Introduction

January 11, 2021 8:49 AM

Graphics pipeline

OpenGL/WebGL:

- A software interface that allows a programmer to communicate with the graphics hardware
- A programming interface for rendering 2D and 3D graphics
- A cross-language multi-platform API for computer graphics

OpenGL (Open Graphics Library):

- Open industry-standard API for hardware accelerated graphics drawing
- Implemented by graphics-card vendors

OpenGL ES: embedded systems version of OpenGL with reduced functions

WebGL makes OpenGL accessible from JavaScript, same underlying graphics architecture

- WebGL 1.0 is based on OpenGL ES 2.0, now supported in almost all browsers
- WebGL 2.0 is based on OpenGL ES 3.0

OpenGL pipeline

- Shapes are discretized into primitives (triangles, line segments formed by vertices)
- Vertex Shader
  - Vertices stored in a vertex buffer
  - When a draw call is issued, each of the vertices passes through the vertex shader
  - On input to the vertex shader, each vertex has associated attributes
  - On output, each vertex has a value for gl_Position and for its varying variables
- Rasterization
  - Data in gl_Position are used to place the three vertices of the triangle on a virtual screen
  - The rasterizer figures out which pixels are inside the triangle and (linearly) interpolates the varying variables from the vertices to each of these pixels
  - It always pick 3 vertices
- Fragment shader
  - Each pixel is passed through the fragment shader, computes the final color of the pixel
  - The pixel is then placed in the framebuffer for display
  - By changing the fragment shader, we can simulate light reflecting off of different kinds of materials

Three.js

- High level library that can use WebGL
- Implements scene and mesh abstractions
- Mesh = geometry + material properties
- Scene contains a hierarchy of mesh objects
- Render a scene using a camera

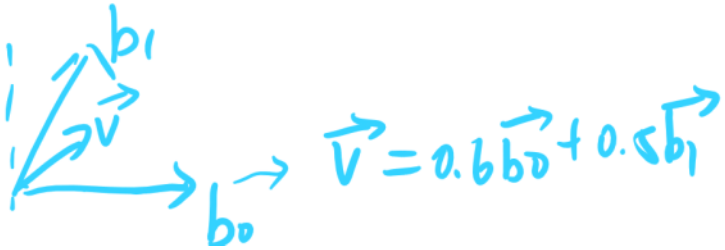
Geometry

January 18, 2021 9:46 AM

Points and vectors

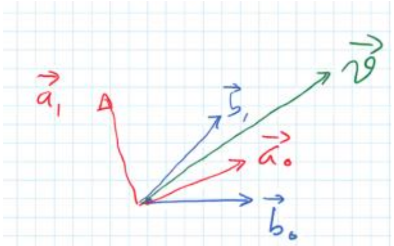
- Point: a real object position in space
    - Origin
- Vector: an algebraic object in space that is associated with operations

Basis and coordinates

- Basis: an independent set of vectors that can produce any vectors in the space by linear combination
    - 

    - The size of the basis is the same as the dimension of the space
- Coordinates of a vector in a basis $\vec{b}$:
    - If $v = v_0 b_0 + v_1 b_1$, where v_0 , v_1 are scalars and b_0, b_1 are vectors we can represent $v = \begin{pmatrix} v_0 \\ v_1 \end{pmatrix}$ by an array of numbers (column matrix)
    - If we pick different basis, v is the same, but the coordinates will change
    - Notation: $\overline{v} = \begin{pmatrix} a \\ b \end{pmatrix}$(column vector), $\underline{v} = (a, b)$ (row vector)
        - i.e. We can write the basis as $\underline{b} = (b_0, b_1)$
        - Then $v = \underline{b}\begin{pmatrix} v_0 \\ v_1 \end{pmatrix}$ as a product of matrices

Linear (4×4) transformations and affine spaces

- Change of basis, matrix representation
    - 

      Given $v = \underline{b}v_b$, want to find $v = \underline{a}v_a$ (Note: v is always the same)

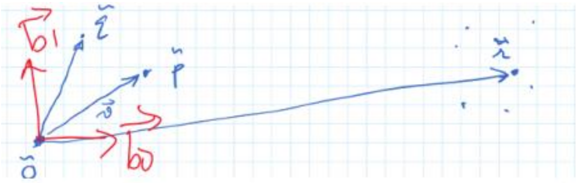
      We can write b_0 in terms of basis $\underline{a}$ by $b_0 = \left(a_0\ a_1\right)\begin{pmatrix} L_{00} \\ L_{10} \end{pmatrix}$, where $\begin{pmatrix} L_{00} \\ L_{10} \end{pmatrix}$ is the coordinates of b_0 with respect to $\underline{a}$

      Similar for b_1, so we have $\left(b_0\ b_1\right) = \underline{b} = \underline{a}\begin{pmatrix} L_{00} & L_{01} \\ L_{10} & L_{11} \end{pmatrix}$

      Let $L = \begin{pmatrix} L_{00} & L_{01} \\ L_{10} & L_{11} \end{pmatrix}$

      Then $v = \underline{b}v_b = \underline{a}\begin{pmatrix} L_{00} & L_{01} \\ L_{10} & L_{11} \end{pmatrix}v_b = \underline{a}Lv_b$

      So $v_a = Lv_b$
- Representing points in affine spaces



- We can add a vector to a point to produce a new point
  $p = o + v = o + b_0 v_0 + b_1 v_1 = \begin{pmatrix} b_0 & b_1 & o \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ 1 \end{pmatrix}$
  - 1 indicates origin
    - If it is 0, it represents the vector in the frame ($v = \begin{pmatrix} b_0 & b_1 & o \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ 0 \end{pmatrix}$)
  - $\begin{pmatrix} b_0 & b_1 & o \end{pmatrix}$ is called the (coordinate) frame
- In 3D space, we thus have vector 4
- We call these coordinates with one extra number "homogeneous" coordinates, since we can represent both points and vectors in the same way

Frames

January 22, 2021 10:05 AM

Notations:

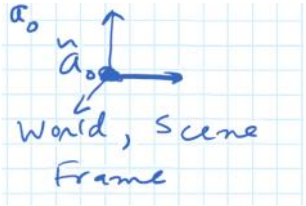
	ours	Textbook
Points	$\tilde{p}$	$\tilde{p}$
Vectors	$\vec{v}$	$\vec{v}$
Column matrix	$\overline{v}$	$\boldsymbol{v}$
Row matrix	$\underline{v}$	$\boldsymbol{v}^{\boldsymbol{T}}$
Basis	$\underline{\vec{b}}$	$\vec{\boldsymbol{b}}^T$
Matrices	$\underline{\overline{A}}$ or A	

Coordinate frame

- $\underline{\tilde{b}} = (b_0\ b_1\ o)$

Homogenous transformation matrices

- Model and view transformation

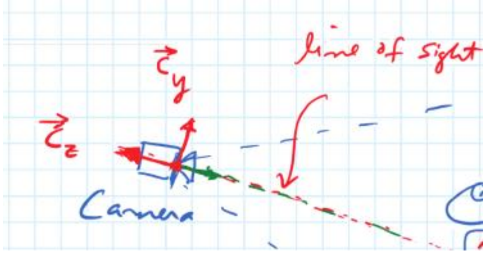
  

  $\underline{\tilde{a}} = (\vec{a_1}\ \vec{a_2}\ \vec{a_3}\ \widetilde{a_o})$

  $\underline{\tilde{b}} = \left(\vec{b_1}\ \vec{b_2}\ \vec{b_3}\ \widetilde{b_o}\right)$

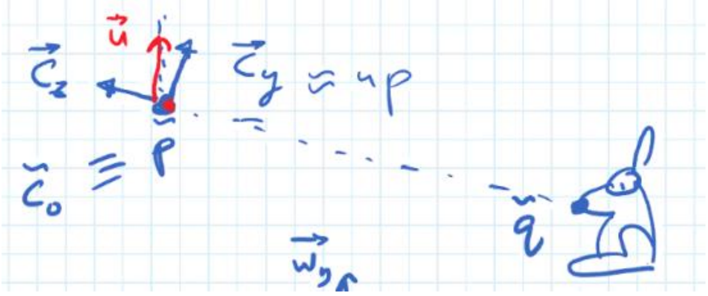
  Want to convert a point in frame b to a,
  the point on the ball is the same in any frame, only coordinates are different
  $\tilde{p} = \underline{\tilde{b}}\overline{p_b} = \underline{\tilde{a}}\overline{p_a}$
  Since $\underline{\tilde{b}} = \left(\vec{b_1}\ \vec{b_2}\ \vec{b_3}\ \widetilde{b_o}\right) = \underline{\tilde{a}}\underline{\overline{B}}$
  We have $\underline{\overline{B}}\overline{\mathrm{p_b}} = \overline{p_a}$
  Here $\underline{\overline{B}}$ is the model matrix
  If we are transforming the world position to camera position, we apply the view matrix

Frames

- World frame $\underline{\tilde{w}}$
  - Scene/stage
- Model/object frame $\underline{\tilde{b}}$
  - Object3D class in Three.js
- Camera/eye frame $\underline{\tilde{c}}$
  - 

  - $\vec{c_z}$ is in the opposite direction of the line of sight
    - We look at the scene along $-\vec{c_z}$
  - $\vec{c_x}$ is pointing towards us (out of page)

Frame transformation matrices (4×4 homogenous transformation matrices)

- Model matrix M
    - Model to world frame
    - $M = \begin{pmatrix} & & & p_x \\ & & & p_y \\ & & & p_z \\ & & & 1 \end{pmatrix}$
        - Last column is the coordinate of the model's origin $\tilde{p} = \left(p_x, p_y, p_z\right)$ in world frame
- View matrix (Camera matrix) C
    - World frame to camera frame
    - For VR, we will have two cameras and two view matrices
    - $C = \left(normalize(\vec{u} \times \overrightarrow{c_z}) \quad \overrightarrow{c_z} \times \overrightarrow{c_x} \quad \text{normalize}(\overline{q} - \overline{p}) \quad \overline{p}\right)$
        - 

        - The last column is always the world coordinate of the origin of the camera $\overline{p}$
        - If we have a point $\tilde{q}$ we want to look at from $\overline{p}$(camera matrix), we can use normalize($\overline{q} - \overline{p}$) for its z value
    - Look at matrix in Three.js $V = C^{-1}$

If $a = wA$

- $A = (v_1 \quad v_2 \quad v_3)$
- Where v_1 shows how we get a_x by w_x and w_y
- v_2 shows how we get a_y by w_x and w_y
- v_3 shows how we get a_0 (origin of frame a) by translating w_0 (origin of frame w)
-

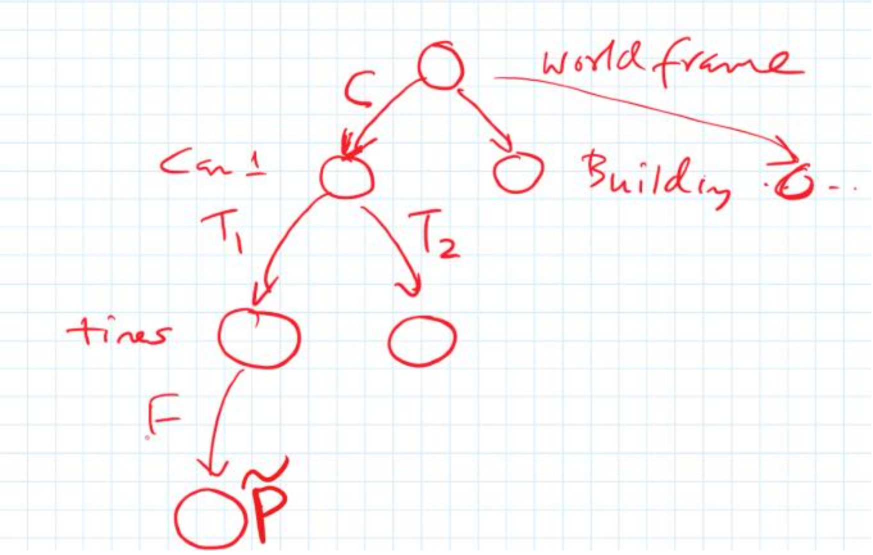
Scene graphs

February 1, 2021 9:58 AM

Suppose we have a model defined in the world frame ($\underline{\tilde{w}}$) and $\underline{\tilde{a}} = \underline{\tilde{w}}A$
Given a point $\tilde{p}$ in world frame and we want to apply a transformation R

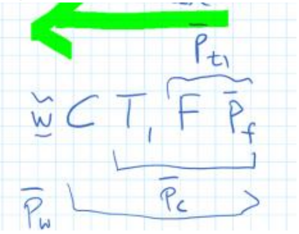
- $R\tilde{p}$ rotates $\tilde{p}$ about the world origin
- If we want to rotate $\tilde{p}$ about a, we need to:
  - Convert coordinates to $\underline{\tilde{a}}$
    - $\tilde{p} = \underline{\tilde{w}}p = \underline{\tilde{a}}A^{-1}p$
  - Apply transform in $\underline{\tilde{a}}$
    - $\underline{\tilde{a}}RA^{-1}p$
  - Convert back to $\underline{\tilde{w}}$
    - $\underline{\tilde{a}} = \underline{\tilde{w}}A$
  - This gives the final position $\tilde{p}' = \underline{\tilde{w}}ARA^{-1}p$
- $R' = ARA^{-1}$ is the rotation transform
  - Similarity transformation

To transform a point in a structure, we need to apply transform matrix layer by layer to get its position in the world
A scene graph is a data structure containing hierarchical transformations



- The world position of $\tilde{p}$ is $\tilde{p} = CT_1Fp$
- In Three.js
  - Child to parent $A.matrix$
  - Child to world $A.matrixworld = CT_1$
- Animations meshes create a skeleton ≈ scene graph
  - Bone is object 3D

Interpreting chains of transformation

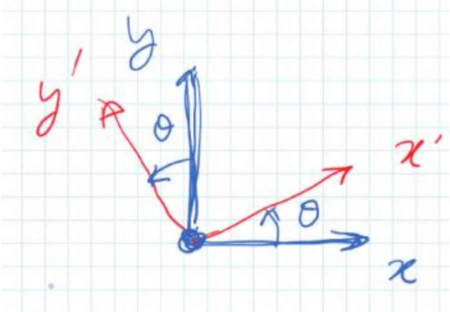
- From right to left: transformation of coordinates from one frame to another
  - 

- From left to right: moving points in a frame

- $\underbrace{\underbrace{\underline{\tilde{w}}\, C}_{\underline{\tilde{c}}}\, T_1}_{\underline{\tilde{t}_1}}\, F\, \bar{P}_f$

Types of transformations

- Translation: $T = \begin{pmatrix} 1 & 0 & 0 & p_1 \\ 0 & 1 & 0 & p_2 \\ 0 & 0 & 1 & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Rotation: $R = \begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix}$ where A is a 3×3 orthogonal matrix
    - Let $u = Rv$, then $|u| = |v|$, isometry
    - $R^T R = I$, (R is an orthogonal matrix)
    - If $R = (R_x\ R_y\ R_z)$, then R_x, R_y, R_z are orthonormal vector
    - Always has $\det R = 1$
    - Can construct any rotation as the product of the three matrices
        - Rotation about the z-axis: $R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$



        - Rotation about the x-axis (the 4th row and column are 0001): $R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$
        - Rotation about the y-axis: $R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$
        - 3 angles are sufficient to represent any rotation
    - s
- Scaling: $S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Reflection: $R_1 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
    - If R_2 is some other full rotation matrix,$R_1 R_2$ is also orthogonal
    - Determinant is -1
- If calculating from left to right, we are manipulating the coordinate
    - E.g. $a = wABC$, where A is translation, B flips the x, y coordinates, and C is rotation
      We first translate the coordinate w by A, then flips the coordinate by B, and rotate by C, all vertices changes accordingly

Aliasing

- Scene made up of black and white triangles, and jaggles around the edges
- Problem: too much information in one pixel

Over-sampling

Multi-sampling

- Render to a high resolution color and z-buffer
- During the rasterization of each triangle, coverage and z-values are computed at this sample level
- For efficiency, the fragment shader is only called only once per final resolution pixel
- Once rasterization is complete, groups of these high resolution samples are averaged together

Animation

February 8, 2021 10:09 AM

Modeling: mesh in rest pose/bind pose
Skeleton(bone): rig, armature
Binding: bind skeleton onto the skin

- Determine which part of the skin/mesh bonds to/moves with the bone
- rigid binding: divide mesh into portions and bind

Geometric skinning

- Rigid skinning: large extorsion around the binding point
- Smooth skinning: add weights to the skinning
- Skinning designate the deformation algorithm: how the mesh/skin is linked to the skeleton(rig)

Rigging: FK vs IK

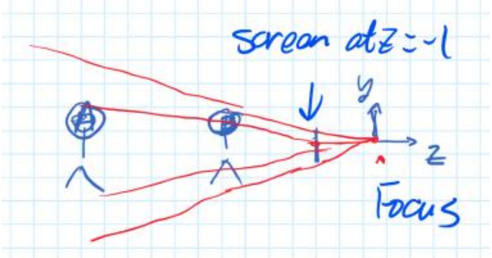
- Forward Kinematic (FK): specify transformation matrix at every joint
- Inverse Kinematic (IK): position a handle at some point, the system computes transformation at other joints

Key framing: using timeline, set key frame at different points of time.

Cameras and projections

February 10, 2021 10:21 AM

For a graphic camera:

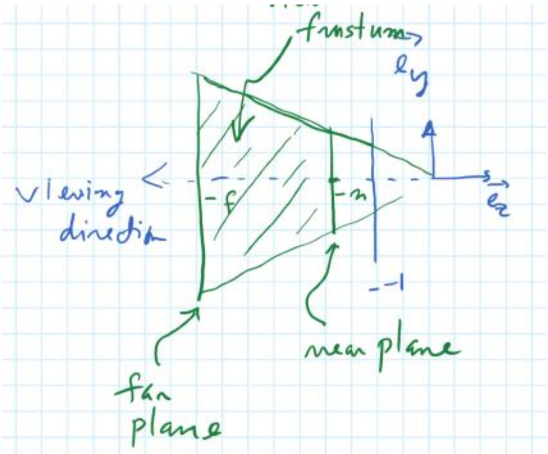
- 

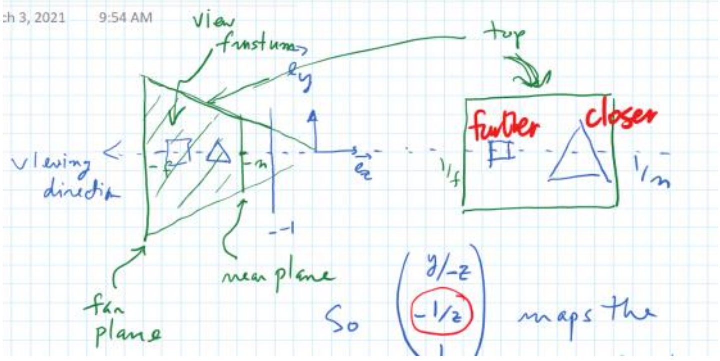
- Focus is at the camera position
- Screen is at camera local space: $z = -1$
- The perspective shortening can be achieved using similar triangles

In homogeneous coordinates

- The point $(y, z, 1)$ is projected to $\left(-\frac{y}{z}, -1, 1\right)$, so that we have $z = -1$
  - This is not a linear combination of y and z, thus, we have to use homogeneous coordinates
- Assume that if p are homogeneous coordinates of a point, wp is equivalent to p
  - $(y, z, 1) = (2y, 2z, 2) = (wy, wz, w)$
  - Can always recover the canonical form by dividing by the last entry
  - So $\left(-\frac{y}{z}, -1, 1\right) = (-y, -z, z)$
- So the projection can be written as
  - $\begin{pmatrix} y \\ z \\ -z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} y \\ z \\ 1 \end{pmatrix}$
    - $P_0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{pmatrix}$ is the projection matrix
    - Left side is called the clip coordinates
    - Major flaw: P_0 is singular, don't know what is in the front or back (losing information)
  - in practice, we use $\begin{pmatrix} y \\ 1 \\ -z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} y \\ z \\ 1 \end{pmatrix}$
    - Then divide by $-z$, we have $\begin{pmatrix} -y/z \\ -1/z \\ 1 \end{pmatrix}$, $1/z$ is more useful than having -1 (depth like)
      - Actually, $\frac{1}{z}$ is $\frac{1}{depth}$

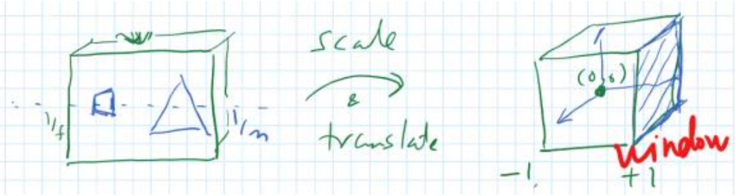
Projective Transformations

- 

- Using the mapping $\begin{pmatrix} -y/z \\ -1/z \\ 1 \end{pmatrix}$, everything in the near plane is mapped into $z = \frac{1}{n}$ and everything in the far plane is mapped into $z = \frac{1}{f}$.
    - It maps the view frustum into a box
    - 

- We want to scale and translator the mapped box into a normalized box (i.e. centered at (0,0,0) with side length = 1)
    - This is called the normalized device coordinates (NDC)
- Projection preserves co-linearity and co-planarity of points

Normalized device coordinates to window coordinates and depth

- Map the (x, y) coordinates to the window
- 

- Rasterizer produces vertices in the NDC, we then map the vertices to the window coordinates
    - In window coordinate, $(0,0)$ actually is a region $[-0.5,0.5] \times [-0.5, 0.5]$(bottom left corner)
    - Need a transform that maps the lower left corner to $[-0.5,0.5]$ and the upper right corner to $[W-0.5, H-0.5]$
    - Transformation is done by the viewport matrix
        - $\begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{W}{2} & 0 & 0 & \frac{W-1}{2} \\ 0 & \frac{H}{2} & 0 & \frac{H-1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \\ z_n \\ 1 \end{pmatrix}$.
        - Problem: too close, the object will blow up, lose precision when far away
- Depth values stored in a depth buffer/access

Depth

- Visibility

    - Opaque objects block light and we need to model this computationally
    - Can store everything hit along a ray and then compute the asset
        - Make sense in ray tracing (one pixel per ray)
        - But in GLSL, we are using fragment shading
- Z-buffer (depth buffer)
    - Triangles are drawn in any order
    - Each pixel in frame buffer stores depth value of closest geometry observed so far
    - When a new triangle tries to set the color of a pixel, we first compare its depth to the value stored in the z-buffer
        - Depth comparison, if z_e is the coordinate in eye frame, we compare $z_n = -\frac{1}{z_e}$
    - If observed is closer, replace the value in the buffer
    - Done per-pixel, no cycle problem
    - There are optimizations, where z-testing is done before the fragment shading is done
- Other uses of visibility
    - Generate shadows
    - Speed up the rendering process
        - If we know that some object is occluded from the camera, then we don't have to render the object in the first place

Rendering

February 12, 2021 10:11 AM

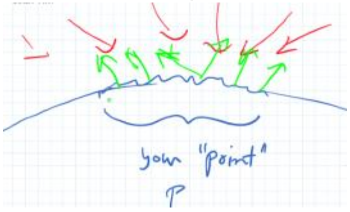
Modeling material appearance

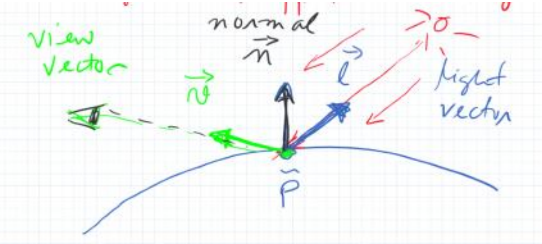
- Rich variety of materials, characterized by surface reflectance and scattering

Shading and shadow

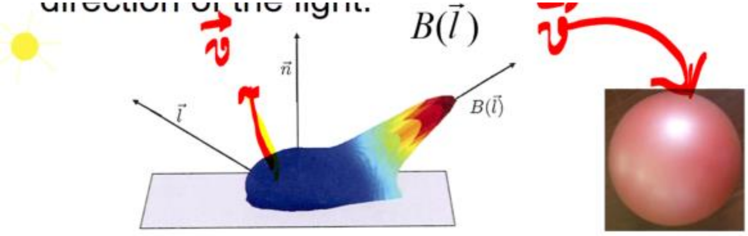
- They are variation of color/appearance over the surface
- Shadow: appearance due to occlusion from another object on a different part of the same object
- For shading, several options:
    - Gouraud shading: compute shading at a vertex to determine vertex color, then interpolate the color to fragments
    - Phong shading: interpolate the normal to the fragments, and do the shading per fragment
        - Current state of the art

Reflection models

- Global illumination: light could arrive from all direction
    - 

    - May also have subsurface scattering
- Simplify light as arriving from a distant light source, approximated as rays
    - 

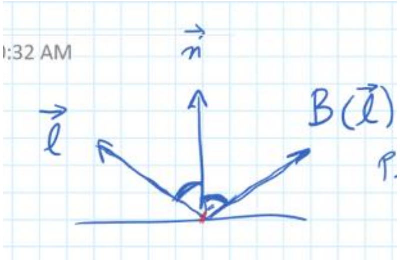
Light blob from PVC plastic

- Plastic will appear brightest when observed in the directions clustered about the bounce direction of the light
    - 

        - Left side is called diffuse lobe, right side is called specular lobe
- Bidirectional Reflectance Distribution Function (BRDF)
    - Models need BRDF as two lobes
    - Diffuse: lambertion
        - Light reflected is the same for all view vectors v, $I_d = kl \cdot n$
        - 

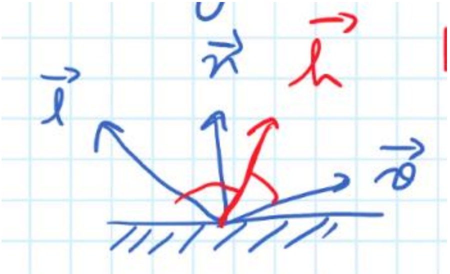
        - Specular

Phong shading:

- Ambient
    - A constant color value
    - A crude hack to capture
    - Global illumination
- Diffuse:
    - Follows Lambert's law of perfectly rough surface
    - Light reflected is the same for all view vectors v, $I_d = k_d l \cdot n$
- Specular = shiny
    - Capture highlights
- 

    - B is the bound vector/perfect mirror like direction
        - It is a simple ellipsoidal approximation to the specular lobe observed in real materials
        - Intensity $I_s = k_s\,(B \cdot v)^\sigma$
        - $B = -l + 2(l \cdot n)n$
        - σ is the shinniness (specular exponent)

Blinn-Phong shading:

- 

- h is the halfway vector, use $h \cdot n$ instead of $B \cdot v$

Normal

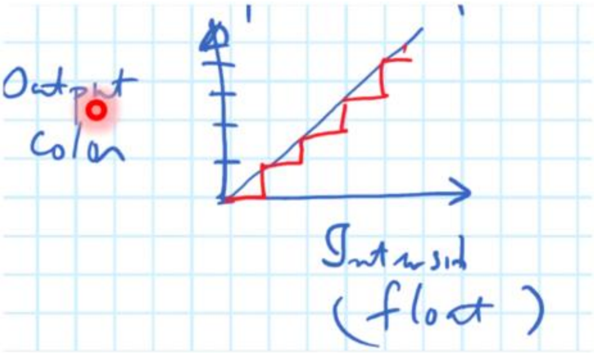
- If the transform is $T = \begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix}$, it does not correctly transform the normal
    - Although it transforms the tangent vector well
- Define n as $n \cdot\ t = 0 = n^T t$ in coordinates
    - If $t_a = Tt$ (T is the transformation matrix), $t = T^{-1} t_a$, want n_a to satisfy $n_a^T t_a = 0$
    - This give $n_a^T T t = 0$ which means $n_a^T T = T^T n_a = n$
    - So $n_a = \left(T^T\right)^{-1} n$, where $\left(T^T\right)^{-1} = \begin{pmatrix} A^{-T} & 0 \\ 0 & 1 \end{pmatrix}$
    - Normal matrix do the work

Heidrich-Seidel model

- Surface with thin fiber on groove like feature
- The microgeometry of a fiber has a lot of potential normal
    - Pick n' to be the projection of l perpendicular to t (tangent vector)
        - $n' = l - (l \cdot t)t.$

Basic Toon Shading

- Small palette of colors

  - 

- Draw silhouette edges ($n \cdot v \approx 0$)

Gooch shading

- Darker areas more visible
- Diffuse=dot(light, normal)
- Calculate cool and warm Gooch colors
  - $k_{cool} = cool_{color} + alpha{*}k_d.$
  - $k_{warm} = warm_{color} + beta{*}k_d.$
- Calculate the final color (mixing/blending the cool and warm colors)

Texturing

March 5, 2021 9:59 AM

Normal mapping

- R, G, B values from a texture are interpreted as the three coordinates of the normal at the point
- Can be used as part of some material simulation

Environment cube maps

- Used to model the environment in the distance around the object being rendered
- Use 6 square textures representing the faces of a large cube surrounding the scene

Projector texture mapping

- Glue texture onto triangles using a projector model instead of the affine gluing model
- Simulate a slide projector illuminating some triangles in space

Shadow mapping

- First create and store a z-buffered image from the point of view of the light
- Compare what we see in our view to what the light see in its view

Texture mapping

- An efficient way to model surface detail using discrete (sampled) data
- Coordinates: parameterization of surfaces
- Images: sampled representations of continuous functions

Texture coordinates:

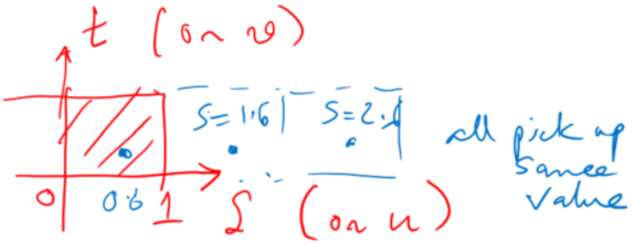
- Map to a flat parameter space $p = f(t)$
    - p is in 3D
    - Can use linear function $p = ft$
        - Can think $\begin{pmatrix} a \\ b \\ c \\ 1 \end{pmatrix} = f\begin{pmatrix} longitude \\ latitude \end{pmatrix}$
- More generally:
    - Sample at a few points
    - Linearly interpolate between the samples (rasterization)
    - We can also interpolate texture coordinates
        - If we know the texture coordinates of each vertex, we will know the texture coordinates of each fragment
        - Look up the color, normal, etc. of the fragment in the texture image

Steps for texture mapping

- Create a texture object and load texels (texture pixel) into it
- Include texture coordinates with the vertices
- Associate a texture sampler with each texture map used in shader
- Retrieve texel (texture pixel) values

To use a small texture image to cover a large object

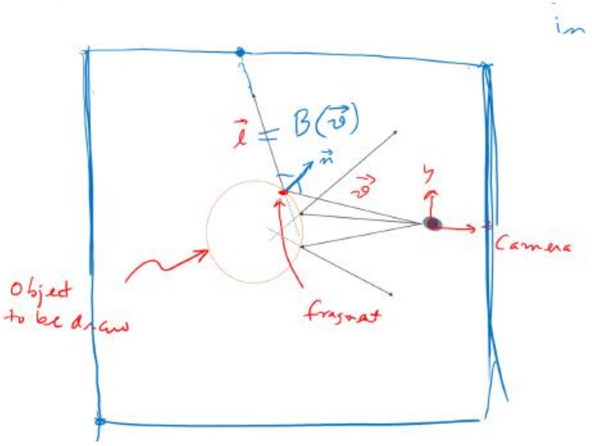
- Repeat wrapping is useful for tiling a large area with the same small texture

- 

Generating texture coordinates

- Can be done in 3D modeling software
- In production, coordinates are designed with model
- Projection, environment maps coordinates can be computed in shaders

Cube mapping

- 

- samplerCube is a special GLSL function that takes a direction vector and returns the color stored at this direction in the cube texture map

Shadow mapping

- First pass: create shadow map, a z-buffer image from the point of view of the light
- Second pass: check if fragment is visible to the light using shadow map

Multi sampling: once per final resolution pixel
Super sampling: per original pixel

Coverage

- Rapid changes in color du to
    - Texture
        - Pre-filtered textures, mip mapping
    - Shading
        - Generally changes slowly, except at edges of triangles
    - Depth discontinuities
        - Check if discontinuity passes through pixel
- Super sampling deals with all at once, but at great cost
- Maybe efficient to handle each one separately
- Estimate partial coverage of pixel by triangle fragment
- Fraction of pixel covered is called alpha
- Coverage function
    - $C = 1$ at any point where the image is occupied
    - $C = 0$ where it is not

Compositing

- Happens after rendering
- Generalize idea of anti-aliasing to representing the coverage of each pixel by an object
- Essential for multi-pass rendering, requiring combination of images
- Simple image compositing

  - Given two discrete images, a foreground I^f, and a background I^g
  - Use foreground pixel if defined, otherwise use back ground pixel
  - May lead to large aliasing
- Alpha blending
  - Over operation (pre multiplied alpha)
    - Composite image color $I^C = I^f + I^b(1-\alpha^f)$.
      - The amount of observed background color at a pixel is proportional to the transparency of the foreground layer at that pixel
    - Composite alpha $\alpha^C = \alpha^f + \alpha^b\left(1-\alpha^f\right)$.
    - Note: the operation is associative but not commutative
      - $I^a\ over\ (I^b\ over\ I^c) = (I^a\ over\ I^b)\ over\ I^c$.
      - $I^a\ over\ I^b \neq\ I^b\ over\ I^a$.
  - Non pre multiplied alpha
    - Composite image color $I^C = \alpha^f I^f + I^b(1-\alpha^f)$.
      - The amount of observed background color at a pixel is proportional to the transparency of the foreground layer at that pixel
    - Composite alpha $\alpha^C = \alpha^f + \alpha^b\left(1-\alpha^f\right)$.

Reconstruction

- Given a discrete image, create a continuous image (get a texture colors that fall in between texture pixels)
- Constant interpolation: sample a pixel and hold until we get the next pixel
- Can also use linear or higher order interpolation
- In 2D, use bilinear interpolation
  - Interpolate in x, then interpolate in y (though horizontal/vertical ordering does not matter)
  - At integer coordinates, we have continuous=discrete, in between, blended continuously
  - Each texel influences a 2-by-2 region

Mip mapping

- Starts with an original texture and then creates a series of lower and lower resolution texture
- Each successive texture is twice as blurry
- Trilinear interpolation: uses 8 pixels to blend

Interpolation

- Sampling (continuous to discrete) and reconstruction (discrete to continuous) bridges continuous and discrete functions
- Examples
  - Fonts
  - Car bodies
  - Meshes
  - Audio
  - Computer animation using keyframes
- Linear interpolation: straight lines between points
  - $C(t) = C_0(1-t) + C_1 t$, separate the data from the model, easy to generalize to higher dimension
- Polynomial interpolation
  - A degree n polynomial can pass through $n+1$ points
  - Will have twisting between linear points
- Splines: piece-wise polynomials of low degree
  - Usually degree 2 or 3
  - Key is to match derivatives at the joints
- Blending functions
  - Linear: $\Sigma_{\mathrm{i}=0}^{1} c_i b_i(t)$, where $b_0(t) = 1-t, b_1(t) = t$.
  - Quadratic: $\Sigma_{i=0}^{2} c_i b_i(t)$, where $b_0 = (1-t)^2$, $b_1 = 2t(1-t)$, $b_2 = t^2$.
    - We can check that $b_0 + b_1 + b_2 = 1$

- Bernstein polynomials
    - Degree 0: $b_0 = 1$
    - Degree 1: $b_0 = 1 - t, b_1 = t$
    - Degree 2: $b_0 = (1-t)^2$, $b_1 = 2t(1-t)$, $b_2 = t^2$.
    - Degree n: $b_i, n = \binom{n}{i} t^i (1-t)^{n-i}$
    - $\Sigma b_i = 1$.

Lighting

April 21, 2021 5:33 PM

Lighting equation is evaluated

- For surface reflection (specular)
- Subsurface reflection (diffuse)
    - Dielectrics only
- Possibly other surface interfaces like clear-coats

Incoming radiance accounts for

- Local lights
- Indirect light bounced from surfaces

Local illumination

- Lighting calculation done without knowledge of objects in the scene (does not depend on the geometry of the scene)
- Assumes only computing lighting from light sources
    - Light sources
    - Environment maps
    - Irradiance environment mapping

Global illumination

- Lighting techniques taking into account objects and geometry within the scene

Light probes

- Storage method
- A cloud of points with lighting data is stored within the scene
    - When sampling, neighboring points are interpolated between
- Often used with irradiance maps to render diffuse global illumination
- Points may be a regular mesh or an irregular cloud of points connected to form a tetrahedralization (P11)

Ambient occlusion

- Not all areas on a mesh can obtain a full hemisphere of incoming light
- Simulate the darkening in areas of occlusion
- HBAO
    - For each pixel, compute the normal from depth info, determine the view angle
    - For pixels in the hemisphere facing towards the view direction, sample depth to determine if light rays would be occluded (Monte Carlo)

Shadows

- Achieved for free if using ray tracing
- One of the most costly passes in a traditional rendering pipeline
    - Re-render the entire scene multiple times
    - Render objects not visible on screen, because they can cast shadows
    - Tests visibility and culling within engine
- Hard, soft shadows, contact hardening
    - Light sources are not usually punctual
        - Light comes from multiple slightly different angles from the same light
        - Causes shadows to be slightly blurred the further you are from a light source
        - Contact hardening
    - Traditional punctual lights cause hard shadows
        - With extra cost, effects can be added to generate soft/contact hardening shadows
- Shadow maps

    - Depth image of the scene from the view of the light
    - If camera depth (in light space) is greater than the shadow map, the light does not contribute
- Shadow atlases
    - When computing final lighting of the scene all shadow maps are needed
    - Store all shadow maps in a small number textures or atlases
    - Render to a portion of the atlases can be achieved by modifying the viewport

Transparency

- Global illumination technique (requires knowledge of other objects in the scene)
- In practice, it requires a forward lighting pass of transparent objects ordered by depth after opaque pass

Reflection

- Implemented with a cube map
- Ray-tracing
    - Single bounce path tracing can give a good reflection when paired with temporal anti-aliasing

Clustered lighting

- Pre-computed pass to gather a list of relevant lights for each cluster
    - Limit each cluster to a finite number of lights
    - Cache lights to a cluster cell texture
- At lighting time, lookup the cluster for the cluster the rendered point is inside
- Common optimization for forward lighting, but can also be applied to deferred lighting