During weeks 9–12. This includes Lectures 17–24, and workbooks 9–12.

Meshes

Lec 17B

Meshes: collections of triangles

- vertex sharing
- vertex reuse
- index set representation (for faces)

Good meshes

- consistency of handeness, consistent normals
- Avoid cracking
- avoid T–junctions

Properties

- Vertex: (barycentric interpolation) vertex colors, vertex normals
- Face: (constant over face) colors, normals

Splitting

- position is the same
- properties are on faces (different triangles)

Good triangles:

- not too small
- not to enlongated (stretched out)

Operation/ Representation

- dfficient display and storage (index set)
    - compact
    - maps well to hard ware: strips/fans, caches, format issues
- Efficient Manipulation

Fancy data structures:

- keep track of edges
- find neighbors quickly
- polygon soup
- vertex sharing/ indexed representations
- uniform patterns: grids, strips/fans
- Topology (how triangles connect) is separated from vertex info

Three

- Geometry: basic mesh, list of vertices, faces, simple JS data structures
- BufferGeometry: similar, more efficient, easy transmission to hardware, need to understand buffers first

Update geometry: geometry.elementsNeedUpdate = True

Normals

Lec 17B

Normals: right hand rule, outward facing normal vector, specify if not obvious or face directions

Specify: compute (real), fake: preface or vertex(interpolate)

Uses: backface culling (do not draw if in the back) and lighting

Lighting

Lec 17C

Lighting: real-world-like process V.S. Shading: mroe general term

local vs. global, the simple models

- Local general: photon arrives from fomem direction, where to go
  - Absorbed, perfect mirror bounce, micro-geometry, subsurface effect?
  - depens on material and direction
  - Distributions of incoming light and outgoing light (direction and amount)
  - Bi-directional reflectance distribution functions (BRDF): how light is reflected at a surface
  - percent reflected: r = B(in, out, color) where in and out are directions
- Local vs global
  - Local: at specific point, each point considered individually, assume light coms from somewhere
  - global: points interact, entire scene considered together, how light transports
- Local lighting: only consider how material reflects light, no global effects(shadows, reflections, spill/bleed)
- Simple local lighting model
  - Assume ideal material
  - make a model
  - tweak the model
  - combine with other models to mix effects

the components of the basic lighting model (ambient, emission, diffuse, specualar)

- Type of models
  - Emission: thins give off light
  - Ambient: light from nowhere
  - Specular: direct reflection
  - Diffuse: rough reflection, rough surface

Diffiuse and Specular models and equations

- Diffuse: Ideal materials: lambertian materials **p47**
  - scatters light in all directions
  - does not matter what direcion you look from
  - direction of light matter
  - Reflection:
    - $r_{diffuse} = \hat{n} \cdot \hat{I}$
    - Amount of diffuse reflection = unit surface normal X unit vector to light source
    - $color = (\hat{n} \cdot \hat{I})\ c_{light}\ c_d$ clight: color/intensity of light(RGB), cd: color of the material
- Specular: shiny things, ideal (perfect mirror) **p55**

    - light direction and eye position matter
    - eye needs to be in exact correct position
    - $\hat{e}$ eye vector and $\hat{r}$ reflection vectors
    - Imperfect mirror: phong model
        - graduate fall off as we get away from the optimal direction, keep $\hat{e} \cdot \hat{r} > 0$
        - raise to the power of shininess (phong exponent: p)
        - shinier: more like a perfect mirror: $p \to \infty$: perfect mirror, p=0: rough
        - $r_{specular} = (\hat{e} \cdot \hat{r})^p$
        - $r_{color} = (\hat{e} \cdot \hat{r})^p \; c_{light} \; c_s$ cs: color of material

Specifying surface (lighting) properties

- phong lighting model
    - $color = c_e + c_a l_a + \sum_{l \in lights}((\hat{n} \cdot \hat{l}\,) \; c_{light} \; c_d + (\hat{n} \cdot \hat{h})^p c_{light} \; c_s)$
    - emissive + ambient + diffuse+ specular
- Three.js, phong
    - color
        - color: diffuse and ambient color (main color)
        - specular: specular color
        - Emissive: emissive color
    - Shininess
        - how bright is the shininess (c_s related to c_d)
        - how focused is the shininess (p)
        - Color: plastic: white, metal is colored
    - metalness: metal is more specular, specular keeps color
    - Roughness: dull or shiny, amount of specularity, size of specularity

When to compute lighting

Use: Once **per pixel** (norm): interpolate normals

Others:

- once per triangle (flat shading)
- once per vertex and interpolate colors (gourand shading)

Texturing

Lec 18B

motivations: how to get more than 3 colors on a triangle

Bad ideas

- break into smaller pieces
- colors per pixel

Basic idea: texture mapping

- Define a coordinate system on the triangle, define a map from coordinate to color
- draw: coordinate, look up the color

steps for “standard” texturing (UV mapping with an image)

Three parts:

- assign coordinates
- loop up the value for the coordinate
- do sth with the value

Basic form: **p6**

- UV coordinates on triangles (2d)
- Images that define colors (look up color in 2d)
- use looked up color as the color for lighting (for each pixel inside triangle inside graphics hardware)
    - get the barycentric coordinate
    - use this to interpolate the UV values (get UV coordinate)
    - use the UV coordinate to loop up the color
    - use the color as the surface color

texture use, coordinate ranges

UV in [0,1]

outside: **p15**

- clamp wrapping
- repaet wrapping
- mirror repeat wrapping

Reuse: loading, processing, and storing textures is resource intensive

Steps:

- Specify UVs for triangles
- specify an image to be used for lookup
- specify how to use the image in the material
- Specify all the different parameters (lookp type, wrapping mode.....)

Lec 18C

basic textures in THREE

Create objects with UVs **p5**

- Primitives have predefined UVs
- Geometries,
    - define UVs for each vertex, face, possibility for layers, not part of faces
    - faceVertexUVs: array of layers, each layer is an array of Faces, each face is an array of UVs for each vertex
    - each vertex has a 2d coordinate, each face has. 3 vertices, each layer has n faces, each object has 1 layer or more

Load textures **p8**

- reuse if possible, share between objects
- different kinds of pre-processing happens, not just an image

Attach as colors to objects **p9**

Others: lightening and material affect color (basicmaterial if no lighting), Y is fliped by default

Lec 18D

Problems in textuere lookups:

- Magnification: a pixel covers less than a pixel in the image
    - basic Sol: nearest neighbor
    - standard sol: Bi-linear interpolation, treat the pixel as.a point, does not consider edges of pixels
- Magnification: one pixel covers an area of the texture
    - Baic idea: **Filtering,** avrerage together all the texture pixels the pixel covers
    - problem: what to average and how to average fast

Filtering

- summed area table, not practice now **p16**
    - estimate shate as a rectangle
    - pre-compute summed area table
    - fast lookups
    - idea: amortization and pre-computation: save time per-pixel by pre-computing the summed area table
- Mip-maps, modern hadware, fancier **p18**
    - approximate filter as a square
    - pre-compute multiple sizes of map
    - look up in correct sized maps
    - problem: anisotropy, the areas are not square **p20**

    - anisotropic filtering
        - Hack: use multiple squares
        - Lookup in a smaller image, half by half till 1 pixel lookup
        - looup in smallest image: bi-linear interpolation
        - Lookup in-between images: tri-linear interpolation, require 8 values (4 per layer)
        - historic storage trick
    - Summary
        - Pre-compute mip-map (images of various sizes): done with texture loading in practice
        - approximate area with square (Center is easy, size is harder): hardware per pixel
        - figure out which image to sample from: hardware per pixel
        - tri-linear interpolate between levels: hardware per pixel
    - Three options **p29**

Lec 18E

Step to load color from texture

- make some geometry
- get a picture
- get the picture in the right form
    - square, matches simple geometry, minimal lighting, lots of parts in one image
- assign UV values to vertices
    - Per face: vertex splitting
- enable texturing

Hardware does:

- UV coordinate per pixel
- texture lookup
- texture filtering (Fast: mip maps)

After color: use as the material color for lighting

Visibility

Lec 19

Process of drawing: triangles, transform (viewing), convert to pixels (raserization), color the pixel (shading)

Among them: clipping, visibility

viewing: transform from tirangles to 2d

rasterization: convert triangles to pixels

Shading (lighting, texturing): color each pixel

visibility vs. clipping vs. culling

Clipping: decide if the triangle is on the screen

- Skip triangles that are off screen
- clip to the frustum
- includes near and far
- when: happen after the triangle is transformed

Visibility: decide if another triangle blocks the pixel

- having near things block father ones
- not know until look at all triangles
- immediate mode might draw in any order
- when: depends on the also (often per-pixel after coloring and considering all triangles)
- assume the objects are solid and not-transparent, things are in view
- closer objects occlude farther ones
- Reason: make things look realistic

Culling: skipping primitives based on fast decisions

- When: early
    - discard a whole group of triangles (Triangles in another room)
    - backface culling (After the normal is transformed)
    - ...

painters algorithm, important in concept, p34

Steps

1. collect all objects
2. sort from back to front
3. draw objects in order (back to front)

Problems

- need all objects to sort, not immediate mode
- what happens with ties
    - cutting objects
- what happens with intersecting objects
- Inefficiency: resort when camera moves
    - use good data struture (BST)
- Inefficiency: draw things that get covered

Z–buffer, used in practice

Goal: order independent, **immediate mode**(1 triangle at a time)

Idea: store the depth per–pixel

Use: an extra number per–pixel, color buffer(RGB), z–buffer(Z)

General

- start with all pixels at max distance
- anything that will be drawn will be in front of the background
- why the far distance of the camera is important
- Historically: limited precision, 16 bits, avoid making far too big

Steps

1. draw pixel x,y with color c and depth z
2. frame buffer FB, Z–Buffer (ZB)
3. Old– no Z: white pixel: FB(x,y) = c
4. New – Z (read/test/write)
    1. pz = ZB(x,y)
    2. if z < px:
        1. FB(x,y) = c
        2. ZB(x,y) = z

Order independent

Problem

- Requires memory
- require read/modify/write
    - memory reads can be slow
    - need to wait until prior writes are done
- requires storing distances, (old: 16 bits, scaling issue)
- Tie
    - Decide if < or <=, order matters
- **Z–fighting**: two objects are very similar distances
    - might be a tie (drawing order matters)
    - numerical noise might break the tie
    - flipping between close objects
- Triangle partially fills a pixel
    - z–buffer is yes/no
    - does not consider partial iflling
    - Aliasing:
        - anti–alising the edges fo the objects is hard
- Overdraw efficiency

    - throw away pixel after it is computed
    - after we have computed the color
    - Wasted effort
    - overwrite is even more wasteful than Z-fail
    - Future: ways to avoid
- Semi-transparent objects
    - back object need to be drawn first
    - need to be blend with correct background
    - close object prevents objects behind from being drawn
    - Solution:
        - sort objects (painters algo)
        - transparent objects first

Summary: clear z-buffer to max distance, replace pixel write with read/test/write

Good:

- simple,
- Use memory now cheap
- generally order independent
- do not need objects ahead of time
- easy to implement in hardware

Prlbems

- sometime order matters
- can have efficiency issues
- z-fighting problems
- aliasing
- do not handle transparency

Lec 20

Interactive rendering:

- every triangle is independent
- every triangle gets lit, shaded, textured, visibility checked...
- fast, all effects possible

Graphics Pipeline, harderware

Steps

1. model objects (make triangles)
2. transform (find point and positions)
3. shade (lighting – per triganle/vertex)
4. Transform (projection)
5. rasterize (figure out pixels)
6. Shade (per pixel coloring)
7. write pixels (with z buffer)

Drawing process:

- Modeling (model transformation), shading, viewing, clipping ,projecting, visibility, rasterization

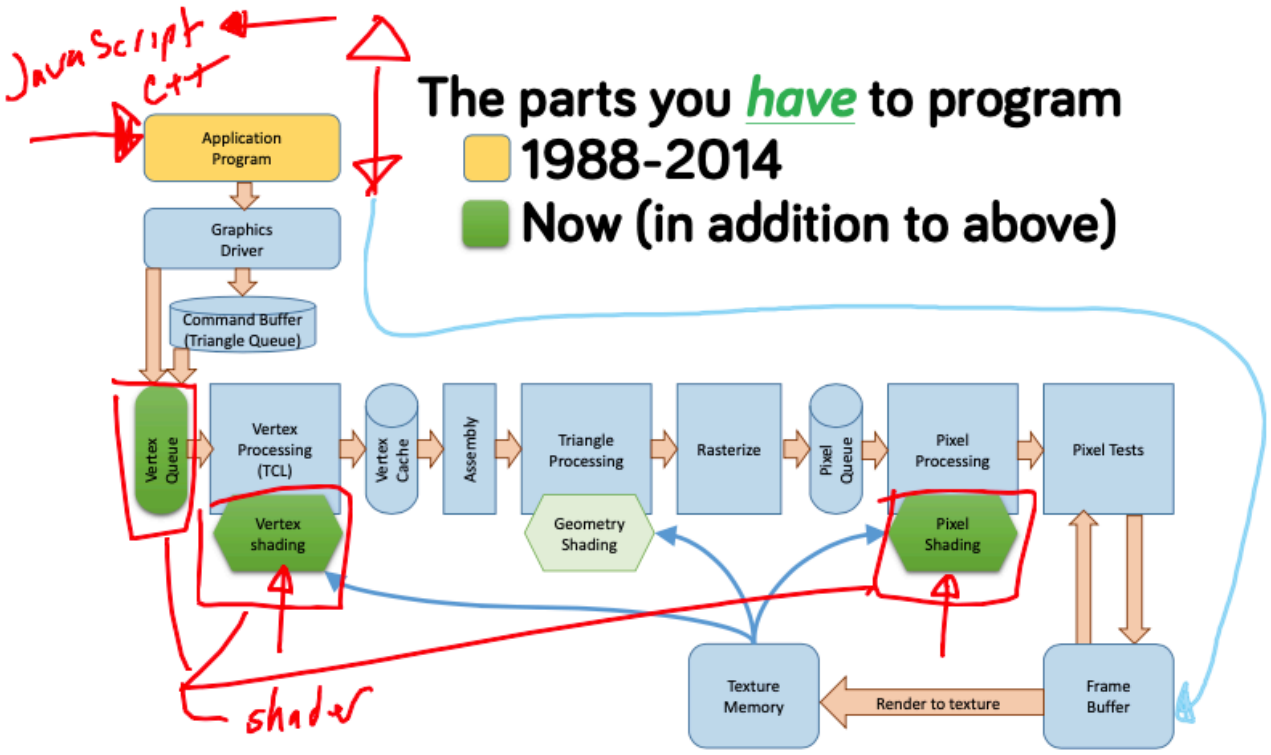
In the pipeline

- Triangles are independent
    - no pipe stall (the param in the next step is defined in the current step **p8**)
    - no complexity of handling stalls
- vertices are independent
    - parallelize, process multiple triangles, pixels at the same time
- pixels in the triangles are independent
- do not need to finish 1 before start 2

Parallelization: fast, require specific model in hardware, program interface is designed for the model

- vertex operations
    - split triangles/re-assemble
    - compute per-vertex not per triangle
- pixel(fragment) operations
    - lots of potential parallelism
    - less predictable
- use queues and caches

History: **p33**



Steps details:

- draw the triangle
  - per vertex, not per triangle
  - allows sharing vertices between triangles, or make all same to get truly flat
  - position, color, normal in vertices
- command buffer
  - split triangles into vertices
- vertex queue: buffer/queue for the vertices to proces s vertices in parallel
- vertex processing (TCl)
  - Compute transform – compute x' and n' on the screen, clip (old days), Light–compute c' per vertex
  - Add info to vertices
  - independent vertex
  - for each vertice
    - in: vertex info x, n, c + global information (current transform, lighting)
    - out: vertex info: x, n, c, x', n', c' (vertex lit color)
- Vertex cache:
  - store several processed vertices in the cache so that we can re–use if triangles share the vertex
  - Before is important to perfermance, now not
- Assembly: put triangles back together, triangles exist
- Triangle processing: in the fixed–function pipeline, not this step (maybe clipping)
- Rasterizer: convert triangles to a list of pixels
  - In: trianlge with values (x', c'...)

    - pixels with values (x', c'....)
    - figure out which pixels a primitive covers
    - turns primitives into pixels
    - interpolate vertex values to interior pixels, barycentric coordinates
        - pixel values from 3 vertices, all are interpolated
- Pixel queue: to process pixels in parallel
- pixel processing: usually color, sometimes z, not in 1988 pipeline
    - pixel in, pixel out, each independent, just change values (re-compute)
    - not change position (make another pixel)
    - change or reject other values
- Pixel tests
    - z-buffer, alpha/color/stencil-tests, color blending
- Read/write cycle from pixel tests to frame buffer
    - needed for each pixel
    - potential memory bandwidth **bottlenect** (problem)
    - failed z-test: removed by z-buffer
- Frame buffer: sent to the screen

Pixel and fragment

- Pixel: a point on the screen
- fragment: a dot on the triangle
    - migt not become a pixel (fails z-test), might only be part of a triangle

Summary:

- transfer info about vertices from application
- process vertices
- assemble/rasterize triangles
- process fragments/pixels
- Test/write pixel results

- how does this lead to the shader model and efficiency issues

Shader Basics

Lec 21

program do:

- set up for drawing
- repeat for every frame (Render)
    - clear the screen
    - for each object: draw a group of triangles

Object: a group of triangles sent together

- the vertices
- how they connect to triangles
- what shaders to use in processing them
- what parameters to use across all of them

limited in what we can change within a group

Constant state: can not change certain things while drawing a triangle, extens to group of triangles

Uniform over group of triangles: camera, frame buffer, light, texture maps....

Three.js: scene and material with common properties

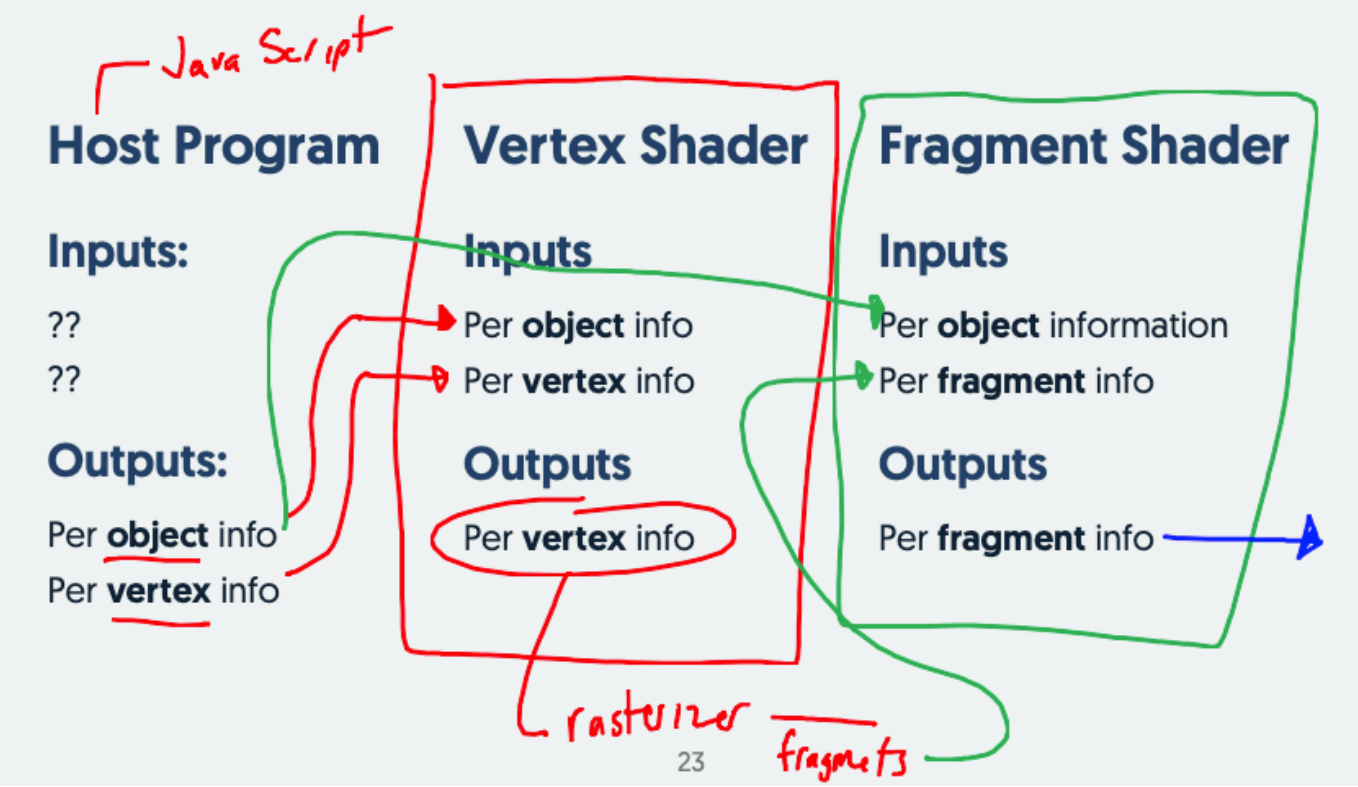
Can specify:

- global info: whole screen, window
- per triangle group info: uniforms
- per vertex info: position, color, normal...
- **NOT** per pixel

Programmable shading into the graphic pipeline

Two main parts

p23



vertex processor

- Input: vertex with info (attributes frmo the host program), output vertex with more info (more attributes)
- before: position, normal, color, UV
- Afer: position, normal, color, UV, computed: screen space position, vertex lit color, screen space normal
- Needed: screen space position for rasterizer/z-test (with other we want for coloring)
- For rasterizer:
    - use the screen coordinate to generate list of fragments
    - interpolate other values so each fragment has the properties

fragment processor

- have (depend on three vertices)
    - screen space position
    - interpolated values (attributes)
- we do:
    - each fragment is processed independently with info from rasterizer
    - figure color and depth
- Before: screen space position, depth (z) value, other interpolated properties
- After: **same** screen space position, depth value, color to write to fram buffer

Shading language GLSL

Shaders: write in shading language

- vertex shaders: compute new attributes for vertices, gl_position
- Fragment shaders: comput color/depth for fragments, gl_fragcolor

History, p26

GLSL: runtime compilation

- parts: shading language, host api (JS)
- C-like, C syntax, strict typing, operator overloading
- math data types (metrics, vectors), built-in functions
- features for tieing things together
- each shader has own program
- connect to other parts via variables, like global variables, special declarations, few built ins

variable types (how shaders communicate)

Uniform

- host triangle groups (JS)
- not changed inside the triangle group
- some built in
- JS: creating mapping of names, convert types, provide mapping table (Three do this)
- P52

Uniforms

```
{
    uniforms: {
        var1 : { value: 10.0},
        var2 : { value: new T.Vec2(1,2);}
    }
}
```

uniforms are one parameter to `ShaderMaterial`

provide a dictionary that maps **variables** to **values**

each value is a dictionary (with the key **value**)

Built-In Uniforms

Some of the uniforms are "built in"

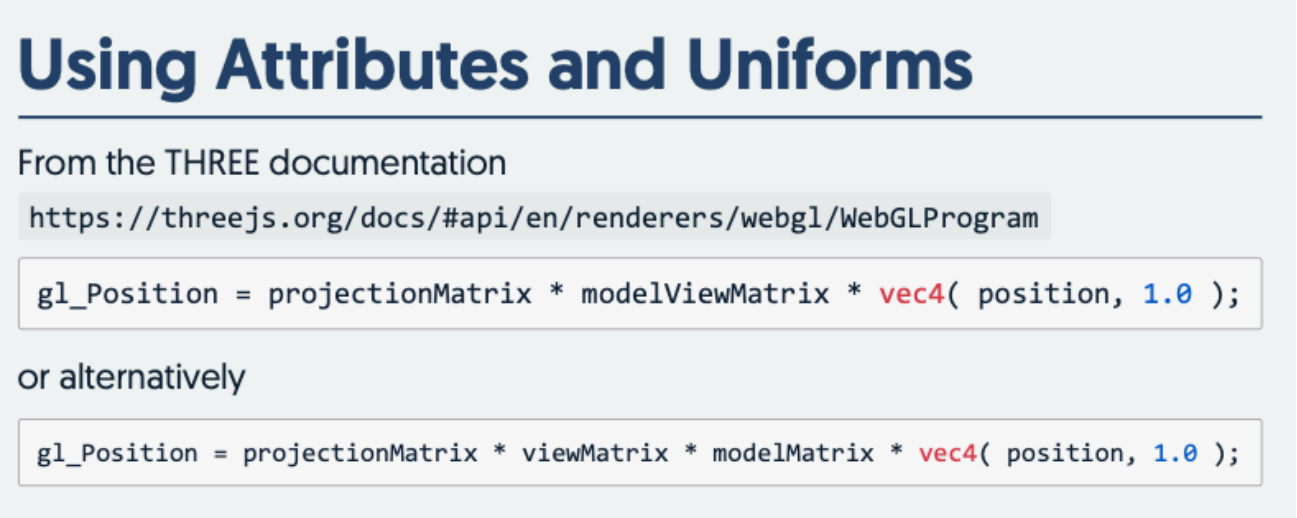
```
uniform mat4 modelMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat3 normalMatrix;
uniform vec3 cameraPosition;
```

These are added to your vertex program

Some of these are added to your fragment program

Attribute **p54**

- Per vertex input, host program
- Position, color, normal, UV
- transfer as block of memory (buffers)
- Three: position, normal, uv

- 

Varying

- per vertex output from vertex shader and input to fragment shader
- interpolate from 3 vertices
- gl_position

Three: ShaderMaterials

connection to the host program

Just strings, separate files and loaded

Steps: compile, link, use

Lec 22

lighting in shaders

Where

- vertex shader
    - compute lighting at every vertex, interpolate colors across triangle
    - per-vertex lighting, gouraud shading
    - old days
- fragment shader
    - interpolate normals and surface colors
    - compute lighting at every pixel
    - per-pixel shading
    - hardware is fast

Same function as before (lec 17), workbook 10

codes: **p40**

Shader Programming

shader idioms (using built–ins)

more efficient and convenient

Mix(a, t1, t2): a * t1 + (1–a) * t2

step, fact, sign, min, max

shader anti–aliasing, **Lec 21 p65, Lec 22 p12**

smoothstep: one **sided** only for 0–1 transition

Dfdx: derivative of function wrt x

Fwidth: derivative of functio nwrt x and y

smoothstep handles texture magnification

no texture minification

- need filtering, hard to do in a general way for procedures: over sampling

Easier to do for images

Lec 22

Noises, randomness **p24**

- pseudo–random
  - pattern too complex to see
  - still controlled/ derterministic
- structured
  - control properties that we care about
- Simple method: color = f(u), r = tract(sin(u) * 10000)
  - problem: aliasing
  - use interpolate to make smooth
- Perlin nonise
  - classic noise function, with more efficient GPUS
  - Noise–controlled psuedo–randomness
  - coherence at different frequencies

displacement map shading, p44

Fake shape with vertex shader, displace points

Use dots to move things in the normal position

- only move vertices, plane the same
- did not change lighting
- spacing depends on mapping
- some triangles get really stretched
- do change the shape

Bad:

- depends on meshing, need small triangles
- messes up the mesh
- need to get things aligned with vertex samples
- may want to make more triangles

Texturing Tricks

fake normals for fake lighting, p49

Flat

- make many triangles
- fake it with texture: smooth with lighting

Not in the right shape, just make it looks right with shape

Lec 24

bump and normal maps

p11

Compare to displacement maps

- no change to geomotry, no shadows, occlusions, vertex or fragment
- both just change normals but in different ways

Normal maps

- texture lookup of normal vector, RGB –> xyz directions, middle value (!28/256) = 0
- need to renormalize
- vector relative to real normal
- deal with xyz of normals

Bump maps

- deal with the height of changes in the normal

Good

- easy to specify surface details
- does not change shapes
- basic lighting effects
- Works iwth lighting
- easy in three

Bad

- not change side view
- not cause occlusions
- not for big effects
- No shadows
- need reflection

skyboxes, p21

Far away: position not matter, orientations do

a box for background,

always styas centered at the camerra, move with the camera

help reflections:

- assume objects being looked at are far away
- view direction matters
- point position doesn't

environment mapping

cube or sphere (around object)

position does not matter, normal matters

assume point is at center of sphere/cube

Lookup direction is based on viewing direction

Require:

- small object/far environment
- eye is far (direction amtter)
- position do not matter
- small curvy objects good, large flat bad

spherical and cubic representaitons for maps

sphere

- sampling issues at pole,
- single images
- capture in 1 photograph

Cube

- sampling issues in corners
- Images are human viewable
- maps nicely to graphics hardware

layered textures

The environment map is light, real scene to create brightness, not limited to small set of light sources

Often states (pre-compute ahead of time)

Dynamic environment map **p34** multi-pass rendering

- draw the scene once or multiple times, remember info
- draw the scene again with the info
- both steps when world changes, step 1 first
- store info in texturs
- no need to draw everything in step1

light maps

Scalling problem **p40**

- Book matching, flipped Y, repeat wrapping

Light maps: put lighting into the texture **p48**

Good:

- can be fancy
- pre-computed

Bad

- must be known ahead
- can not change when camera moves, object moves

Special case

- self shadowing: importing for conveying shape, pretend light comes from all directions (ambient)
- amount each point is visible
- corners and crevices are dark

Ambient Occlusion shading (all directions for objects, blocks, light)

- pre-compute for all points with special tools or clever hacks
- like a light-map, ambient occlusion map

shadow maps p52

non-local

only one way to do shadows

Idea: can light see the object? -> put "camera" from light source

Step

1. take a picture from light position
    - camera at light source
    - what objects are visible to the light (others are shadow)
    - use this picture as the shadow map
2. draw the regular picure
    - for each pixel on an object
    - see if pixel is visible in shadow map

Test: check the pixel is the same pixel in the shadow map

- check color (no)
- Common: check depth
    - Problem: small errors will lead to larger
    - shadow map is a z-buffer (depth map)

Resolution

size matters

- Spotlight (Can be small)
- directional light source (Area)
- point light? (needs to be a cube/sphere)