What I learned in CSE 167x: Computer Graphics

Owen Jow

Last updated August 13, 2018

1 Unit 0

In this unit, I got to review some of the freshman-year math relevant to computer graphics, in particular the dot product $\mathbf{a}^T\mathbf{b}$, which can be used to compute the angle between two vectors, and the cross product $\mathbf{a} \times \mathbf{b}$, which can be used to compute a vector orthogonal to its inputs.

The dot product can also be used for projection. When $\mathbf{a}$ is a unit vector, the projection of $\mathbf{b}$ onto $\mathbf{a}$ is merely $(\mathbf{b} \cdot \mathbf{a})\,\mathbf{a}$. When $\mathbf{a}$ is not a unit vector, we have to divide by $\mathbf{a} \cdot \mathbf{a}$ as well.

I also clarified the difference between rasterization and ray tracing: rasterization goes from geometric primitives to pixels, while ray tracing goes from pixels to geometric primitives.

2 Unit 1

In the second unit, I first reviewed some of the most common linear transforms (scaling, shearing, and rotation). Because these transforms can be represented as matrices, they inherit the properties of matrices. For example, if applying the transform $\mathbf{A}$ to a point $\mathbf{x} + \mathbf{y}$, one can either compute $\mathbf{A}(\mathbf{x} + \mathbf{y})$ or $\mathbf{A}\mathbf{x} + \mathbf{A}\mathbf{y}$.

Next the unit went more into depth about rotations, particularly 3D rotations. I learned that rotation matrices $\mathbf{R}$ are necessarily orthogonal (i.e. their columns and rows are orthonormal vectors, and $\mathbf{R}^{-1} = \mathbf{R}^T$). Ravi also made clear the fact that the rows of $\mathbf{R}$ can be seen as the $\mathbf{u}, \mathbf{v}, \mathbf{w}$ axes of the destination (rotated) coordinate frame, and that rotating a vector is equivalent to considering the sum of its projections onto each of these $\mathbf{u}, \mathbf{v}, \mathbf{w}$ axes.

To my jubilation, we then derived Rodrigues' rotation formula (the matrix form of an axis-angle rotation). The formula involves adding the projections of the rotated vector onto each of the three axes (one given, two inferred). One component doesn't change during rotation (the projection onto the given axis); the other two are based on the vector's projected rotation in the plane orthogonal to the given axis. When visualizing this, I always think of the axis being the up vector, the vector-to-rotate pointing up-right, and then the rotation being into the page. (This is how Ravi drew it in lecture.)

Later, I learned how surface normals transform, assuming it is known how *points on the surface* transform. If the transformation for points on the surface is $\mathbf{M}$, then the transformation for surface normals is $(\mathbf{M}^{-1})^T$, meaning if $\mathbf{M}$ were just a rotation, then the surface normals would transform in the same way as points on the surface. But usually this will not be the case.

`gluLookAt` setup: the camera is at a certain position with a certain orientation. The `gluLookAt` transform can be seen as translating everything in the world s.t. the origin of the coordinate frame is at the camera, then rotating everything in the world s.t. the coordinate frame aligns with the given viewing direction and up vector. At the end of it all, the camera will be at the origin, looking down the negative z-axis.

- The `gluLookAt` matrix is applied to points in world coordinates, and is analogous to the extrinsic matrix (it puts things into camera/eye coordinates).
- Note that the given camera viewing direction and up vector might not be orthogonal, and the viewing direction takes precedence – i.e. we will always use the viewing direction first, and then just take the component of the up vector perpendicular to that.

Once points are in eye coordinates, we need to project them onto 2D as normalized device coordinates. To do so, we can use *orthographic projection* (transform a cuboid viewing region to the $[-1, 1]$ cube) or *perspective projection* (transform a frustum viewing region to the $[-1, 1]$ cube).

- In orthographic projection, parallel lines are preserved and further objects don't look smaller.
- In perspective projection, parallel lines converge and further objects look smaller.
- The $[-1, 1]$ cube represents normalized device coordinates, and contains z-buffer values in addition to standard x and y. The z-values are quantized, i.e. everything within a certain bucket will be drawn at the same depth.

The perspective projection matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \quad or \quad \begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It can be scaled by $-d$ because of homogeneous coordinates (we always divide out by the final coordinate, so such scaling doesn't matter). The perspective projection matrix is analogous to the intrinsic matrix, a difference being that the one shown here doesn't have the same complexity.

The frustum near and far planes have dimensions controlled by the `fovy` and aspect ratio parameters. The scale of the far plane w.r.t. the near plane is a byproduct of the field of view configuration, which describes *how large of an angle* the camera is seeing.

Thanks to the homework I know that GLM/OpenGL's matrices are column-major and will hopefully not forget this for a while.

3 Unit 2

3.1 Graphics Pipeline

The first thing I took away from this unit was a fuller understanding of the graphics pipeline. In the simplified pipeline from lecture, the vertex shader first takes vertices from *object coordinates* to *camera coordinates* to *clip coordinates* – optionally modifying/defining geometry and other vertex properties along the way. Next, the rasterizer associates the projected shapes on the screen with pixels and interpolates vertex properties over the fragments between the vertices. Finally, for each fragment, the fragment shader sees the interpolated values and outputs a color/depth, upon which we can perform depth testing and update the front buffer.

Here's a summary:

1. *Vertex shader*: vertex processing, with MVP transform (clipping, perspective division, and the viewport transform to follow; clipping/perspective division is what maps the space to $[-1, 1]$). The vertex shader operates on vertices, does processing, and gives outputs corresponding to vertices.

2. *Rasterization*: interpolation of vertex properties across pixels between vertices (automatically handled by hardware). Also called "scan conversion," because you scan over the pixels line by line and assign values to them via interpolation within and across lines.

3. *Fragment shader*: takes interpolated values at pixels, operates on these, and outputs color and depth for each pixel. After this step, depth testing (hidden surface removal) and buffer-writing (usually to back buffer first, as per double buffering) can be performed.

Clarifications:

- **Fragments.** A fragment is a part of some geometry that was associated with a pixel; there can be multiple or no fragments for a pixel region. After rasterizing a primitive, it is turned into a bunch of fragments, one for every pixel (or sample portion of a pixel) that it covers. From [2]: "*In computer graphics, a fragment is the data necessary to generate a single pixel's worth of a drawing primitive in the framebuffer.*" For example, we would need the depth of each fragment in order to perform depth testing; one triangle's fragment might be in front of another triangle's fragment at the same pixel. In general, fragments from multiple primitives can contribute to a single pixel.

- **Shaders.** A shader is a simple program that runs on the GPU in parallel on all *vertices* or *fragments* or *whatever* and computes properties for those things (technically it can compute whatever it wants). "Fragment shader" is synonymous with "pixel shader," despite fragments and pixels not being the same.

3.2 OpenGL

OpenGL has some special conventions. For example, OpenGL's camera frame looks down the negative z-axis with the y-axis pointing up, while the standard vision camera frame looks down the positive z-axis with the y-axis pointing down [to go from the former to the latter: after applying the extrinsic matrix, apply a 180-degree rotation around the x-axis (which flips the y- and z-axes)]. For projection, points are centered around zero, so if you want your points in the ranges $[0, w]$ and $[0, h]$ instead of $[-w/2, w/2]$ and $[-h/2, h/2]$, you should translate (x, y) by $(w/2, h/2)$.

Clarifications:

- **Camera coordinates.** "Camera coordinates" ("in the camera frame") means that points are in a coordinate frame at the position of the camera (e.g. the pinhole) and, in OpenGL, with the z-axis pointing in the direction of the physical image plane.

3.3 Lighting/Shading

Finally, the unit helped me refresh some lighting and shading concepts that I had long forgotten. First of all, two styles of shading:

- **Gouraud shading**, which computes lighting colors *in the vertex shader* based on the vertex normals; these vertex colors are then interpolated bilinearly at pixels during rasterization.
- **Phong shading**, which computes lighting colors *in the fragment shader* rather than in the vertex shader. In the vertex shader, we just compute the normals. Then, during rasterization, we interpolate the normals instead of the colors. The fragment shader uses the interpolated normal at each pixel for lighting computations. (Phong shading cannot happen in the vertex shader because the normals are not interpolated until rasterization.)

During local shading, we compute the light reflected by each surface toward the camera, which depends on the surface normal, the direction to the viewer, the direction to each light, and the material properties of the lights and objects. Each light has a color, and the objects also reflect certain colors parameterized by *ambient*, *diffuse*, and *specular* values. Objects also have material properties such as *shininess.* The light and material properties are combined through a lighting model (e.g. Blinn-Phong) to compute the color that the viewer will ultimately see for each fragment.

The Blinn-Phong model, assuming constant attenuation, is

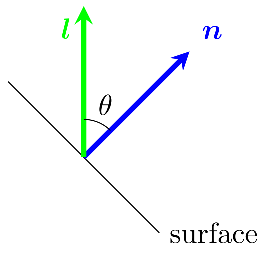
$$k_a L_a + \sum_i L_i [k_d \max(\mathbf{n} \cdot \mathbf{l}, 0) + k_s \max(\mathbf{n} \cdot \mathbf{h}, 0)^s]$$

for

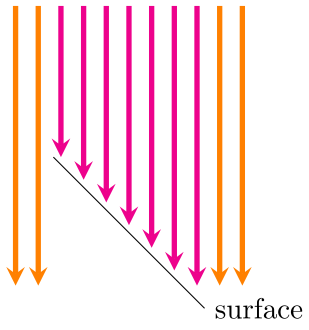
- k_a the ambient reflection color,
- L_a the ambient light color,
- i an index for each light,
- L_i the color of light i,
- k_d the diffuse reflection color,
- k_s the specular reflection color,
- $\mathbf{n}$ the surface normal,
- $\mathbf{l}$ the direction **to** the light,
- $\mathbf{h}$ the half vector between $\mathbf{l}$ and the viewing direction, and
- s the shininess.

The dot products are cosines, since the $\mathbf{n}$, $\mathbf{l}$, and $\mathbf{h}$ are taken to be unit vectors. They represent *Lambert's cosine law* at work, which says that the amount of light falling on a surface is proportional to the cosine of the angle θ between the light direction $\mathbf{l}$ and the surface normal $\mathbf{n}$. We assume that a "beam" or bundle of light rays are falling on a surface in the $-\mathbf{l}$ direction.

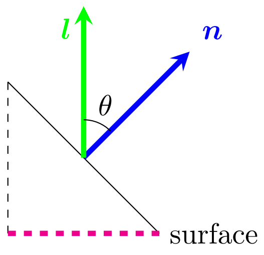
This is related to foreshortening (in which a line appears shorter as the angle between the line and the line of sight decreases).



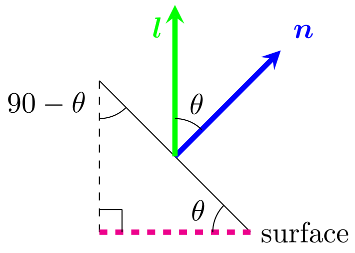
Setup.



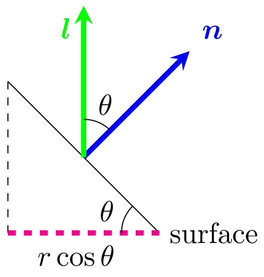
Light falls on the surface in the $-\mathbf{l}$ direction. The surface is tilted, so the orange rays will not hit it.



The dotted magenta line represents the component of the surface available for the light to strike.



The bottom-right angle is θ. We can see this by shifting $\mathbf{l}$ and $\mathbf{n}$ up and to the left, s.t. $\mathbf{l}$ aligns with the left edge of the triangle. Then the upper-left angle is $180-\theta-90=90-\theta$, and the rest follows easily from there.



If the length of the tilted surface is r, then the length of the relevant magenta line (the component able to receive light) is $r\cos\theta$. **And thus it is proportional to** $\cos\theta$ **as Lambert's cosine law says**.

4 Unit 3

This unit felt the most familiar to me. It covered basic recursive ray tracing, where

- a primary ray is cast through each pixel on the screen; then from the nearest intersection point (if there is one) a shadow ray is sent to each light and, if visible, the light's illumination contribution is made
- this ray-casting process is repeated recursively if the intersected material is reflective/refractive

In basic ray tracing, there are only point lights, meaning by default there are only hard shadows. We must cheat to get area lights/soft shadows, e.g. by representing area lights as grids of point lights.

Note that *ray tracing*, as a rendering method, is in contrast to *rasterization*, the previous rendering method. In ray tracing, we go pixel by pixel and intersect rays with the 3D scene ("for each pixel, for each object"). In rasterization, we go object by object and compute the colors of all pixels covered by each object ("for each object, for each pixel in the object").

- Much of Whitted ray tracing can also be implemented within a rasterization framework, but it'll often be difficult, forced, or unintuitive (and will provide no efficiency benefit whatsoever).
- It's difficult to produce many realistic effects such as shadows and refraction with rasterization, whereas ray tracing has no trouble (indeed it is natural for ray tracing to do these things).
- Unlike rasterization, explicit scene geometry (i.e. vertices) need not be defined for ray tracing. We just have to be able to run an intersection routine with each object.

Ray tracing is also in contrast to *path tracing* (not covered in 167x). In path tracing, rays act like light in the real world; after making an initial intersection, a ray continues to bounce around the scene until it hits a light or is (randomly) terminated. So now the areas of the lights matter. This produces more realistic images than ray tracing does, but takes longer and involves more noise.

- Ray tracing only has direct illumination.
- Path tracing introduces indirect (global) illumination: light bouncing off of other objects.

5 Acknowledgments

This document was inspired by Yining Liu's "what I learned" writeups. The CSE 167x course was run by Ravi Ramamoorthi on edX. A large thank-you to everyone involved.

6 Resources

[1] A Stack Overflow post to help understand `gluLookAt` (camera position & orientation handling)

[2] A Stack Overflow post to help understand fragments