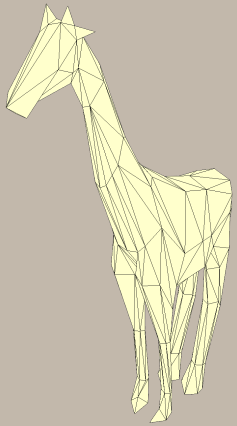


Lecture 19 - Visualization

Philip Caplan

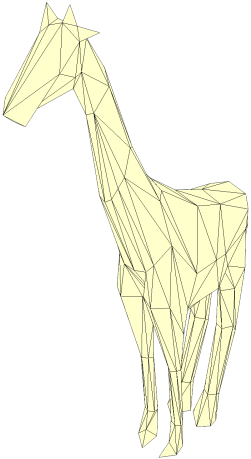
CSCI 0422 - Geometric Modeling (Spring 2022)

Middlebury

Learning objectives

By the end of this lecture you will be able to:

- identify and describe how mesh data is passed through the graphics pipeline,
- describe the roles of each shader involved in the graphics pipeline,
- transform vertex coordinates from object space to screen space,
- calculate ambient, diffuse and specular terms in a fragment shader.



Getting started.

Exercise for today is posted on the course website (**go/cs422 -> Week 12T -> exercise**):

https://csci422-s22.gitlab.io/home/demos/visualization.html

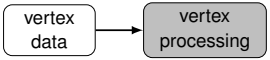
solution to class exercise will be available (via "solution" button) after class.

source for `flux360` **is now available in** `src/flux-base/flux360/` **(after** `make update`**)**

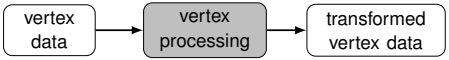
The graphics pipeline processes vertices, then fragments.

vertex
data

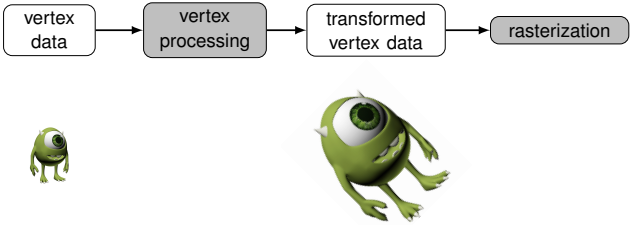
The graphics pipeline processes vertices, then fragments.



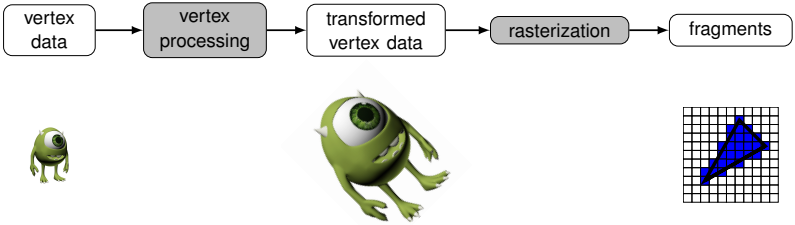
The graphics pipeline processes vertices, then fragments.



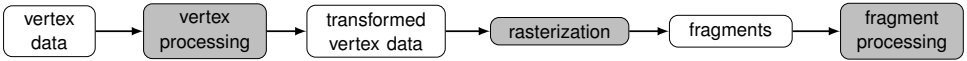
The graphics pipeline processes vertices, then fragments.

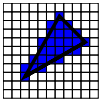


The graphics pipeline processes vertices, then fragments.

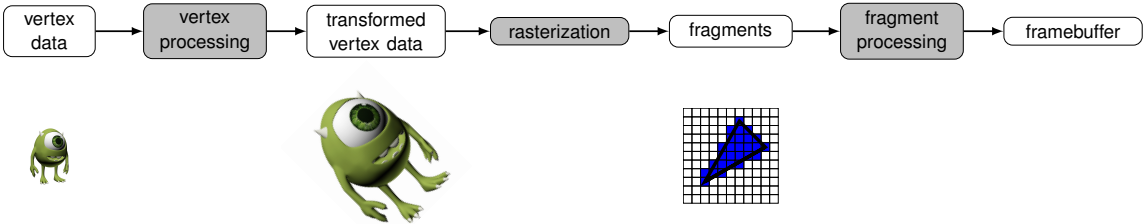


The graphics pipeline processes vertices, then fragments.





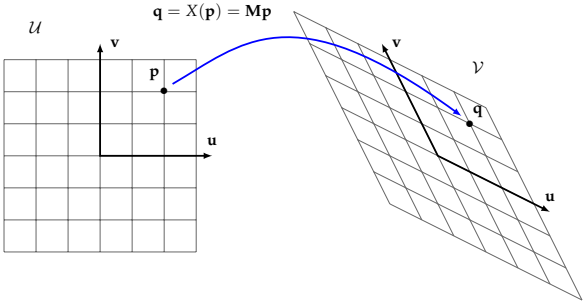
The graphics pipeline processes vertices, then fragments.



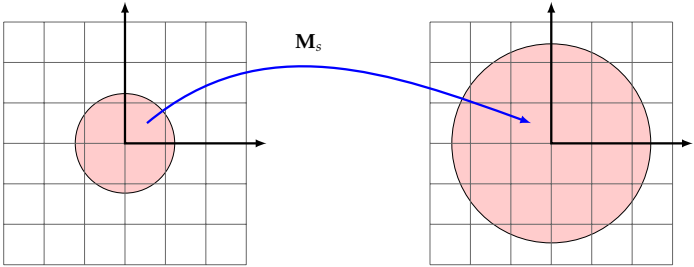
Linear transformations map points from one space to another.

$$\mathbf{q} = X(\mathbf{p}) = \mathbf{M}\mathbf{p}$$

$$\mathbf{p} = X^{-1}(\mathbf{q}) = \mathbf{M}^{-1}\mathbf{q}$$

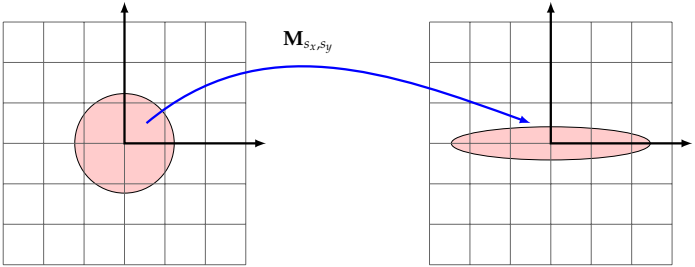


Recall some 2d transformations: uniform scaling



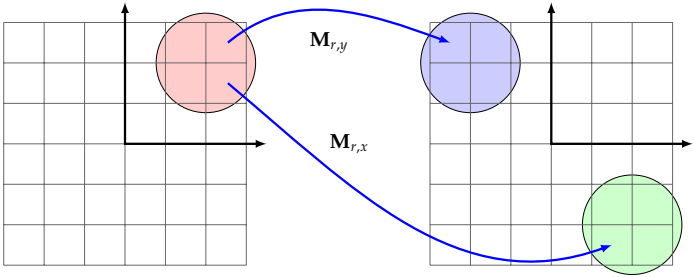
$$\mathbf{M}_s = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}, \quad \text{e.g :} \quad \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} 2p_x \\ 2p_y \end{bmatrix}$$

Recall some 2d transformations: non-uniform scaling



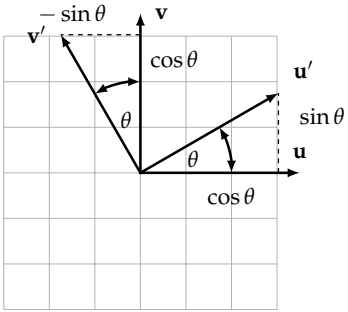
$$\mathbf{M}_{s_x,s_y} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}, \quad \text{e.g :} \quad \begin{bmatrix} 2 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} 2p_x \\ p_y/3 \end{bmatrix}$$

Recall some 2d transformations: reflections



$$\mathbf{M}_{r,x} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \mathbf{M}_{r,y} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Recall some 2d transformations: rotations



$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

What about translations? $\rightarrow$ homogeneous coordinates.

Main idea: add an extra component which is equal to 1 so we can use the same framework to handle translations.

2d transformations

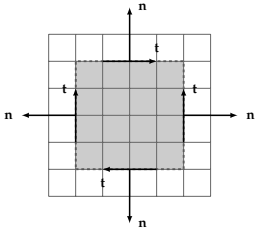
$$\mathbf{T} = \begin{bmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ \mathbf{1} \end{bmatrix}$$

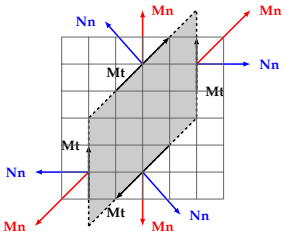
3d transformations

$$\mathbf{T} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ \mathbf{1} \end{bmatrix}$$

We must transform normals differently.

$\rightarrow$ because we want them to still be normal to surfaces after a transformation!





$$\mathbf{n}_M^T \mathbf{t}_M = (\mathbf{N}\mathbf{n})^T (\mathbf{M}\mathbf{t})$$
$$= \mathbf{n}^T \underbrace{\mathbf{N}^T \mathbf{M}}_{\mathbf{I}} \mathbf{t} = 0$$

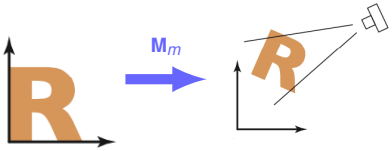
Therefore, transform normals using

$$\mathbf{N} = (\mathbf{M}^T)^{-1}.$$

We transform from model $\rightarrow$ world $\rightarrow$ camera $\rightarrow$ canonical view $\rightarrow$ screen.

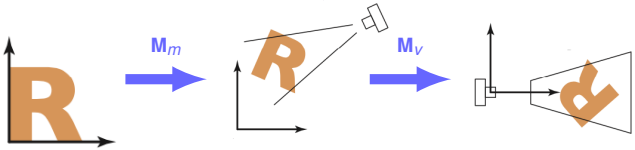
object space

We transform from model $\rightarrow$ world $\rightarrow$ camera $\rightarrow$ canonical view $\rightarrow$ screen.



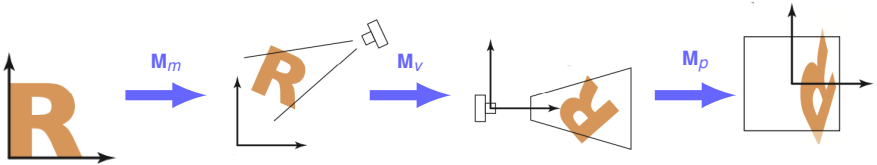
object space world space

We transform from model $\rightarrow$ world $\rightarrow$ camera $\rightarrow$ canonical view $\rightarrow$ screen.



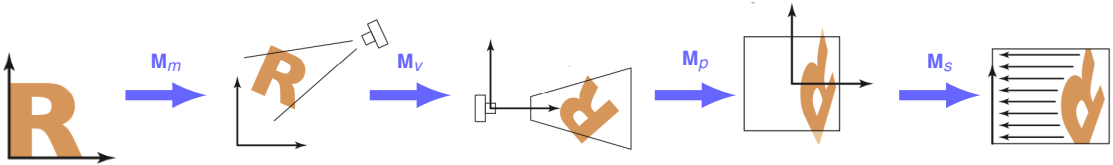
object space world space camera space

We transform from model → world → camera → canonical view → screen.



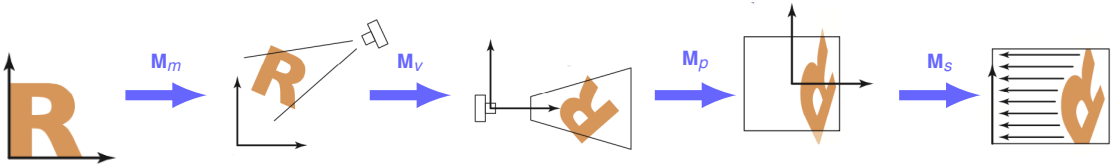
object space world space camera space canonical view volume

We transform from model → world → camera → canonical view → screen.



object space | world space | camera space | canonical view volume | screen space

We transform from model → world → camera → canonical view → screen.



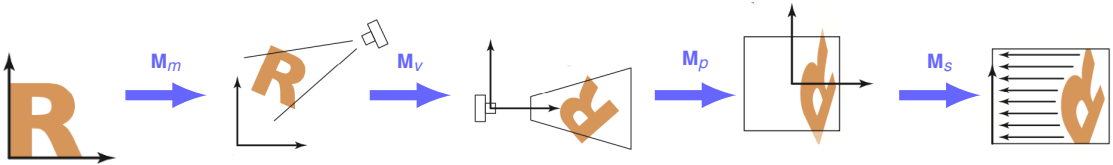
object space | world space | camera space | canonical view volume | screen space

model-view-projection matrix

$$\mathbf{M}_{mvp} = \mathbf{M}_p \mathbf{M}_v \mathbf{M}_m$$

`WebGL` does the multiplication by the viewport matrix ($\mathbf{M}_s$) for you.

We transform from model → world → camera → canonical view → screen.



object space | world space | camera space | canonical view volume | screen space

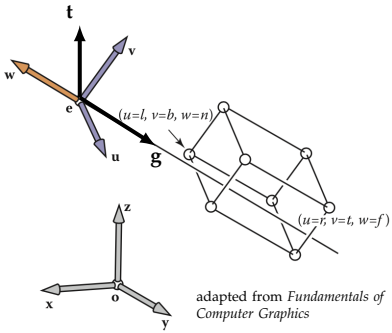
model-view-projection matrix

$$\mathbf{M}_{mvp} = \mathbf{M}_p\mathbf{M}_v\mathbf{M}_m$$

`WebGL` does the multiplication by the viewport matrix ($\mathbf{M}_s$) for you.

RIGHT TO LEFT

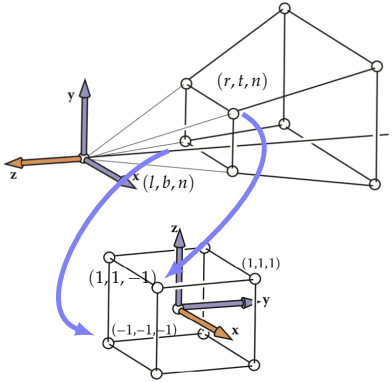
Transform your 3*d* world to camera space by a change-of-basis.



adapted from *Fundamentals of Computer Graphics*

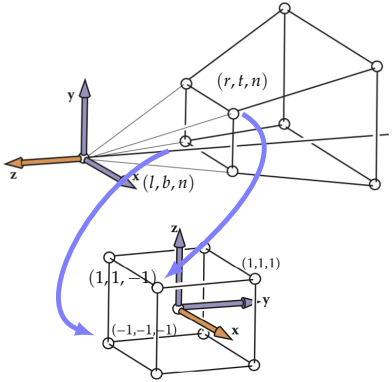
$$\mathbf{M}_V = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective projection matrix maps truncated pyramid to canonical box.



$$\mathbf{M}_{p,p} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

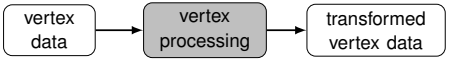
Perspective projection matrix maps truncated pyramid to canonical box.



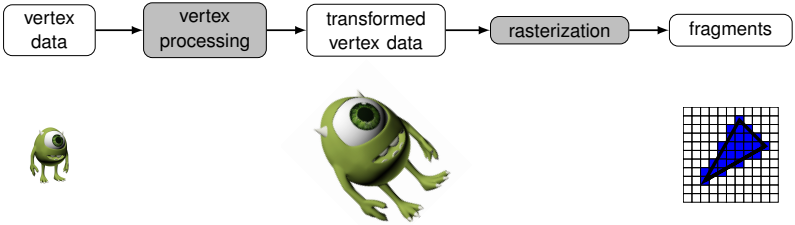
$$\mathbf{M}_{p,p} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The graphics pipeline processes vertices, then fragments.

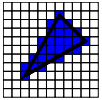


The graphics pipeline processes vertices, then fragments.



The graphics pipeline processes vertices, then fragments.

vertex data → vertex processing → transformed vertex data → rasterization → fragments → fragment processing → framebuffer



Vertices processed in a vertex shader.

written in a language called GLSL (similar to `C` with functions & types for linear algebra)

```
// here are the attributes passed to the vertex shader
attribute vec3 a_Position;
attribute vec3 a_Normal;

// here are the uniforms passed to the shader program
uniform mat4 u_ModelViewProjectionMatrix;
uniform mat4 u_NormalMatrix;

// we will pass the vertex color to the fragment shader
varying vec3 v_Color;
varying vec3 v_Normal;

void main() {
  // at the very least, you need to assign a special variable called gl_Position
  gl_Position = u_ModelViewProjectionMatrix * vec4( a_Position , 1.0 );

  v_Normal = mat3(u_NormalMatrix) * a_Normal;

  // we need to assign another variable called v_Color
  // since we plan to use it in the fragment shader
  v_Color = a_Position;
}
```

Vertices processed in a vertex shader.

written in a language called GLSL (similar to `C` with functions & types for linear algebra)

- **attributes**: input data to the vertex shader
- **uniforms**: constants used for every vertex/fragment
- **varyings**: outputs of the vertex shader to the fragment shader

must assign `gl_Position`

Fragments processed in fragment shader.

written in a language called GLSL (similar to `C` with functions & types for linear algebra)

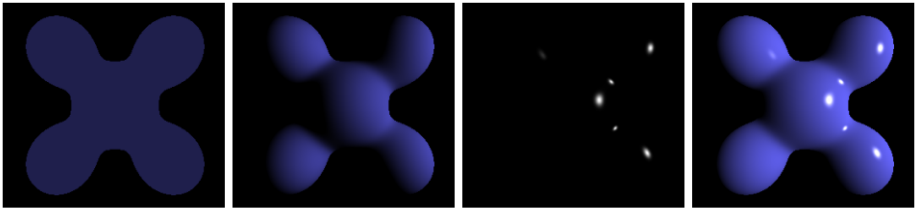
```
// the interpolated v_Color varying assigned in the vertex shader gets sent here
varying vec3 v_Color;

void main() {
  // you always need to assign gl_FragColor to color the pixel
  gl_FragColor = vec4(v_Color,1.0);
}
```

must assign `gl_FragColor`

note: `flux360` uses WebGL 2.0 so things will look a bit different.

Phong reflection model combines ambient, diffuse and specular terms.

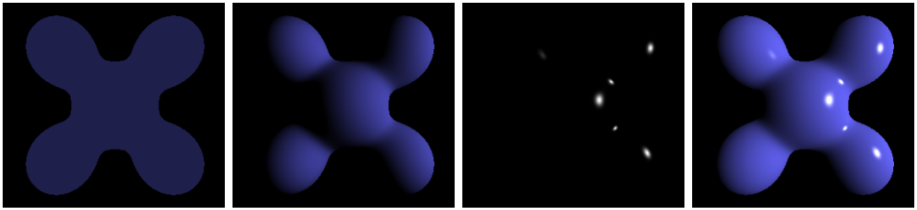
$$c = c_a + c_d + c_s$$



Ambient + Diffuse + Specular = Phong Reflection

Phong reflection model combines ambient, diffuse and specular terms.

$$c = c_a + c_d + c_s$$



Ambient + Diffuse + Specular = Phong Reflection

multiple lights?

$$c = c_a + \sum_{i \in |\text{lights}|} (c_{d,i} + c_{s,i}) = c_a + \sum_{i \in |\text{lights}|} \left(k_d L_{d,i} \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s L_{s,i} (\mathbf{r} \cdot \mathbf{n})^p \right),$$

Diffuse term can be computed using Lambert's law.

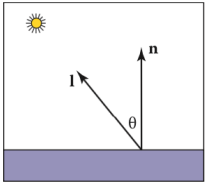
$$c_d = k_d L_d \max(0, \mathbf{n} \cdot \mathbf{l}).$$

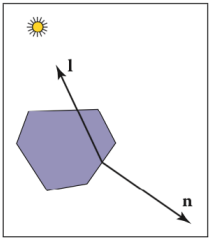
or

$$c_d = k_d L_d |\mathbf{n} \cdot \mathbf{l}|.$$

- k_d (RGB color) is diffuse reflective component of **material**
- L_d (RGB color) is diffuse component of **light**

$k_d L_d$ means a <u>component-wise</u> multiplication!



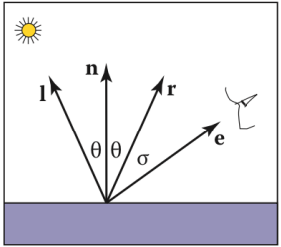


Specular term can be computed using Phong's reflection model.

$$c_s = k_s L_s (\mathbf{r} \cdot \mathbf{n})^p$$

where the direction of a perfect reflection is:

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}$$



- k_s (RGB color) is diffuse specular component of **material**
- L_s (RGB color) is specular component of **light**
- p is the **shininess**: higher p means smaller highlight

$k_s L_s$ means a <u>component-wise</u> multiplication!

Exercise: complete the fragment shader.

go to exercise in **go/cs422 -> Week 12T -> exercise** and complete the shader directly in the editor.

Here is what the vertex shader looks like:

```
attribute vec4 a_Position;
attribute vec3 a_Normal;

uniform mat4 u_ProjectionMatrix;
uniform mat4 u_ViewMatrix;
uniform mat4 u_ModelMatrix;
uniform mat4 u_NormalMatrix;

varying vec3 v_Normal;
varying vec3 v_Position;

void main() {
  gl_Position = u_ProjectionMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;
  v_Normal    = mat3( u_NormalMatrix ) * a_Normal;
  v_Position  = ( u_ViewMatrix * u_ModelMatrix * a_Position ).xyz;
}
```

useful functions: `normalize`, `dot`, `length`, `reflect`, `pow`, `max`, `abs`
extra: add silhouette + toon shading (see notes)

Rest of class time to work on Final Project.