, we'll hold a list of the vertices it uses:

```
0 = (-1.0, -1.0, -1.0)
1 = (1.0, 1.0, -1.0)
2 = (1.0, -1.0, 1.0)
3 = (-1.0, 1.0, 1.0)

f1 = (0, 1, 2)
f2 = (0, 2, 3)
f3 = (0, 1, 3)
f4 = (1, 2, 3)
```

    - So, this scheme reuses the IDs for each vertex, but in Polygon Soup each face has to have its own ID for each vertex - meaning this'll actually use less memory when there are lots of vertices at play!

- Alright, 2 down, 2 to go - we'll cover the other 2 storage schemes on Friday!

//**********/
/
//********** Corner Polyhedra Representation - March 15th, 2019 **********//
//**********//

- Spring Break is nearly upon us, and the attendance in the room is reflective of the fact
  - I count 43/150 people in class today - which is actually pretty good!
  - BUT WE'VE LOST LUKE! WAGH!
    - Nevermind, Luke get!

- Now, for your homework, let's go back to finding the intersection points
  - To find the intersection with the flat, HORIZONTAL plane (which we'll assume for THIS homework), you'll plug the ray into the equation:

    r0.y + dir.y*t = plane.y + coneHeight

    - Solve for T, find the intersection point, and make sure the point is within the cone's radius - done!
      - The surface normal is REALLY easy, then - it'll just be pointing straight up!

- Now, we were going through different polyhedra representations on Wednesday, and got through 2 of them: the "Polygon Soup" and "Shared Vertex" schemes. Let's keep going!

- Yet another method is the CORNERS method
  - This method ONLY works for objects entirely composed of triangles, but it works really well for them
    - It was actually invented by another professor here at Georgia Tech: Jarek Rossingnac!
  - The basic idea is that we say each triangle's vertex has a "corner," with 3 corners per triangle - but it's NOT the same thing as a vertex
    - Similar to the Shared Vertex table, we'll create a GEOMETRY TABLE with the (x,y,z) coordinates of each vertex:

[(x1, y1, z1),
(x2, y2, z2),
(...)]

- We'll then have a VERTEX TABLE with a list of vertex indices, like this:

[0, 1, 2, 1, 0, 3, ...]

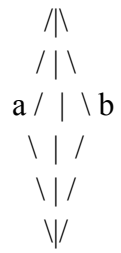
- Each group of 3 vertices in this table defines a triangle; so, the 1st triangle is made up of corners (0,1,2), the 2nd triangle is (1,0,3), etc.
  - So, for a given corner index "corner" in this list, it'll belong to triangle # (floor(corner / 3))
    - From this, we can easily get from a particular corner to a particular triangle!
- IMPORTANTLY, the order of these vertices in each triangle should respect orientation (e.g. always being counter-clockwise)
  - As you're going to see, this allows us to find adjacent triangles
  - How do we make sure all these triangles are oriented the same way? That's actually a little bit complicated, but it can be done - let's ignore that question for now
- Using this table, we can say the "next corner" in a triangle will be:

corner.next = 3*corner.triangleNum + ((c + 1) % 3)

- "Greg, what the heck is this doing?" Well, it's saying that to find the next vertex of a given corner, we'll jump to the start of that corner's triangle, then (using the modulus) rotate to that corner in the triangle, then one more
  - Similarly, to find the previous corner, we'll just find the "next-next" corner (literally corner.next.next)

- Finally, we'll have a 3rd table: the ADJACENCY INFORMATION, or the "Opposite Table," telling us which triangles are next to one another
  - To do this, we just have to add one more piece of information to each corner: "corner.opposite," which gets us the ID of the opposite corner
    - What's the opposite corner? Well, if we have 2 triangles that

share a base, then the "opposite" corners A/B are the 2 that don't share a vertex:



- If we have the Vertex Table, this actually isn't too hard to compute! Here's how we do it:

```
for each corner a:
   for each corner b:
      // NOTE: "corner.vertex" means vertexTable[corner]
      if (a.next.vertex == b.prev.vertex AND
      a.prev.vertex == b.next.vertex):
         opposite[a] = b
         opposite[b] = a
```

  - So, if the corners share the same adjacent vertices, it means they must each be the "3rd point" in their triangle, and that their triangles are adjacent, so they're opposite!
- But why do we care about knowing these opposite corners? What do we "get" from this? Well, a LOT!
  - We can calculate the corner's left/right neighbors really easily:

```
corner.rightNeighbor = corner.next.opposite
corner.leftNeighbor = corner.prev.opposite
```

    - This gets us another corner,yes - but it's a corner that's INSIDE the right/left adjacent triangle, and we can get that triangle's ID really easily!
  - We can also calculate the SWING of the triangle, which lets us rotate to all the triangles around a given vertex:

```
corner.swing = corner.next.opposite.next
```

- This means we can process all the triangles around a given vertex super easily, which is difficult in Shared Vertex schemes and straight-up IMPOSSIBLE in Polygon Soup!

- It takes a little while for this representation to sink in (remember that corners are NOT the same thing as vertices - vertices can be shared between triangles, corners can't), so why are we spending so much time on its gory details?
  - Well, because it is a legitimately elegant way of storing triangle meshes: it's compact, but it gives you a LOT of information for not a lot of storage
    - ...there's also a decent chance this'll reappear on an exam sometime in the near future, but surely you don't care about THAT...

- Alright, and with that: break time! See you all in 9 or 10 days!

//**********/
/
//********** Bezier Curves - March 25th, 2019 **********//
//**********//

- Alright, today we're going to be shifting gears into a different world of graphics: curves!
  - The book chapter on this section is quite good, so make sure to take a peek at it if you're interested

- First off, let's start with a definition: a CONVEX HULL for a set of points is the smallest convex shape that contains all those points
  - This is a simple concept, but it's important!
  - A CONVEX COMBINATION is a point that can be expressed as a "convex" combination of other points in a set
    - For a line segment p1/p2, for instance, a point "a" is a convex combination of points p1/p2 if it can be written as:

      a = w1*p1 + w2*p2

      - Where w1 + w2 = 1 and the weights are non-negative, meaning a is on the line segment
    - For a triangle defined by points p1/p2/p3, we say a point "b" is a convex combination of those points if it can be expressed as:

b = w1*p1 + w2*p2 + w3*p3

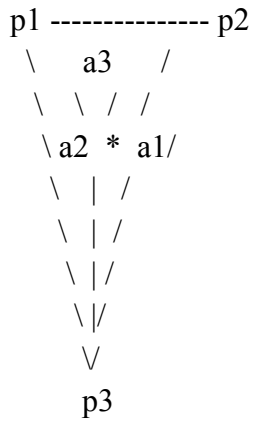
- Where w1 + w2 + w3 = 1 and all the weights are non-negative, meaning b is inside the triangle
- "Think of a convex combination like this: each of the points has a rope attached to our point, and they're all pulling their ropes tight. If we can get our point by pulling on the ropes like this, then it's a convex combo!"

- Generalizing this combination definition, we can say that for a set of "n" points p1...pn, and a given point "c" written as:

c = w1*p1 + ... + wn*pn

- ...we can say that c is INSIDE the convex hull if all the weights are non-negative and sum up to 1!
- Hold on to your eyeballs, because that's an important fact!

- Building a bit weirdly off of this, let's talk about some mysterious thing called "barycentric coordinates"
- For a triangle ONLY, the convex weights w1/w2/w3 for a given point "g" are called the BARYCENTRIC COORDINATES of that point
- If we know where our point is, then we can figure out these weights based on the area of the different triangle and it's interior sub-sections:



- Where the total area A = a1 + a2 + a3, and the individual weights'll be:
- W1 = a1/A

- W2 = a2/A
- W3 = a3/A

- These coordinates have a lot of nice properties we won't get into right now, but trust me, they're there!

- Now, let's start using these points to talk about a really common, useful type of curve, called BEZIER CURVES
  - These are a whole family of curves named after the mathematician Pierre Bezier, but we'll mostly be talking about CUBIC, or "degree 3," Bezier curves
  - For this curve, we'll have a start point, an end point, and 2 intermediate "control points" that define how the curve is shaped as it goes from the start to the endpoint
    - What're a few properties of this curve?
      - Bezier curves INTERPOLATE through p1/p4 (the start/end point)
      - The curve is tangent at its endpoints to the line segments P1/P2 and P3/P4
      - ALL points on the curve are inside the convex hull of p1, p2, p3, and p4
      - Cubic bezier curves give independent control of the endpoints and tangents (i.e. we can change where the endpoint is without affecting the tangent, and vice-versa)
        - We can't do that with a line segment, or even with quadratic Bezier curves - it's the lowest-degree curve that gives us this control! This is a big reason cubic Bezier curves specifically are popular among CG people
    - If we want to, we can then chain multiple of these curves together into multiple, longer curves
  - "Now, equations sometimes strike fear into people's hearts, but we've got to look at them - so here we go"
    - Recall that the parametric line segment equation was where we had 2 points P1/P2, and a point on that line q(t) was defined:

$$
\begin{aligned}
q(t) &= (x(t), y(t)) \Rightarrow \\
&= (x1*(1-t) + x2*t, \\
&\quad\ y1*(1-t) + y2*t) \\
&= P1*L1(t) + P2*L2(t)
\end{aligned}
$$

    - Where L1(t) = 1-t, and L2(t) = t (our "basis functions")

- Notice that L1(t) + L2(t) will add up to 1 for ANY value of t
- and remember how convex combination weights should add up to 1 as well? We can consider L1 and L2 to be the weights of this equations!
    - We can use this general idea of control points, multiplied by weights, to define our Bezier curve itself
    - ...and remember: technically, this IS a degree-1 Bezier curve!
- For cubic Bezier curves, then, what can we do?
    - Well, we've got four total control points (including the endpoints), and we will have four basis functions/"weights" B1 to B4
    - Therefore, a point on our curve q(t) will look like:

$$q(t) = B1(t)*P1 + B2(t)*P2 + B3(t)*P3 + B4(t)*P4$$

- What ARE these basis functions, though? We need them to add up to 1 and be non-negative for values of t between 0 and 1, so in general:
    - B1 will start at 1, then sharply drops off to 0
    - B2/B3 will be "humps" in the middle values of t, indicating the curve will be tugged towards them
    - B4 will be a flipped version of B1, starting off at 0 and curving exponentially to 1

- So, the big ideas for today are the idea of convex combinations and their non-negative weights, and the idea of using basis functions that meet this requirement to define curves. We'll go over what EXACTLY these equations look like for a Bezier curve...next time!

//**********/
/
//********** Bezier Curves (cont.) - March 27th, 2019 **********//
//**********//

- Professor Turk looks like he wants to tell us something...
    - The ACM Turing Award is basically the Nobel Prize of Computer Science, and the 2019 award has been given to 3 researchers who pioneered neural nets - so, give them a hand! And/or blame them for the flood of MASHIN LRNIN NAO!
- On a sadder note, some students were upset that the TAs couldn't give them an

extension on the 3rd project, and complained about it to them
    - "The TAs didn't do anything wrong, guys. I'm the one who has to make that decision. If you're going to get mad, please only get mad at me."

- Alright, on Monday we started talking about Bezier curves, and SPECIFICALLY the ever-popular cubic Bezier curve
    - We also talked about convex combinations, how we could define points inside of a convex hull by assigning weights - and this is actually how Bezier curves are made!

- We figured out on Monday that the weights for a cubic Bezier curve are given by 4 basis functions - but how did Bezier figure out what these functions were? Did they come to him in a dream?
    - Luckily for us, there's actually some intuition we can use to figure out what these basis functions are!
        - Imagine that we draw line segments connecting the 4 control points, like this: P1P2, P2P3, and P3P4
            - If we take the midpoints of each of those line segments, we'll end up with 3 midpoints: M12, M23, and M34 (or "L2, H, R3")
            - If we then connect those midpoints with line segments, we'll end up with 2 line segments: M12M23, and M23M34...
            - ...and if we repeat this and connecting the midpoints of THOSE line segments (L3 and R2, respectively) with a single line segment, and take the midpoint of that final line segment (which we'll call L4 or R1), we'll end with a single point
            - As it turns out, that single point is EXACTLY at Q(1/2) - the halfway point along the curve!
    - How does that help us? Well, once we know that point, we can draw the Bezier curve by dividing the curve in half into TWO curves, with control points:

        (P1, L2, L3, L4)
        (R1, R2, M34, P4)

        - We can then keep doing this division until we've got enough points along the curve to draw a good-enough approximation using pixel!
            - Is this the best way to draw a Bezier curve? Probably not - but this SUBDIVISION scheme is a good start

- There's a better way of doing this Bezier subdivision, though - so let's look at the GENERAL SUBDIVISION way of drawing a Bezier curve
  - To find the point 3/4 of the way along the curve, for instance, we won't look for the midpoint of each line segment continuously - INSTEAD, we'll take the point 3/4 of the way along each line segment!
    - Then, EXACTLY like we did before, we'll connect those points with 2 line segments, find the 3/4 point along each of them, connect it with a line segment, and then the 3/4 point along THAT last segment will be Q(3/4) for the whole curve!
  - How do we find the point along each line segment? Well, for the P1P2 line segment, the 3/4 point would be:

    pt = P1*(1/4) + P2*(3/4)

    - OR, in general, for a point "t" percentage along a line segment:

      pt = P1*(1-t) + P2*t

  - Once we have this point, we can use it to calculate the points for the next line segment, letting us calculate down to the actual point on the curve we care about!

- With that figured out, we're finally ready to see what those Bezier basis functions look like - here we go!
  - For the first control point,

    B1(t) = (1-t)^3

  - For the 4th/last control point, it would be:

    B4(t) = t^3

    - ...look familiar? This is the same weighting we had for the straight line segments! This comes directly from our general subdivision technique!
  - For the 2nd and 3rd points, the functions look a little different, since we're "mixing" the two endpoints together:

    B2(t) = 3t*(1-t)^2
    B3(t) = 3t^2 * (1-t)

- So, the B1/B4 and B2/B3 basis functions are symmetric about t = 1/2 - they're inverses of each other!
    - Now, I won't ask you to derive the basis functions for a quadratic or quartic Bezier curve, or anything like that - but if you understand what we're doing with the subdivision technique, you should be able to!
        - If you want to see a really cool illustration of this, check out "Jason Davies Animated Bezier Curves" on Google

- Before we carry on, why do we care about Bezier curves at all? Because in many cases, pixels just aren't good enough - we NEED curves to have shapes that look sharp at any resolution, which is important for things like font rendering, architectural drawing, etc.
    - For cubic Beziers in particular, that "independent tangent" property is important

- Now, take a look at the following sequence: "3, 4, 9, 18, 31." Do you see a pattern here? If not, don't worry - we're about to talk about POLYNOMIAL EVALUATION!
    - For instance, take a look at the function here:

        f(t) = at^3 + bt^2 + c*t + d

        - How many operations does it take to evaluate this? Well, b*t^2 takes 2 multiplies to compute, and a*t^3 takes 2 more (since it's t^2 * t), so it'll take 5 multiplies and 3 additions
    - We can do better than this using HORNER'S RULE! For a cubic polynomial, we can cut down the number of operations by grouping things cleverly:

        f(t) = ((at + b)*t + c)*t + d

        - Now, we only have 3 multiplications and 3 additions - that's an improvement!
    - Can we do any better, though? As it turns out, we can!
        - Consider a scale between 0 and 1, split up into equally-sized pieces each of length "h"
        - Let's suppose, then, we've got a linear function "f(t) = at + b"
            - In that case,

                f(0) = a*0 + b = b

f(h) = a*h + b = f(0) + ah

- As it turns out, then, THIS is also true:

f(2h) = f(h) + ah
f(3h) = f(2h) + ah
(...)

- We can just keep adding ah! Instead of having to recompute
- Does this hold for higher-order functions, though, like quadratics? It turns out that it does!
  - If we have a function f(t) = at^2 + bt + c, then:

f(t+h) = a(t+h)^2 + b(t+h) + c
= at^2 + 2aht + ah^2 + bt + bh + c
= f(t) + 2*aht + ah^2 + bh

- Notice that this part we're adding to the function is now a LINEAR equation!
  - (obviously simpler, but missed if there was more significance to this?)
- This is known as the FINITE DIFFERENCE technique!

- So, what's the next number in this sequence? 48 - but how did I get that?
  - Look at the space BETWEEN each of these numbers: 1, 5, 9, 13, 17...
    - So, the SPACING is increasing by a linear amount, which makes the actual sequence (3, 4, 9, 18, ...) look like it's increasing quadratically! Neat!

//**/
/
//********************** 3D Surfaces - March 29th, 2019 **********************//
//**//

- Alright, project 4 has been posted and you should start it as soon as you can!
  - As you know, there'll be 5 total projects in this course, each worth 14% of your total grade
  - This project doesn't have two different parts, but it's still a relatively challenging assignment (although it should be easier than the raytracer) - so make sure you're giving yourself time!
    - Specifically, this project is all about writing GLSL shaders, and

you're going to be writing shaders to accomplish 4 different kinds of effects

---

- Now, for your homework, let's briefly leave this rational plane for a world of imagination (but not "pure" imagination) - complex numbers!
  - As I'm sure you remember, complex numbers are numbers that have both a real and an imaginary part, like this:

    $Z = a + b*i$

    - We can graph complex numbers on a 2D ARGAND DIAGRAM, where we plot the real part (a) on the X-axis and the imaginary part's coefficient (b) on the Y-axis
  - To multiply complex numbers, we have to multiply both terms of it, per the ancient distributive rule you might recall from algebra; for instance,

    $Z^2 = (a + bi)^2 = a^2 - b^2 + 2abi$

    - So, "a^2 - b^2" is the real part, and "2abi" is the non-real part
  - If we graph "z^2," we'll see that complex numbers with a "length" less than one will go towards the origin, but numbers with a length greater than one will DIVERGE and keep growing farther away
    - If we continue squaring the number iteratively, this trend will continue
    - (if the length = 1, of course, it'll just stay put!)

- So, that's what happens if we iterate a function f(z) = z^2, but what about for different functions?
  - Take a function where we just add a constant complex number "C" to z^2, for instance:

    $f(z) = z^2 + C$

    - If we iterate this function, will it stay near the origin, or diverge? It's actually tough to say ahead of time!
  - To draw shapes, people will often iterate functions like this for awhile, and then shade it one color if it diverges and another color if it stays near the origin
    - In particular, this function above will get us a JULIA SET, a type of

fractal!
- "z_0", our initial z-value input for the function, will come from the screen position/texture coordinates, where the complex coefficients (a, b) will be based on our pixel coordinates (x,y)
- For Julia sets in particular, you'll usually want to keep the values for both "a" and "b" within the [-1.5, 1.5] range
- Different values of C will get us different Julia sets, but for a given fractal C will NOT change as we iterate, or change pixels, or anything - it just stays the same

- Okay, so that's all the fractal knowledge you need to finish your homework - now let's get back to Bezier curves!
- How do we actually draw these things in the first place? There are a few ways we can do this:
- 1) We COULD directly solve the cubic basis functions (using Horner's rule to speed it up)
- 2) If we know the values of H we're using, we can use the finite difference method to draw this even faster
- This is usually the most common method, but the other two have their place in certain situations
- 3) We could do a recursive version of curve subdivision - keep subdividing until we get to an acceptable smallness, and then just draw the pixel at the midpoint/ three-quarters point/etc.

- ...so, that's 2D Bezier curves, but where do we go from here? To 3D curves, and 3D curved surfaces!
- That's quite a jump to make, though, so before we dive straight into 3D curves, let's instead talk about PARAMETRIC SURFACES first
- Remember parametric lines? For a given t-value in the 1D range [0,1], we'd be able to map that value to a distance along a line segment between 2 points
- What, though, if that "parameter space" for t wasn't 1D, but 2D? What if we had TWO values (t, s), each with a [0, 1] range, that formed a box of possible parameter values?
- Well, if we have 3 points P1, P2, and P3, then we have enough information to define a 3D plane/parallelogram - and we can use (s, t) to define any point on that plane!
- How does that work? Well, assuming P1 is in-between the two other points, let's say the parallelogram is defined by 2 vectors:
- The "T vector" from P1 to P3 = P2 - P1

- The "S Vector" from P1 to P2 = P3 - P1
- We'll then say that P1 is the corner of our parallelogram, and that we'll have some delta values for both S and T in all 3 directions:

dx_s = p2.x - p1.x dx_t = p3.x - p1.x
dy_s = p2.y - p1.y dy_t = p3.y - p1.y
dz_s = p2.z - p1.z dz_t = p3.z - p1.z

- Using that information, we can then define any point "Q" on that parallelogram as:

Q(s, t) = P1 +

- How can we make 3D surfaces out of this, though? Well, let's see...
  - Let's start with a simple surface: a cylinder!
    - If we think about it, a cylinder is just a rolled-up plane; so, how can we "roll up" our S-T plane to get a cylinder?
    - Let's say that any point on the cylinder is defined as:

Q(s, t) = [x(s,t), y(s,t), z(s,t)]

- Where:

x(s,t) = cos(s)
y(s,t) = sin(s)
z(s,t) = t

- So, S defines our position AROUND the circular end of the cylinder, and T defines our height!
  - So, if we can draw a curved surface like that, why don't we try drawing a "doubly-curved surface" - a SPHERE!
    - Here, we'll say our ranges for T and S are slightly shifted, so that:
      - T is in the range [-pi/2, pi/2]
      - S is in the range [0, 2*pi]
    - If we think about this like a globe, T will be our latitude (the "height"), and S will be our longitude around the sphere
      - With that, then, we can define our x, y, and z positions as:

x(s,t) = r(t) * cos(s) = cos(t) * cos(s)

y(s,t) = r(t) * sin(s) = cos(t) * sin(s)
z(s,t) = sin(t)

- Where "r" here is the radius of the sphere - which, if we think of any slice of the sphere as a circle, will just be cos(T)! (not sure I see this?)

- Alright, we've just dipped our toes into the world of 3D surfaces, so come Monday to keep diving in (and experience the slimy joys of increasingly strained metaphors from me)!

//**********/
/
//********** Bezier Patches / Subdivision - April 1st, 2019 **********//
//**********//

- I'm feeling very foolish today, but not in the jovial sense
---

- Alright, on Friday we started talking about 3D curves - and today, we'll be extending Bezier Curves into 3D BEZIER PATCHES
  - Instead of being in 2D, of course, this is 3D - and instead of having just 4 control points, there are 16 control points - WHOA!
    - That's a lot, but basically, the surface is defined by 4 different Bezier curves, each of which requires 4 control points
  - Using these patches, though, we can make just about any smooth 3D surface by hooking them together (cue the apparently-famous Utah teapot!)

- Let's start looking at CUBIC BEZIER PATCHES in more depth, now
  - As we said, 4 Bezier curves define a Bezier patch, with a total of 16 3D control points
    - We can define a regular Bezier curve in 2D or 3D, but with Bezier patches, it really only makes sense to define them using 3D points
  - There are 2 parameters this time: S and T!
    - Each of these parameters are in the range [0, 1]
    - We'll label all our control points from P00 ... Psdir,tDir ... P33
    - We'll then say that all the points with the same "S-index" define a single Bezier curve (e.g. P00 to P03 defines a single curve P0(t)), giving us the points for 4 different Bezier curves!
      - Now, you might think that our 3D surface would smoothly go

through each of these 4 curves, right? But that's NOT usually the case - oftentimes the surface won't even touch the 2 middle curves!
- As it turns out, if we define our Bezier curves in the "S-direction" instead, it'll still create the exact same surface

- So, with these 4 Bezier curves, how do we create our Bezier surface?
- Well, for a given T value, we'll find the 3D point on each Bezier curve for that value:

p0 = P0(t), p1 = P1(t)

- We'll then use THOSE 4 points to define a new Bezier curve, which gives us the points on the surface!

- So, lay this out EXPLICITLY, here's how we find a given point Q(s, t) on our Bezier patch:

1) Create 4 Bezier curves using our points: G1(t), G2(t), G3(t), G4(t)
2) Calculate 4 points on each of these curves based on the T parameter (P1)
3) Create a NEW Bezier curve using those 4 points
- ALL the points on this new Bezier curve will be on the final Bezier patch
4) Calculate point Q(s,t) on that new curve, using S as our parameter

- So, which points does this surface actually pass through?
- Well, it's only guaranteed to go through the 4 corners of our patch - but then how can we "stitch" 2 different Bezier patches together so their edges meet up?
- To do this, we need the 4 control points defining the "edge" of the Bezier patch to be the same for both patches
- If we ALSO want the transition between the 2 surfaces to be smooth, we need to make sure the tangents of the 2 patches line up, so the adjacent Bezier curve to that edge on each patch needs to ALSO match up

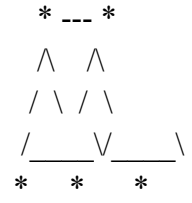
- "There'll probably be 1 or 2 questions on these patches on the final, so make sure you've got the gist of these things"

- Alright - Bezier patches take a LOT of work to define, with 16 control points and whatnot, and they also mean our surface has to be divided up into these roughly-quadrilateral smooth areas
- Those are definitely some restrictions, and Bezier patches end up not being used much outside of CAD modeling

- Instead, modern animations and games tend to use a different kind of curve instead, known as "subdivision surfaces"

- Why do animators prefer these types of curves? What makes SUBDIVISON SURFACES any better?
  - Well, Bezier patches have the following problems:
    - You have to somehow chop up surfaces into quadrilateral patches, even if they're not very rectangular
    - Getting a smooth, continuous surface is messy - you basically need 12 out of 16 control points to match up EXACTLY with their neighbors
  - In contrast, subdivision surface schemes can turn ANY polygonal mesh into a smooth surface (depending on your algorithm), continuity comes for free from the algorithm, and it's generally pretty easy to program!
    - This technique first appeared in 1978, and was independently discovered by the research pairs Catmull-Clark and Dao-Smith
      - Dao/Smith's version is
      - Clark would later found Netscape (which would become Mozilla), and Catmull is currently serving as the President of Pixar
    - Even though this is an older technique, though, it took until the 1990s for mathematicians to prove the algorithm had these nice properties, which led to them catching on with animators

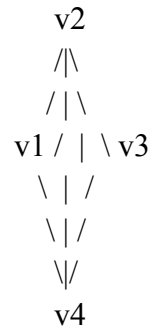
- There're a few different variations of subdivided surfaces; let's look at them in turn
  - The first is the LOOP SCHEME (invented by Charles Loop), which only works with triangles but has "C2 continuity" (i.e. you can take 2 derivatives of the surface and still have it be continuous everywhere)
    - To do this, suppose we have an original surface with 3 triangles and 5 vertices, like this:



    - We'll then add a vertex roughly to the midpoint of every line segment (though NOT quite), turning each triangle into 4 smaller Triforce-like triangles
    - From there, here's what we'll do to actually subdivide:

1) Compute the locations of the new vertices on the edges
   - To compute this point, for a line segment in the middle of 2 adjacent triangles, we'll say the position is:

$$V = 3/8(v2 + v4) + 1/8(v1 + v3)$$



   - Notice that we have 2 weights 3/8 and 2 weights 1/8, adding up to 1.0 - this is a convex combination, so the vertex'll stay in our surface's convex hull!
2) Move the positions of the old vertices
   - Basically, we're trying to smooth the surface by smooshing these new vertices together
   - To do this, we'll move them based on the VALENCE of a point (i.e. the # of triangles around a given vertex)
      - Most commonly, there will be 6 triangles around the vertex (i.e. a valence of 6), meaning the point is totally surrounded by triangles
   - For each point, if the point has a valence of "k," the new position will be:

$$V' = V*(1 - k*b) + b*(v1 + v2 + ... + vk)$$

      - Where "V" is the position of the original vertex, "k" is the number of triangles around the point, v_1 ... v_k are the positions of the vertices around our point, and beta is the weight
         - There are several values for b, but the original value was b = 3/(8*k) for k > 3 and b = 3/16 if k=3
   - If the point has k =/= 6, we call them EXTRAORDINARY POINTS, and they're annoying to deal with; they'll only have C1

continuity
3) Make smaller triangles again (i.e. the midpoint-1-to-4 thing)

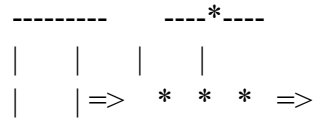
- We'll keep doing this until the surface is smooth enough
  - Luckily, we usually don't need to call this many times to end up with a pretty smooth surface, since the # of triangles increases by a factor of 4 each step!
- So, remember that Loop subdivision requires a triangular mesh, it has a polygon growth factor of 4, etc.

- Alright, we'll get to Catmull-Clark subdivision next time!

//**********/
/
//********** Catmull-Clark Subdivision- April 3rd, 2019 **********//
//**********//

- We're starting off today's lecture with a video - "Geri's Game," from Pixar wayyyy back in 1997
  - Besides being enjoyable, this is the first film Pixar ever did with sudvidision surfaces (and if you look at the "Executive Producers" in the credits, Edwin Catmull is right there!)
    - If you look at the original mesh for Geri, his face is mostly made up of quadrilaterals, but with a few triangles here and there
  - How do you end up with creases in "smooth" subdivided surfaces? Tony DeRose figured out a way to do it in this movie with Geri's wrinkles, so there must be a way!

- If Pixar uses Catmull-Clark subdivision, then it should be good enough for us
- so let's look at it!
  - CATMULL-CLARK subdivision is C2 ALMOST everywhere, and is for surfaces made up of mostly quadrilaterals
    - To subdivide this, for all the polygons in the mesh, we'll first add a new vertex at the middle of the whole polygon AND one to each edge of the polygon, and then connect the edge vertices to the center vertex



- Notice that this will turn triangles into quadrilaterals, giving us an all-quad surface after the first subdivision
- More mathematically, if we have a quadrilateral with original vertices V1 to V4, then:
  - The new center vertex, V, will have position:

$$V = 1/4 * (V1 + V2 + V3 + V4)$$

    - For a polygon with "K" vertices, the center vertex in general will be:

$$V = 1/k * (V1 + ... + Vk)$$

    - Once again, notice that this is a convex combination!
  - For the vertices on an edge, this is a little bit more tricky
    - In the typical case of two quads that share an edge, the new point on the line the polygons share will be:

$$v = 1/16(v1 + v2 + v3 + v4) + 3/8(v5 + v6)$$

      - Where the 1/16 vertices are the ones that don't form the edge in question, and the other are the two "line segment" vertices
    - MORE generally, for 2 polygons of any number of sides the weight will be an even 1/4 for the 2 vertices forming the line segment and 1/4 for the 2 polygon's central vertices
- After adding these new vertices, though, we need to move the old vertices to actually smooth the surface - how do we do that?
  - Well, the math is a little bit messy, but let's say that "ei" are the new "edge vertices" on the mesh and "fi" are the new center vertices
    - In that case, we'll say the constant points E and F are:

$$E = 1/k (e1 + ... + ek)$$
$$F = 1/k (f1 + ... + fk)$$

- THEN, for a given original vertex "v", its new position will be:

$$v' = E/k + F/k + v*(k-2)/k$$

- With that calculated, how smooth will this Catmull-Clark surface be?
  - As we said, MOST points will have C2 continuity, and will have a valence of 4 (i.e. they'll be quads)
    - ...where will it be C1, then? Only at the "extraordinary vertices" of the surface

- Okay, that's Catmull-Clark subdivision in general, but most surfaces aren't perfectly smooth - they have creases and sharp edges on them (e.g. the edge of an otherwise-smooth desk)! How can we replicate that with subdivision?
  - To do this, we can "tag"/label certain edges of the polygon to be SHARP
    - If a new vertex is being created, we'll ONLY form a new point on that edge using this formula:

$$V = 1/2(v1 + v2)$$

    - Where v1/v2 are the vertices making up the sharp edge's line segment
      - This way, the new points will only be on the line, and won't change that outline!
  - If an old vertex is between 2 sharp edges, then, we'll change the formula for its new position to be:

$$v' = (6v + v1 + v2)/8$$

    - Where v1/v2 are the adjacent vertices to "v" along the sharp edge
      - This means that "v" is only allowed to slide along the existing sharp edge - again, not disturbing the outline!
  - This doesn't *quite* cover all the cases for sharp edges (what if there are 3 sharp edges leading to a vertex? What about vertices at the end of a sharp edge?), but it should give you the general idea

- Sometimes, we might also want SEMI-SHARP EDGES: cases where we don't want a knife-sharp edge, but we DO want it to hold its shape in some way
  - For these, basically, we want to have a sharpness "parameter" for a given edge, where "S=0" doesn't do any sharpening, "S=1" is a little bit sharp,

"S=2" is more sharp, and so one, up until "S=infinity" (perfectly sharp)
- The rule for this is actually surprisingly simple: we'll use the "sharp" subdivision rules on a vertex "S" times, and then apply our smoothing rules once
- This adds a bunch of polygons without changing the edge's shape, then tries to smooth those now-smaller polygons (which'll result in a more constrained smoothing)

- Now, we'll be covering platonic solids on Friday, but before we get to that we NEED to talk about regular polygons
- A REGULAR POLYGON is one that is convex, has all sides equal-length, and where all angles within the polygon are the same
- Common examples of these are equilateral triangles, squares, regular pentagons/hexagons/septagons/octagons/etc.
- "Poor Heptagon - we can't agree on a name for 7-sided shapes"
- Years ago, Professor Turk thought that Heptagons just didn't exist out "in the wild," and then one day in CVS he found a pill container with one side for every day of the week - "so I had to buy it!"
- What about a pentagram? It has equal angles and equal sides, but it's NOT convex!

- As it turns out, you can tile your floor with equilateral triangles, squares, and hexagons - but what about the rest? We'll get to that on Friday - adios!

//**********/
/
//********** Platonic Solids - April 5th, 2019 **********//
//**********//

- Chicago: land of the sea-like pizzas (and Carl Sandburg)

- So, we were talking about regular polygons yesterday: polygons that have equal sides, equal angles, AND are convex (*squints suspiciously at pentagram*)
- "Pentagrams are great for summoning demons in games, but not much else"

- As it turns out, there's a nifty mathematical problem called REGULAR PLANER TILING
- The problem is this: suppose we want to fill an infinite plane using just

one type of regular polygon, all the same size, without any gaps or overlaps
    - (I suppose mathematicians just have a cabal of infinite warehouses somewhere, in desperate need of interior decorating)
- A SQUARE TILING is easy - it's just a bunch of squares all put together!
    - Here, each of the tiles has 4 sides (of course) and each vertex has a valence of 4
- We can also do a DUAL TILING, where we take the regular tiling, swap the vertices and faces (i.e. put a vertex in the middle of each polygon, put a polygon over each vertex), and then rotate the edges to connect the vertices
    - This will result in the valences and the number of sides swapping values
        - In the square tiling case, we just end up with a SELF-DUAL: the polygons have 4 faces, and the vertices have valence 4, so all that happens is that the tiling gets shifted a bit by the dualing operation
- Not all the tilings, though, are so trivial:
    - A TRIANGULAR TILING has 3 sides, but results in a valence of 6; we basically put the triangles in a bunch of hexagonal shapes!
        - So, 3 sides, valence of 6 - if we "dual" this tiling, we'll actually end up with a hexagonal tiling, with valence 3
            - Why don't we see this tiling used in more apartments? Well, it's probably due to practical ceramic reasons; the triangle tips are sharper, and more likely to break
    - So, a HEXAGONAL TILING then will conversely have 6 sides and a valence of 3, and will "dual" into a triangular tiling
        - This looks like a honeycomb, so mathematicians, in their infinite wisdom, have nicknamed this the "Honeycomb tiling"

- (Professor Turk takes this opportunity to pass out a bunch of plastic models of platonic solids)
    - "I want to give out all of them now, so you all have a chance to hold a platonic solid"
- This leads us into the first PLATONIC SOLID we'll talk about: the CUBE!
    - As we know, a cube (or a "hexahedron") has 6 faces, 8 vertices, and 12 edges
        - ...what if we tried our "dual" tiling operation on this 3D cube, though? Let's try it!
- So, we put a vertex in the middle of each face, connect them with lines, and what do we end up with? An OCTAHEDRON - the next platonic solid!
    - This is a solid with 8 faces, 6 vertices, and 12 edges- huh! The number

of edges seems the same, and the numbers of faces/vertices have flipped!
    - So, the cube/octahedron are duals of each other

- It looks like dualing a platonic solid just swaps the number of edges/vertices - cool!

- As it turns out, there's an even simpler platonic solid: the TETRAHEDRON, with 4 sides, 4 vertices, and 6 edges
    - This looks like a pyramid with a triangular base, and could be constructed by taking 4 "opposite" vertices of a cube and connecting them
    - What's the dual of a tetrahedron, though? Let's see...
        - As it turns out, it just makes ANOTHER tetrehadron; like the square tiling, it's a self-dual!

- So, there are two more platonic solids; let's consider them
    - From the top-down, a tetrahedron looks like a "circle" of 3 triangles, and an octahedron looks like one with 4 triangles - what happens if we make one with 5 triangles?
    - Well, we'd end up with an ICOSAHEDRON, composed of triangles and with a total of 12 vertices, 20 faces, and 30 edges
        - If we dual THIS, we end up with the 5th and final platonic solid...
    - A DODECAHEDRON, composed of pentagonal faces and with 20 vertices, 12 faces, and 30 edges

- ...and as it turns out, that's it; there aren't any more platonic solids
    - If we add a 6th triangle to our "fan," it'll just lie flat; it won't form a valid 3D shape
    - Similarly, there's

- So, 5 platonic solids - what are they used for? Well, lots!
    - They're VERY useful for many-sided dice
    - A soccer ball is a close relative of the dodecahedron/icosahedron; it's known as a "truncated icosahedron"
    - Radiolarans, a microscopic creature, actually have an icosahedral body
    - Geodesic domes come from the nice property that icosahedrons can be subdivided into a sphere
    - Cubes are also very useful since they're the ONLY platonic solid that can tile a 3D space
        - If you're allowed 2 solids, though, an "octet" of tetrahedrons/octahedrons will also tile in 3D, and actually form a very

strong structure sometimes used for building struts
- In chemistry, you find the patterns of molecules adhering to one another tend to form platonic-solid-shaped lattices
- ...and in computer graphics, they tend to be very good "starting blocks" for building other shapes

- Now, let's talk about a very famous mathematical equation:

$$V + F = E + 2$$

- Where "V" is the number of vertices, "F" is the number of faces, and "E" is the number of edges
- This is EULER'S EQUATION, and actually holds not just for platonic solids, but for a whole host of other shapes
- In fact, it'll hold for any 3D solids that don't have "handles" - it's pretty general!
- Why "+2"? That doesn't seem very clean, but remember: vertices are 0-dimensional, edges are 1-dimensional, and faces are 2-dimensional
- So, the equation is basically saying "the number of even-dimension geometric objects (vertices and faces) is equal to the number of odd"
- So, the 2 is the number of volumes - there's an inside and an outside for each 3D solid!

- "...I'd like the platonic solids to come back, please"
- All right, that's the day!

//**********/
/
//********** Fractals - April 8th, 2019 **********//
//**********//

- So, the last lecture is rapidly coming up, and we don't have anything to talk about for it!
- You'll actually have an opportunity to vote on what we'll talk about for the last lecture, so think about what you want to learn!

- Today, though, is FRACTAL DAY!
- Fractals are, loosely speaking, infinitely detailed geometric objects

- They started out as mathematical oddities, but as it turns out, fractal-like patterns show up EVERYWHERE in nature (trees, rivers, mountains, coastlines, etc.)
  - The common theme of these things are recursion and iteration on geometry
- Way back in the 1880s, Georg Cantor (one of Professor Turk's favorite mathematicians) came up with a simple, strange pattern:

  - Draw a straight line segment:

  - Remove the middle third of that line:

    --- ---

  - Remove the middle third from the remaining 2 lines...

    --- --- --- ---

    - ...continue to infinity

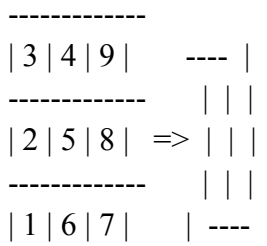
    - - - - - - - -
    . . . .

  - If we keep doing this, we'll end up with a sort of FRACTAL DUST, where we have an INFINITE number of unconnected points but, surprisingly, a total length of 0!
    - How can we have something infinite in quantity, with no length? That's mathematically crazy!
- As it turns out, there's a 2D analogue of this known as "Sierpinski's Carpet" that was found in 1916:

  - Take a square
  - Divide it into 9 smaller squares, and remove the middle one
  - Do the same thing for each of the 8 remaining squares, ad infinitum

  - Mathematically, this means that the surface is completely covered in holes, but somehow it still remains a single connected surface
- More usefully, however, are what are known today as PEANO CURVES
  - Consider a square, divided into 9 sub-squares, and visited in the

following order:



- You know the drill by now; for EACH of these 9 squares, let's divide it into 9 smaller ones and do the same thing!

- This'll get us an "infinitely wiggly" curve that passes through EVERY point in the 2D square, and has no tangent - that's insane!
    - This, as it turns out, is one type of SPACE-FILLING CURVE that can touch all the points in a 2D plane, and that are sometimes used in 3D printing (since it means we never have to lift our "pen point" to fill in something)

- A final example of fractals is the KOCH SNOWFLAKE, where we take a straight line segment, split it into thirds, replace the middle segment with 2 lines to make a triangle-shaped "bump", and then repeat for each of the 4 resulting line segments
    - This'll make a snowflake-like figure that's infinite in length but that has NO tangent anywhere

- Those're some examples of fractals, but again: what ARE these things? To answer that, we need to consider the concept of dimensionality in geometry
    - So, what do we mean when we say a square is a "two-dimensional" shape? Well, we mean that if we doubled the length of the square's sides, it would quadruple in area; we'd end up with 4 copies of our original square!
        - Similarly, for a 3D cube, doubling one of the cube's sides would get us a cube 9x the volume of the original cube!
    - Because of this, a common definition of a shape's "dimension" is the HAUSSDORF-BESICOVICH DIMENSION, which states that:

        dimension = log(repetition factor) / log(linear scale factor)

        - So, for a 2D square, this'd give us:

            d = log(4) / log(2) = 2 * log(2)/log(2) = 2

- That's pretty common sensical, but how would this apply to our fractal examples, like the Koch snowflake?
  - Well, each iteration of the pattern turns 1 line segment into 4, so the replication factor should be 4
  - ...but how does the length change? Well, each of the 4 line segments will end up being 1/3 the size of the original, giving us a scaling factor of 3 (I think?)
    - Using these numbers, we end up with a dimension of:

      $d = \log(4) / \log(3) \sim= 1.2$

    - Not an integer dimension? That's WEIRD!

- For a long time, these were just considered fun shapes to draw, but in the 1970s Benoit B. Mandelbrot noticed that these fractal patterns show up EVERYWHERE in nature, and tried to study them seriously
  - As just a few examples:
    - Blood vessel patterns seem VERY similar to space-filling curves
    - Rivers seem to bifurcate in a fractal way
    - Some star distributions seem to follow this pattern
  - Lawrence Carpenter quickly built upon this work, and came up with an algorithm for drawing mountain-like shapes in 2D:
    - Start out with a straight, 1D line segment of some length
    - Add a new point at the midpoint of the line segment and raise it by a random amount
    - For each of the 2 new line segments, add a new point in the middle as well and raise it by a random amount, cutting the "scaling factor" for the maximum height in half
      - This decreased scaling at each iteration is CRUCIAL so that we don't just generate random noise
    - ...continue iterating until you're satisfied
  - We can extend this "CARPENTER MIDPOINT ALGORITHM" into 3D as well!
    - Start out with a flat, 2D triangle
    - Add a point to each midpoint of the 3 line segments, connect the 3 points together, and raise it by a random amount
    - ...continue iterating until you're satisfied
  - These sorts of mountains, and similar algorithms for them, are used pretty frequently in film and games to create procedural landscapes

- As you might remember, our Julia set from Homework 4 is ALSO a fractal -

let's take a look at that!
- If our constant we're adding is 0, we just end up with a circle of points
- Otherwise, it'll form a complex fractal, which might be fully connected OR turn into disconnected fractal dust
  - Luckily, there's a way to tell which it'll end up being:
    - If the initial point Z_i = (0, 0) at the origin ends up staying constrained near the origin, the Julia set is connected
    - Otherwise, the set'll be disconnected!

- Julia sets are very closely related to one of the most famous fractals of all: the MANDELBROT SET!
  - In Julia sets, we change the value of Z_i but keep the value of C itself fixed
    - In the Mandelbrot set, we do the opposite: we keep Z_0 fixed at (0, 0), but vary the value of C we're using!
      - This means that any "C" points that are part of the Mandelbrot set will give us a connected Julia set, while any other choices of C will give us unconnected Julia sets!
  - As it turns out, the WHOLE Mandelbrot set is connected - unlike some Julia sets, it's all one big piece
- There're also 4D variants of the Mandelbrot known as the MANDELBULB, and they look REALLY cool!

//**********/
/
//********** Volume Rendering - April 10th, 2019 **********//
//**********//

- Alright, we're voting on what we should have the last lecture be on - let's go!
  - Keep in mind that these are TECHNICALLY still lecture materials, and so a question or two on whatever topic you choose *might* still appear on the final...
    - And, after a highly technical system of hand-raising, the winner is: GAME RENDERING METHODS!
      - We'll also try to talk about the runners-up, which were virtual reality, procedural content generation, and fluid simulation (with maybe a little bit of non-photorealistic rendering thrown in for good measure)

- So, with half the class gone by, let's talk about VOLUME RENDERING!
  - This is a technique that's used a lot in the medical field, and the idea is that, instead of storing a 3D object as a bunch of polygons, we store them using VOLUME DATA: values given in a 3D grid
    - Each data point in the grid is known as a VOXEL (for "volume element"), one for each vertex of the grid
      - ...the X, apparently, just wanted to inject some pizazz
    - Typically, this is just a 3D rectangular grid; a typical size for a grid is in the hundreds to thousands of voxels long on a side (e.g. 128x128x128)
  - Where does this data come from? Usually, it'll come from real-world sources where we need to know what's going on INSIDE an object as well, like CAT scans (measures X-ray permeability), MRIs (measures hydrogen), various engineering datasets like:
    - fluid simulations with pressure, vorticity, etc.
      - For the fluid simulation, note that we do NOT store the velocity at each point; that'd make it a vector field, which is a different beast entirely
    - Temperature gradients for a room
    - Stress/strain information for a building
    - ...and so on

- How do we actually render these voxels, though?
  - There are 2 main approaches we can take:
    - ISO-SURFACE METHODS, where we create polygons based on the voxel data, and then render those
    - DIRECT methods, where we create a rendering directly from the voxel data (via raytracing, etc.)
      - This is more commonly used when we need transparency

- Let's look at the Iso-Surface methods first
  - We could just put a cube wherever a voxel appears, and color it based on its data, but that'd give us a VERY blocky model - we can do better than that!
  - Instead, we'll usually need more sophisticated techniques; let's look at a common one called the MARCHING CUBES algorithm!
    - For simplicity, we'll look at a 2D version of it instead, called the "Marching Squares" technique
      - Suppose we have a 2D grid of data, each voxel of which contains a

single scalar value (it could be temperature, density, etc.)
- Arbitrarily, we'll say that any values >= 5 are "inside" our shape, and values less than that are "outside" (i.e. they'll be treated as empty space)
- So, we'll go through and label each of the voxels as inside/outside - and then the magic happens!
- Along each line segment connecting 2 of the voxels, if one voxel is inside and the other is NOT inside the surface we'll linearly interpolate our cutoff value (in this case, 5) between those 2 points
- We'll then connect all of those interpolated points together with line segments so that all the "inside" points are closed into the shape
- In 3D, we'd connect these points with polygons instead of a line segment, and each one of the points would be vertices instead

- We'll talk more about the 3D version of this algorithm come Friday

//**********/
/
//********** Volume Rendering (cont.) - April 12th, 2019 **********//
//**********//

- (blank)
- ...
- ...
- BOO! I HAVE SCARED THEE!

- So, on Wednesday, we started to talk about volume rendering, where our model is stored as a 3D array of "voxels"
- We began talking about how we might render these models using the "marching cubes" algorithm, and went through a 2D version of it!

- This 3D version, the MARCHING CUBES algorithm, is one of the most cited in all of computer graphics, so let's talk about it!
- Assume we're looking at a "neighborhood" of 8 different voxels, each forming the vertex of a cube, and each one considered to be either "inside" or "outside" of the surface

- Because this is now in 3D, there're more possible cases for us to consider (about 14 or 15 in all), but the idea is the same: based on which vertices are inside/outside, we draw a different kind of polygon to connect them, eventually "enclosing" all of the inside vertices in a polygonal shell
    - Probably the most complicated case is where we have 4 inside vertices in a tetrahedral pattern, which'll result in a hexagonal polygon
- Eventually, because of the rules for these different cases, all of the polygons we create will link up with one another to form a continuous surface - and then we can render that!

- So, that's the isomorphic way of rendering these volumes, but what about the DIRECT rendering approach?
    - There are several different techniques to do this; we'll be mostly considering the raytracing approach, but most of them work using a similar several-stage process:
        1) Classify each voxel
            - Is the voxel representing bone? Air? Skin? Muscle?
            - Based on this data, we can determine what we need to render the surface (e.g. its surface color, opacity, reflectance, etc.)
        2) Calculate the voxel's shading
            - This is where we figure out what its normal is, any lighting/reflections, the color it should be rendered as, etc.
        3) Actually render the voxel
            - This is where we do visibility checks (is the voxel hidden?), etc, and actually draw the voxel on a screen
    - Usually, we'll assume that each voxel is filled with a bunch of tiny, identical, opaque particles of its material
        - Each individual voxel will have an OPACITY value between 0 and 1 (0 is transparent, 1 is opaque) representing the chance that a light ray will pass through the voxel, and a COLOR (i.e. the color of the voxel after lighting, etc. is taken into account)
            - Usually, the voxel's color will be calculated using something like this:

                finalColor = a_i*c_i + (1 - a_i)*(everything else)

            - Where "a_i" is the opacity of the voxel and c_i is the color we've calculated for the voxel's surface

- What's this "everything else," though? Well, it's the color of everything BEHIND the voxel, letting us handle transparency
    - For a given ray, this might mean recursively calculating that color by shooting another ray behind the voxel (a la reflection from earlier in the semester)

- How can we actually use raytracing to directly rendered these voxels, though? How do the pieces all come together?
    - Oftentimes, we'll have a ray that's going diagonally through a bunch of voxels, and that means we can't render every same-material voxel the exact same (e.g. the corners of an amber cube are more translucent than the center, because they're thinner)
        - Because of this, we'll need to interpolate our opacity/color within each voxel using TRI-LINEAR INTERPOLATION to figure out how transparent, etc. the voxels are
    - Where do these opacities come from, though? We mentioned it came from the "classification" step, but what does that actually mean?
        - Let's say we have some data from a CAT scan, and each voxel has a floating-point value associated with its density
            - Air might have a very low density value, while muscle has a certain density "range" and bone has another density range
                - What if two of these density ranges seem to overlap? (e.g. it's very difficult to distinguish, from the data, if something is tough muscle or weak bone?)
                    - That's a very real-world issue, and with the Broom of Fuggedaboutit Professor Turk is choosing to push it underneath the Rug of Ignorance (ostensibly within the Room of Messy Graphics?)
            - Once we've figured out what the voxel should be, we give it the corresponding color/opacity for its material
    - How do we figure out the surface normals? Well, trust me, PLENTY of techniques exist - but alas, we don't have enough lecture time to get to them

- So, that's a wrap for volume rendering, and now we can start getting into some of the fun stuff YOU guys chose to bring upon yourselves - starting off with NON-PHOTOREALISTIC RENDERING!
    - The goal here, oftentimes, are to try and capture the hand-drawn feel of traditional animation and art

- One common feature of early animations were these thick outlines for the characters, which directly inspired a style of rendering called CEL SHADING!
    - There are 2 especially prominent features at play here:
        - Dark outlines/silhouettes for the objects
            - To actually make these silhouettes, Professor Turk is aware of two general approaches:
                - To render the object normally, then do edge detection on the pixels (using a Laplacian filter, perhaps, where we try to)
                    - A "Laplacian" basically means taking a 2nd derivative in 2D/3D, and'll be relevant when we get to fluid simulation
                - To identify possible silhouettes based on the polygon normals
                    - The idea here is that if a polygon's normals are facing towards the eye, it's "front-facing," while if it's facing away from the eye it'll be a "back-facing" polygon
                        - Wherever a forward/back facing polygon are adjacent, we say that edge where they meet is a silhouette edge, and render it as a dark line!
        - Simple shading with relatively few colors
            - The idea here is to use the traditional shading equation where "V = N*L," but to NOT render V as a continuous range of colors
                - Instead, we'll categorize it into discrete categories and render ALL the values within that range as a single color (e.g. 0 <= V <= 0.2 is dark green, 0.2 <= V <= 0.5 is regular green, etc.)
    - One of the first video games to use real-time Cel shading was "The Legend of Zelda: The Wind Waker," which looks distinctly more cartoony than a traditional 3D shader

- Alright, next week we'll start going through more end-of-year lecture material - hurrah!

//**********/
/
//********** Fluid Simulation - April 15th, 2019 **********//

//**//

- So, fresh from the weekend, let's start off the day by talking about Project 5 (implementing Loop subdivision on a corners-representation model)
  - Going from corners to vertices is EASY - but going from a vertex to a corner? That's harder
    - Why would we EVER want to do this, though? To process all the corners around a vertex, the easiest way is to get a corner near the vertex and then use the swing operator!
    - The "vanilla" corners representation doesn't let us do this, but we can still do this efficiently with some extra information:
      - For each vertex, just store an arbitrary adjacent corner along with it; then, you can start at that corner and work your way around!
  - Another issue you might've encountered is that, when subdividing, we need to make a new vertex for each edge - in other words, a vertex for each pair of opposite corners!
    - That means we need to make sure we only create ONE vertex for each pair, instead of two; you need to think of some way to handle this

---

- Today, we're going to cover another topic you voted to hear about: fluid simulation for animation!
  - As it turns out, Professor Turk is pretty familiar with this; he's had graduate students do research on this for theses he's overseen before

- Why simulate fluids, though? Well, because there're a BUNCH of applications for realistic fluid physics:
  - Animated movies
  - Computer games
  - Medical simulations (e.g. blood flow in the heart)
  - ...and, most importantly, because it's FUN!

- There's actually a whole field known as COMPUTATIONAL FLUID DYNAMICS, and it borrows quite a bit from engineering and mathematical simulations
  - For graphics applications, we usually prefer a technique known as the "finite differences" method

- Now, here's a very scary slide you might not understand at first: the NAVIER-STOKES EQUATIONS!

- These look AWFUL, like a bunch of nonsense greek letters, but don't worry
- we'll go through it!
    - There are two of these equations:

        laplacian*u = 0

        - This is the INCOMPRESSIBILITY equation, and basically says the change in

            u_t = k*laplacian(u) - (u*gradient)*u - del*p + f

        - This "u_t" represents the change in velocity of the particle, and has 4 different terms:
            - The "Diffusion term", k*laplacian^2(u)
            - The "Advection term," -(u*gradient)*u
            - The "Pressure term," del*p
            - The "Body force," f

- Before we dive into each of these, we need to talk about "Finite Difference grids"
    - Here, we say that all of the values in our simulation live on regular square grids, which gives us:
        - A SCALAR FIELD, representing how much "material" is in each cell (e.g. smoke or dye that'll be pushed around by the liquid)
        - A VECTOR FIELD, representing the fluid velocities at that grid point

- The DIFFUSION TERM looks like this:

    c_t = k*laplacian(c_t)

    - Where c_t is the change in value, and k is the "diffusion rate," saying how fast the material spreads out
        - What's the "laplacian," though? Well, it's basically just the change in value of the cell "t" relative to its neighboring values
            - In the finite difference method, we'll approximate this laplacian by getting the values of the 4 cells adjacent to us
    - So, using this equation, for a given cell "cxy," we'll update the cell's value via diffusion like this:

        c_xy = c_xy + k*dt*(c_(x-1, y) + c(x+1, y) + c_(x, y-1) + c_(x,

y+1) + 4*c_xy)

- In other words, we're taking the current value, and (scaled by some amount) adding the differences between our current value and the neighboring values (i.e. why we're subtracting "ourselves" 4 times)
- Interestingly, diffusion is the EXACT same as blurring mathematically (it's actually equivalent to Gaussian blur)

- What the Navier-Stokes equations say is that we need to diffuse the VELOCITY of our fluid, so let's do it!
  - We'll represent the velocity of a given cell "ij" as

    u_ij = (xVel = u_ij^x, yVel = u_ij^y)

  - In the context of these equations, "k" is equal to the VISCOSITY of the fluid
    - Interestingly, viscosity does NOT have to be constant throughout the simulation; it can vary by position, by temperature, etc.
      - (Video of a simulated bunny candle melting; "no bunnies were harmed in the making of this film")
  - We'll then separately diffuse the x/y velocities (the xVel has NO effect on the yVel, and vice-versa), like so:

    u_ij_xVel = k * laplacian(u_ij_xVel)
    u_ij_yVel = k * laplacian(u_ij_yVel)

- The second term is the ADVECTION TERM, which is all about pushing stuff around
  - Specifically, we're doing something a bit backwards known as "semi-lagrangian advection," where we choose a place for the cell values to end up and calculate how much material actually moves to that destination
    - The equation looks like this:

      u_t = -(u*gradient)*u

      - Where, again, u_t is the change in value, u*gradient is a CRAZY notation for the gradient of u, and u is the current value of whatever we're updating
    - Again, we'll calculate this separately for the X/Y cases
  - Advection is relatively easy to code, and it's stable even at large timesteps

- The PRESSURE TERM is related very closely to incompressibility by something known as DIVERGENCE!
  - A high-divergence vector field is where we have a little bit of material moving into something and a LOT of stuff coming out; low divergence is the opposite, where we have a lot of stuff going into a space by little coming out
    - Both of these are unrealistic! For incompressible fluids like water, we'd expect ZERO divergence, meaning the amount of stuff going in is EXACTLY balanced by however much is getting pushed out
  - To enforce this incompressibility, we'll first do our velocity/advection calculations
    - We'll then find the "closest" possible vector field to our current state that's divergence free, calculate the divergence, and then calculate the pressure
      - How do we measure divergence? The symbol for it is a dot product, like this:

        divergence(u) = del*u_ij

        - Using the finite-differences method on a grid, we'll do this by basically asking "how much material is coming into the cell from one side, and leaving it on the opposite side?"
      - Equation wise, that looks like:

        divergence(u(i,j)) = (u_x(i+1, j) - u_x(i-1, j)) + (u_y(i, j+1) - u_y(i, j-1))

    - To calculate our pressure, we'll start off with this:

      u_new = u - del*p

      - And then take the divergence of both sides...

        del*u_new = del*u - del*(del*p)

      - Which, since the divergence on the left-hand side should be 0, gets us this:

        del*u = laplacian(p)

- We already know del*u from our equation above, but we DON'T know what the laplacian of p is
- We can say the new value of p is:

p_new = p * epsilon(del*u - laplacian(p))

- Basically, we take the old pressure, change it a tiny bit (somehow), and
- More mathematically, the new pressure is this:

p_new(i,j) = p(i,j) + epsilon(del*u(i,j) - laplacian(p))

- Which, at last, we can use to get our divergence-free velocity (I think?):

u_new = u - del(p)

- So, this gets us the "nearest" vector field from our original that's divergence free!

- Okay, this is a LOT - I don't expect you to follow every little step, but I want you to see that every individual step isn't that hard. You really could code this up on your own if you wanted to!

- So, putting this all together, how do we actually simulate a fluid?

1) We'll first diffuse the velocities
2) We'll then advect those velocities
3) ...and then add in any body forces (e.g. gravity)
4) We'll then do pressure projection (i.e. calculate the pressures)
5) Then diffuse the scalar VALUES for the fluid (the amount of smoke/dye/etc. inside the fluid)
6) Finally, we'll advect those scalar values, and repeat from step #1
   - If you want to see a concrete example of this stuff, here's an article that discusses implementing this specifically for games: http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf

- To model objects INSIDE of the fluid, we can do rigid-body simulations using effects very similar to our pressure projection calculations
  - Basically, we'd solve our Navier-Stokes equations as if the objects were

a fluid, calculate the forces on those rigid-body cells from collisions/densities, and then enforce rigid motion for all of those cells

- What about different sizes of fluids? What's the difference between simulating a lake, a puddle, and a droplet?
  - Well, at small scales (centimeter-sizes), surface tension effects become FAR more important, whereas they're negligible for surfaces much larger than that
    - The idea here is that drops of water will be convex, drawn together by surface tension
      - On hydrophilic surfaces, the drop tends to spread out more; if it's hydrophobic, though, the water drop will bead up and become more spherical

- ...and, a messy conglomeration of note-taking later, that's it for today! Come open the next goody-box of topics on Wednesday; 'til then, adio!

//**********/
/
//********** Virtual Reality - April 17th, 2019 **********//
//**********//

- Don't be happy it's today because it's not tomorrow; be happy that it's today. And when tomorrow comes, even if it feels rough, be happy for that, too; it'll pass!
  - Also, fill out the CIOS! Greg Turk's done an awesome job teaching throughout this whole semester, and he deserves some credit for that!

- Alright, we're gonna talk about VIRTUAL REALITY today, and there're 3 BIG pieces any VR system needs:
  - We need to track the user's head motion
  - We need to create images of the virtual world in real time
  - We need to display those images convincingly to the user

- The history of VR is actually a rather long one, with 3 sort of distinct "waves" in history
  - In the 1960's, Ivan Sutherland first started looking into the fundamentals of VR
    - Ivan Sutherland (who's still alive as of 2019!) is considered the

"father" of computer graphics as an academic field, and he famously created the world's first GUI-based program (a constraint-based design package called "Sketchpad," which directly inspired CAD) and arguably the first object-oriented programming language
    - He also famously described computer graphics as "a looking glass into a mathematical wonderland"
- After winning a Turing Award for his research, he worked at the University of Utah for several years trying to create an "Ultimate Display" that would be as close to real life as possible
    - As a prototype, they built a device with all 3 of the key VR components (a tracking device, a head-mounted display, and real-time rendering), but it was actually an AUGMENTED REALITY device, projecting virtual stuff onto the real world
        - However, the device was bulky and INCREDIBLY unwieldy, and required a ceiling-mounted pipe to track movements, earning it the nickname "The Sword of Damocles"
- Several research labs continued pursuing this kind "virtual environment" work (such as Henry Fuchs at UNC (Professor Turk's Phd. advisor)), but it certainly wasn't mainstream
- In the late 1980's, Jaron Lanier founded a company called VPL - "Virtual Programming Language" - and coined the term "virtual reality"
    - VPL sold complete virtual reality systems that had all the key components, a magnetic motion tracking system, and a "data glove" that let people use their hands to interact with the virtual world
        - ...it wasn't very accurate at ALL, but it did work!
    - The work Lanier did here drew a LOT of attention from the tech field, so what happened?
        - Well, the technology at the time was still a little crude, and it didn't live up to the hype: the images weren't realistic, the tracking wasn't good enough, it was heavy, and motion sickness was a constant issue
            - MOTION SICKNESS is when there's a disconnect between how fast your eyes see you moving and the acceleration your body feels
                - Some people have thought this is because the body thinks you've been poisoned when you get dizzy, leading to good ol' vomit-on-the-wall syndrome
        - There was also no obvious market for the tech - everyone thought it was cool, but no one wanted to actually buy it!
    - So, once again, VR work died off, and was confined to a handful of research labs

- Then, in 2010, Palmer Luckey made a custom head-mounted display prototype that had surprisingly good motion tracking - enough to significantly decrease the motion sickness problem
    - After a successful Kickstarter campaign, he founded Oculus VR to expand on his display, and in 2014 Facebook bought Oculus for ~3 BILLION dollars, basically confirming that this was a field they wanted to take very seriously

- That's some of the history, but what about the tech that goes into VR systems today?
    - Nowadays, we have GPUs EVERYWHERE that can render millions of polygons at well over 60 Hz, which is a MASSIVE advantage we have over the early 1990s VR attempts
        - Because of this, image synthesis itself is mostly a solved problem, perfect photorealism notwithstanding
    - More tricky is the Head-Mounted Display, which'll be how we display these images to the user
        - We need the HMD to deliver separate images to each eye so that we can get a 3D parallax effect
            - Oftentimes this is done using 2 small LCD or OLED displays, with lenses to make the image appear at a comfortable distance (instead of being mashed directly into your eye)
                - Ideally, this HMD will also have a wide field-of-view
                - Having it be as light as possible would also be nice, both for user comfort and because heavy headsets can contribute to motion sickness
                - Wireless headsets would also be a plus, so that users can't trip on wires as they walk (that's an actual concern!)
- Let's take a closer look at the Oculus Rift's headset, specifically
    - The headset is able to obtain a pretty wide field-of-view using FRESNEL LENSES to focus the light
        - This means that, instead of a traditional thicker lens, we can get a similar focusing effect by having "slices" in the lens; in exchange for a little bit of image quality, we get a much smaller, lighter lens!

- The last piece of the puzzle is tracking systems - and there's still a LOT of variability in how these work!
    - Our main concern here is to figure out the x,y,z position of the user's head AND their head's orientation, all at 60 Hz or faster
        - We then need to pass this information to the rendering system, render

from the user's position, and put that image on the screen
- As mentioned, there's several techniques we could use:
  - Magnetic tracking systems have fallen out of favor, since metal inside the room can mess up their calibration
  - Nowadays, OPTICAL systems are pretty popular
    - "Outside-in" systems uses an external camera to track "beacons" placed strategically on the HMD
      - This is the system Oculus uses, where they have invisible Infrared LEDs scattered around the HMD, which the camera observes
        - The camera then uses traditional computer vision techniques (triangulation, etc.) to figure out where the HMD is; the more beacons it observes, the higher its accuracy
      - However, the accuracy here decreases as we get farther away from the camera, making this technique best for seated positions where the user isn't moving around much
    - "Inside-out" systems put the camera on the HMD itself, and then try to observe reference points placed throughout the room to figure out where it is
      - An example of something that does this is the HTC Vive, which has a larger tracking area than the Rift
        - Here, there are two "lighthouses" that sweep the room with a sheet of laser light: one horizontal sheet, one vertical
        - The headset then has a bunch of light sensors on it (not true cameras per se) that track this laser light to figure out its position

- So, what're the challenges for VR today?
  - Well, motion sickness still remains a problem; it's getting better, but it's not perfect by any means
    - Some people have almost no motion sickness issues, while for other people it's debilitating
  - There's also the lack of a "killer app" that'd give a reason for everyone to buy this
    - Video conferencing, video games, data visualization, medical operations - there're some use cases, but there hasn't been anything so compelling yet to justify everyone buying a $300 headset
    - Augmented reality might also be a thing to watch out for - the two

big systems around today (Magic Leap and Hololens) are both pretty similar and aren't perfect, but it's still early days

- Is this 3rd round of VR going to be "the one?" I don't know, and the market certainly doesn't know, but there are some very large companies pushing to make this a reality

- Alright, two more lectures to go! Come Friday to hear the next one!

//**********/
/
//********** Game Rendering - April 19th, 2019 **********//
//**********//

- Alright, the final exam is in ONE WEEK, in this room, from 11:30 AM to 2:10 PM
  - The exam IS cumulative, so stuff like 3D transformations, etc. are fair game, but the exam will focus more on the 2nd half of this course
  - BRING A LAPTOP! Most of the exam'll be on canvas (there's a chance there'll be a paper portion, but Professor Turk hasn't made up his mind)
    - The exam will be open-book/open-notes again, you just can't look stuff up on the internet

---

- So today, per your request, we're gonna be talking about GAME ENGINES!
  - Games have a lot of different components (collision detection, physics, animation, sound, etc.), but we're particularly concerned with game rendering techniques
    - Games need to run FAST (at least 30fps), so efficiency is a huge priority
    - There are a variety of popular game engines: Unity, Unreal, Cryengine, Source, etc. (many of which are free for educational use - they wanna get you hooked as a student)

- The key 300-point font headline here is REAL-TIME RENDERING: we need to balance having a high-quality image with fast render speeds, and those goals are directly opposed to one another
  - Different games prioritize these differently
  - There are two kinds of rendering in games:
    - IMMEDIATE-MODE RENDERING is when the CPU sends polygons to the GPU one-by-one, which lets us change object position per frame
      - This is great for dynamic stuff that's moving, but it can be SLOW

since the CPU has to send stuff to the GPU every frame
- This is what Processing uses
- RETAINED-MODE RENDERING is where we send polygons to the GPU just ONCE, and then render that
- This collection of polygons on the GPU is stored in the "vertex buffer object" (VBO); the CPU will ask the GPU to render it when needed, and because all of it's already on the GPU, no communication between the two is needed! That mean it's FAST!
- Almost all games use retained-mode graphics because of its dramatic speed boost - but, alas, it makes it harder to have a bunch of moving stuff on-screen at once

- So, a different strategy of speeding things up: DRAW FEWER POLYGONS!
- This seems REALLY obvious, but it's important, and a ton of effort has gone into making this work!
- One way is by using POTENTIALLY VISIBLE SETS
- This is where, for indoor scenes, we pre-calculate what parts of the scene are visible from a given room
- If the player is inside of that room, then, we know what rooms we don't need to draw! So we just ignore those rooms, which can lead to a BUNCH of saved time
- An alternate technique for doing this is the PORTALS method, where we'll define a bunch of invisible "portals" in bottlenecks of the rooms, like doors and windows
- If we can see the portal, we need to draw everything behind it - but if we can't, we can ignore everything behind the portal!
- Another common way of doing this is LEVEL-OF-DETAIL meshes
- The idea here is that each 3D model has several different versions of different complexity (either auto-generated or handmade by an artist)
- When the player is far enough away, they can't notice the higher detail anyway, so we'll use the less-expensive, low-quality model
- Interestingly, we can also go in the opposite direction with TESSELLATION, where we add more polygons to the model in a subdivision-like process if the player gets too close
- This alone will often make characters look too smooth, though, so we'll often combine this with a DISPLACEMENT MAP
- Because this can happen on the GPU natively, it's actually surprisingly fast

- Since lighting calculations (shadows, etc.) are expensive, what can we do to

speed them up here?
- One technique is BURNED-IN LIGHTING, where instead of actually calculating the lighting, we'll cheat!
  - Instead, we'll pre-compute the lighting ahead of time, store it in a LIGHTMAP texture, and then just use that texture without any light computations at all!
  - HOWEVER, this doesn't work for reflective surfaces, or if the object is going to move
- Another lighting technique that's NOT for efficiency - but instead for higher quality - is AMBIENT OCCLUSION
  - This is basically us trying to estimate how much "sky" or indirect light is visible at a point, mimicking light on a cloudy day
    - This'll result in deep creases in the object looking darker, giving it depth, and it's generally great for realistic shading (ESPECIALLY outdoor shading)
  - This is often done offline and baked into the textures, but various techniques have emerged to calculate this in real-time instead
    - How can we do this without ray-tracing? You can try to fake it with Z-buffers, but Professor Turk isn't familiar with the state-of-the-art techniques here

- Another common things games do: POST-PROCESSING!
  - The idea here is that we'll calculate the image of a scene, then use pixel shaders on the GPU to modify the image AFTER it's rendered
    - We'll usually consider the original scene as a texture, but we might consider separate objects as separate images and recombine them later (such as in G-Buffers)
  - We can use these to get motion blur/depth-of-field effects, vignettes, bloom, lens flares, etc. - let's go through these!
    - MOTION BLUR is where we pretend that a camera shudder is open for a non-instant amount of time; in real life, this makes moving objects appear "streaked" and blurry, so
      - To actually create it, we'll save the depth map of the image (e.g. the Z-buffer); the depth, combined with the X/Y position, can give us the 3D position of each pixel, which we can use to figure out how far each pixel moves between frames
      - We'll then apply a blur to each pixel, based on the direction it's moving and how far it's moved
    - DEPTH-OF-FIELD means that not all objects should be perfectly in focus, with other objects (near or far) being blurred

- To do this, we'll again render the image and save the depth map, then blur the parts of the scene that
    - We DON'T want to blur silhouettes, though, so we'll often just blur the "far" layer and keep objects closer than the "focal plane" in focus
- VIGNETTES are where the corners of our view are darkened
    - In a game, this is EASY to do; we'll just color the pixel based on how far away it is from the center of a screen!
- FOG/HAZE is when particles in the air, like dust or rain, scatter the light, especially for distant objects, making far-away objects look washed out
    - This is actually really easy; we'll render the image, save the depth/Z-buffer, and then blend it with the fog color based on depth
        - We could combine this with particles having a fog texture to get some more dynamic smoke effects
    - A more difficult affect are LIGHT SHAFTS (or GOD RAYS) through fog, where the light passes through some "participating media" and scatters the light in 3D
        - This is often done by using voxels to store how much fog/haze is in each 3D position, then stepping through the voxels to see how much light has been scattered
        - This is computationally tricky to do fast, but techniques exist to do this in real time
            - (cue "Book of the Dead" demo from Unity3D)
- BLOOM is the idea that bright lights seem to "glow" in the region around them
    - We can do this in post by rendering an image ONLY with the lights, blurring that image, and then re-compositing it with the rest of the image
- LENS FLARES are "echos" of bright lights bouncing around inside of the camera
    - We can do this by identifying bright pixels near the center of the screen, copying them radially, blurring them/increasing their size, and adding them to the image

- Let's now talk about something VERY recent: real-time ray tracing in games!
    - As we know, there are effects that we can do far more realistically with ray tracing than through rasterization: global illumination (i.e. bounce lighting), correct reflections, better depth-of-field, faster ambient occlusion, soft shadows, etc.

- For now, these are largely done as "mixed" effects, where most of the scene is still rendered with raster techniques, and then ray tracing is added on
    - (cue demo video for "Pika Pika")
- You'll notice this demo also talked about real-time self-learning agents, since Nvidia's raytracing cards also had hardware to help with neural nets

- Many of these techniques (and many, MANY more) are gone over in SIGGRAPH'S 2018 course on game rendering

- Alright, we'll have our last lecture on Monday - come to hear about procedural content, and possibly to say goodbye!

//**********/
/
//********** Procedural Content Generation - April 22nd, 2019 **********//
//**********//

- EXAM, THIS FRIDAY, THIS ROOM, 11:20 AM!
    - Be there, or be square (and receive a fairly large whack in the GPA-sensitive parts of your body)
    - Again, BRING YOUR LAPTOP! Most of the exam will be on Canvas; you're allowed to use electronic notes if you have them, but you may NOT use online resources

---

- The last lecture, and the last of our topics, is going to be looking at Procedural Content Generation (or: how I learned to stop worrying and make stuff using random numbers)
    - "If you forget EVERYTHING else from this lecture, remember: random numbers!"
    - What is "content," though?
        - Well, it's the "stuff" inside a video game or film, in particular the graphics, sound, story, AI, and so on
            - Traditionally, humans have had to create all of this stuff by hand, but that can be VERY slow: for graphics in particular, we need to model every bush, tree, NPC, car, mailbox, toothless newt, bespectacled wizard, etc.
    - If we could relieve some of this awful burden from ourselves and put it

onto our computer overlords, that'd be great!

- Procedural generation is used in a BUNCH of stuff, but particularly in games
  - Minecraft is probably the most famous of these, generating all of its terrain randomly
  - Simcity ALSO uses procedural generation to algorithmically place trees, NPCs, etc.
    - "Simcity sucked me in about a year and a half ago, and I had to quit it cold-turkey"
  - Spore uses PCG to generate different kinds of creatures

- So, the goal of PCG is to save time by creating content using algorithms, but what does that look like?
  - In particular, the algorithm needs to create a VARIETY of objects, with different shapes, sizes, colors and features- and to do that, we need random numbers!
    - One of the most famous techniques for this is PERLIN NOISE, which lets us get "controlled random numbers" in the range [0, 1] by passing in a 1D, 2D, or 3D position
      - If we change our position, though, the random value we get out changes SLOWLY, as if we were walking down a hill
        - This makes Perlin noise great for creating textures that aren't just random "white noise", or for making hills, or anything where we need the values we get out to be related to each other
        - A few more example applications of this noise:
          - Using it as a bump map (either as-is or after some processing to get different patterns)
          - To get "turbulence" by combining different "frequencies" of this noise (e.g. to get soap-bubble patterns, or marble textures)
        - Ken Perlin actually received an Academy Award for this research, since his noise functions are used EVERYWHERE in films
      - The way this function actually works is (roughly) that we pass in a seed value, get the numbers we need, and then interpolate between them
        - We can use 3D Perlin noise as well, where we interpolate between 8 points instead of a mere 2

- One common use of PCG, though, is to create terrain for games and movies

(mountains, dunes, cliffs, etc.) - "But GREG," you ask, "HOW!?"
- Well, we can use different noise textures as heightmaps, or generate the terrain with fractals *cue examples reel*
  - One thing missing from most of these examples is erosion from water and wind - but, as it turns out, this has been an area of research as well
    - We can simulate water erosion by pretending that our terrain is on a grid, and that each step "rain" is deposited throughout the grid
      - Fast-moving water will "pick up" some of the sediment from the ground, while slow-moving water will drop it back down, scooping out valleys and rivulets
- We might also want to allow for "user-guided terrain," where we try to give people some control
  - Howard Zoo, a former Phd. student here at Tech, did some research on this topic some years ago, and his results managed to look pretty cool

- But terrain isn't the only thing we want to make - we might also want to make PLANTS!
- One well-known way of doing this is using grammars, such as L-Systems (which are a context-sensitive grammar, if you're familiar with such things)
  - A GRAMMAR (for computer science purposes) is just a set of rules where we replace something with another thing
    - These come up a LOT in compiler design, but we can also use them for our own nefarious purposes
  - These are rules that guide how the segments of the plant branch and grow
    - We'll start out with a single plant segment, and then have various "rules" for what we replace each section with and how it's modified at each step
  - These L-systems can also communicate information down the system - "Is a flower blooming? Is a hormone present?" - and so on
    - How would we create a flower in the first place? Well, flowers typically have their petals in a spiral pattern, and it turns out you can get a pretty realistic flower by arranging them in a spiral with an angle of ~137 degrees (a number related to the golden ratio), known as the "phyllotaxis" method
- Where does the randomness come into grammars, then? Well, we can assign random probabilities to which rule we use, or for what the results of a rule will be (e.g. how many petals the flower has)

- Can we do this for human-made objects, like buildings? As one famous builder said, YES WE CAN!
    - We can use similar grammar systems to the ones people use for plants to modify building objects, using rules to add windows, floors, various details, modify the floorplan, etc.
        - Another group, interested in creating a virtual Pompeii, created a grammar system that generates city plans for ancient Roman towns
    - Just like we can make a single building, we can also generate a map of city streets!
        - The idea here is that the user provides some information about what's in the scene: important resources, population centers, rivers and lakes, etc.
        - From there, we can apply some grammar rules to figure out a reasonable layout of streets, fill in the blocks with building models, and so on
            - We can allow artists to specify what directions the roads should go in, where certain buildings should be placed, etc., if we so choose (e.g. using "tensor design")
    - If any of you have played "Cities: Skylines," we can control where some of the roads and building zones are, but the algorithms themselves control traffic patterns, individual buildings, and so on - you name it!

- (There is an example of No Man's Sky's procedural generation on the screen, but I have chosen to stop taking notes. I apologize for the inconvenience)

- Alright, and that's it for today - and for the class! Come to the final on Friday, but otherwise, live your lives! Carry on! You're free!