Niall Williams | CSC 361 - Computer Graphics | Notes

August 21, 2018 (homework notes)

Chapter 1

Computer graphics are everywhere. Everything you see on your screens is generated using discoveries from the field of computer graphics. Movies, video games, images, visualizations, simulations, and design apps, etc.

History of Computer Graphics

It began with Ivan Sutherland's Sketchpad program as part of his Ph.D. It was a program that allowed you to draw lines and create basic shapes. Ivan is now considered the **father of computer graphics**. He also invented the first HMD.

After this breakthrough came a lot of further improvements and additions to CG: raster algorithms, implementation of parametric surfaces, hidden-surface algorithms, and the representation of points by homogeneous coordinates (projective geometry in 3D graphics). The next decade came with the *z*-buffer for hidden surface removal, texture mapping, Phong's lighting model, keyframe-based animation, and ray tracers. The Utah Teapot was invented by Martin Newton and came out in 1975. The creation of SIGGRAPH signified that CG was a well established subfield of computer science.

Flight simulator is known as the "killer app". First VR app.

At the same time, hardware advances became more common and races to build the best hardware were common.

In the 90's, 3D graphics began to replace 2D graphics as hardware got to the point that it could support real-time 3D rendering. At the same time, the use of 3D effects in movies became pervasive. Toy Story was the first fully computer generated movie. Quake was the first full 3D game. OpenGL came out in 1992. This was huge because it turned CG into a high level process, which meant everyday programmers could get into CG easily.

Overview of a Graphics System

A graphics system can be split into 3 steps:

Input → Processing → Output

Graphics systems can be either **non-interactive** or **interactive**. Clicking a youtube video is a non-interactive system: You don't do anything after the click. A program like Photoshop is interactive, since the output (what you draw) changes in real-time in response to your inputs.

Common input devices are:

- Keyboard
- Mouse (pointing device): invented in 1963/4 at Stanford (Xerox Park)
- Touchpad (pointing device)
- Pointing stick (pointing device)
- Trackball
- Spaceball
- Tablet (digitizing device)

- Haptic device
- Joystick
- Wheel
- Gamepad
- Camera
- Touchscreen
- Data gloves

Common output devices are:

- CRT monitor - Contains an array of pixels, known as the **raster**, on which pixels are colored and displayed. Raster algorithms select and color the pixels to generate an image. A memory location called the **color buffer** (in the CPU or graphics card) typically contains 32 bits of data per raster pixel -- 8 bits for each of RGB, and 8 for the alpha value.
- LCD monitor - Each pixel contains three subpixels made of liquid crystal molecules, which separately filter lights of the primary colors.
- Laptop and mobile device displays - Typically use TFT-LCD display technology.
- 3D display - There are two kinds of 3D displays. I don't think their implementation matters. If they are discussed in class, read page 18 of the book.

List of everyday encounters with CG:

- Snapchat filters
- Anything generated on a display (videos, text, video games)
- Weather channel effects
- Graphs
- ATM displays
- VR devices

NPR - field of graphics studying making 2D cartoon images, non photorealistic
Video called "40 year old 3D computer graphics"

August 22, 2018 (class notes)

Major areas of CG

- Modeling: develop mathematical representation of objects.
- Rendering: Process of generating a 2D image from a 3D model
- Animation: Create illusion of motion through sequences of images
- User interaction
- Virtual reality
- Visualization
- Image processing
    - Commonly used on assembly lines. If you want to assemble a box of chocolates, you need to place the pieces in the correctly shaped slots. Same applies to other manufacturing

- 3D scanning
- Computational photography
    - Google maps car. Take videos and reconstruct the 3D shape of the environment that was captured in the 2D video
- Entertainment
    - Video games, cartoons, visual effects, animated films
- Design
    - CAD/CAM
- Training
    - Simulation
- Information
    - Medical imaging
    - Information visualization
    - Ex: Look at brain to diagnose and catch diseases early (Parkinson's)

Jim Blinn is famous for the Blinn-Newell method, and he modified the Phong reflection model. **He invented the lighting model that is the default in OpenGL.**

August 23, 2018 (homework notes)

OpenGL (Chapter 2)

The **object space**, or **world space**, is the virtual area in which OpenGL renders the objects we tell it to.

The OpenGL window has its coordinate system origin *(0, 0, z)* located in the **bottom left corner** of the window. The x-axis extends horizontally to the right, and the y-axis extends vertically upwards.

Orthographic projection, viewing box and world coordinates

The projection statement in OpenGL is `glOrtho(`*left, right, bottom, top, near, far*`)`, which specifies a viewing box with corners corresponding to these points in the glOrtho() call. **Note that the *near* and *far* values are flipped in sign.**

The rendering process in OpenGL has two steps:

1) **Shoot:** Objects are *projected perpendicularly* onto the front face of the viewing box (the face on the *z = -near* plane). The front face of the viewing box is called the **viewing face** and the plane on which it lies is the **viewing plane**.
2) **Print:** The viewing face is *proportionately scaled* to fit the rectangular OpenGL window. Thus, the final image generated is dependent on the aspect ratio of the OpenGL window.

The size and location of the rendering in each coordinate direction are independent of how the axes are calibrated, but determined rather by the *ratio* of the original object's size to that of the viewing box in that direction.

Note that the coordinate system in OpenGL is **rectangular** (3 axes are mutually perpendicular) and **it forms a right-handed system**.
The OpenGL window is called the **screen space**.

Always set the parameters of `glOrtho()` so that *left < right, bottom < top, and near < far.* The aspect ratio of the viewing box should be set same as that of the OpenGL window or the scene will be distorted by the print step.
The perpendicular projection onto the viewing plane in `glOrtho()` is called orthographic projection, orthogonal projection, or parallel projection.

The OpenGL window can be placed in a certain place on the computer monitor that the programmer specifies. The coordinates of the computer monitor are set up so that the origin is located in the **top left corner**.

August 25, 2018 (homework notes)

2.4 - Clipping

Clipping is what happens when objects are not visible in the rendered scene. Objects drawn outside of the viewing box are **clipped** so that they are not seen. Clipping is a stage in the graphics pipeline.
The viewing box has six faces. And OpenGL clips the scene off at the edge of these faces that lie on six different planes. Thus, the planes are called **clipping planes**. We have the ability to define the clipping planes as well as the viewing box planes.
OpenGL draws polygons after triangulating them as so-called **triangle fans** with the **first vertex of the polygon as the center of the fan**. This can change the render depending on which vertices are moved relative to the center of the fan. See exercise 2.9.

2.5 - Color, OpenGL State Machine, and Interpolation

The `glColor3f(`*red, green, blue*`)` call specifies the **foreground color**, or **drawing color**, which just refers to the color applied to objects being drawn. Each component's value ranges from 0.0 to 1.0 and determines the component's intensity.
The color values are **clamped** to the range [0, 1]. So, if a parameter is set outside this range, it is automatically set to the nearer of the two range limits.

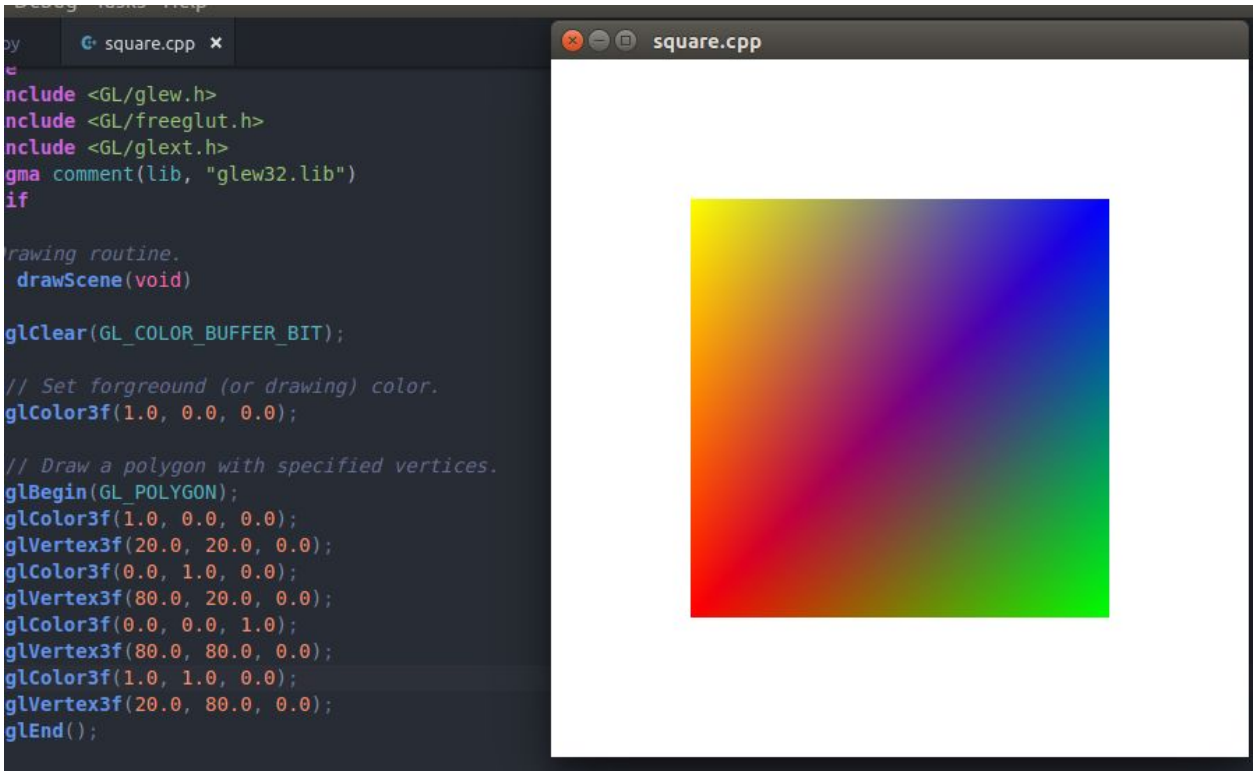
The `glClearColor(r, g, b, a)` call specifies the **background color**, or **clearing color**. The fourth parameter is the **alpha** value, which specifies transparency.
The statement `glClear(GL_COLOR-BUFFER_BIT)` clears the window to the specified background color, meaning that every pixel in the color buffer is set to that color.

Foreground color is one of a collection of variables, called **state variables**, which determine the state of OpenGL. Some other state variables are **point size, line width, line stipple, material properties**. OpenGL is often called a **state machine** because it remains in its current state until a declaration is made that changes a state variable.

If you draw a polygon with a color and then change the color and draw a new polygon, it is said that the first color **binds** to the first polygon (actually, it binds to its four specified vertices), and the second color to the second polygon. The bound values specify the color **attribute** of the polygons. The values of those state variables that determine how an object is rendered collectively form a primitive's attribute set.

Note that OpenGL draws in **code order**, which is simply the order in which code is written.

The colors are bound to the vertices, which means you can create a multi-colored polygon by binding different colors to different vertices:



The different color values bound to the four vertices of the above square are **interpolated** over the rest of the square. This is most often the case with OpenGL: **numerical attribute values specified at the vertices of a primitive are interpolated throughout its interior**.

2.6 - OpenGL Geometric Primitives

The **geometric primitives**, or **drawing primitives** or **primitives**, of OpenGL are the parts that programmers use to create objects like balls and boxes.

GL_POINTS - Draws a point at each vertex.

v0, v1, …, vn-1

GL_LINES - Draws a **disconnected** sequence of line segments between vertices, two points at a time. If *n* is not even then the last vertex is ignored.

v0v1, v2v3, … vn-2vn-1

GL_LINE_STRIP - Draws the connected sequence of segments. This sequence creates a **polygonal line** or **polyline**.

v0v1, v2,v3, …, vn-2vn-1

GL_LINE_LOOP - This is the same as GL_LINE_STRIP except an additional segment *vn-1v0* is drawn to complete a loop. This sequence creates a **polygonal line loop**.

v0v1, v2v3, …, vn-2vn-1, vn-2v0

GL_TRIANGLES - Draws a sequence of triangles using the vertices three at a time. If *n* is not a multiple of 3, the last 1 or 2 vertices are ignored. The given order of the vertices for each triangle determine its **orientation**. Orientation is important because it enables OpenGL to decide which side of a primitive, front or back, the viewer sees.

v0v1v2, v3v4v5, …, vn-3vn-2vn-1

GL_TRIANGLE_STRIP - Draws a sequence of triangles, called a **triangle strip**, in the following order:

v0v1v2, v1v3v2, v2v3v4, …, vn-3vn-2vn-1 (if *n* is **odd**)

v0v1v2, v1v3v2, v2v3v4, …, vn-3vn-1vn-2 (if *n* is **even**)

GL_TRIANGLE_FAN - Draws a sequence of triangles, called a **triangle fan**, around the first vertex in the following order:

v0v1v2, v0v2v3, …, v0vn-2vn-1

GL_POLYGON - Draws a polygon of at least 3 vertices in the following order:

v0v1... vn-1

glRectf(*x1, y1, x2, y2*) draws a rectangle lying on the z = 0 plane with sides parallel to the x- and y-axes. The rectangle has diagonally opposite corners at (*x1, y1,* 0) and (*x2, y2*, 0). The rectangle is 2D and its vertex order depends on the situation of the two vertices (*x1, y1,* 0) and (*x2, y2*, 0) with respect to each other. I really don't understand this, so you should read page 43 again.

You can use glPolygonMode(*face, mode*) to choose a different drawing mode for the primitive.

When drawing a polygon one must be careful in ensuring that it is a **plane convex**, i.e. it lies on one plane and has no "bays" or "inlets"; otherwise, rendering is unpredictable. So, it is recommended that we instead **avoid polygons and rectangles altogether and use exclusively triangles**.

2.7 - Approximating Curved Objects

The points that collectively form the shape we are drawing are called the **sample of points** or the **sample** from the shape. In the case of curved shapes, the denser the sample the better the approximation.

For a circle, points are placed equally spaced apart starting with the angle *t* = 0 and then incrementing it successively by 2π/`numVertices`.

The following conventions should be followed when writing code:

1) Program the “Esc” key to exit the program
2) Describe the user interaction at two places:
    a) The command window using `cout()`
    b) Comments at the **top** of the source code

August 28, 2018 (homework notes)

2.8 - Three Dimensions, the Depth Buffer and Perspective Projection

The most important thing to note about 3D graphics is the **depth buffer**, which is needed to create realistic 3D scenes. The depth buffer is also called the **z-buffer**. When the z-buffer is enabled, parts of the scene that are occluded are not rendered by OpenGL. This process of removing occluded sections is called **hidden surface removal** or **depth testing** or **visibility determination**.

For a given set of points belonging to objects in a scene, only the points with the **largest** *z* value are rendered in the final image.

The z-buffer is a block if memory containing z-values, one per pixel. When a scene is rendered with depth testing, when a primitive is being rendered, each of its pixels is compared with that of the one with the same (x, y) coordinates that are currently in the depth buffer. If an incoming pixel’s z-value is greater, then its RGB and z values replace those of the current one in the buffer; else, the incoming pixel’s data is discarded.

OpenGL provides another kind of projection, since orthographic projection is not good for 3D scenes. This other option is called **perspective projection**, which is done using the glFrustum() call. The call glFrustum(*left, right, bottom, top, near, far*) creates a **viewing frustum** which is just a truncated pyramid, but you knew that already. The plane *z = -near* is called the **viewing plane**. The parameters of glFrustum are usually set so that the frustum lies symmetrically about the z-axis. *right* and *top* are positive, and *left* and *bottom* are negative. *near* and *far* should both be positive and near < far.

The frustum pyramid originates from a vertex that represents where the observer is standing. This apex is the **center of projection (COP)**. With perspective projection, **the projection is no longer perpendicular**. Instead, each point is projected along the line joining it to the apex. Perspective projection causes **foreshortening**, or **perspective transform**, because objects further away from the apex appear smaller.

Perspective projection is more realistic than orthographic projection since it simulates the human eye. The second rendering step where the viewing face is proportionally scaled to fit onto the OpenGL window is exactly as for orthographic projection. The scene is also clipped according to the frustum planes. Note that orthographic projection can also be thought of in terms of a vertex for the camera position, but in this case the camera is located at infinity. Foreshortening can cause some distortion, so some applications like architectural modeling tools use orthographic projection. Since OpenGL captures the image of an object by intersecting rays projected from the object, like a real camera, OpenGL is said to implement the **synthetic camera** model.

August 29, 2018 (class notes)
Remember that negative z is “into” the screen (the direction you’re looking in when looking at the screen). This means usually your objects are drawn in the negative-z area of the world space.

August 31, 2019 (homework notes)
2.10 - Approximating curved objects once more

September 3, 2018 (class notes)
See paper

September 4, 2018 (homework notes)
3.1 - Vertex Arrays and Their Drawing Commands
OpenGL offers specific devices, the **vertex array** data structures, which make it easy and efficient for the user to centralize and share data.

Commands like glDrawElements() and glDrawArrays() are for **retained mode** rendering, while commands like glBegin()-glEnd() are for **immediate mode** rendering. In **immediate mode**, the client forces rendering by the server (the GPU). In **retained mode**, the client provides the server only with instructions to perform and the data to use, allowing the server to optimize prior to rendering.

Remember that the display routine is called repeatedly if there is animation. It’s very inefficient to store static data in this routine, or perform computations there which actually can be done once initially and the results saved. The rule is to store vertex attributes in vertex arrays, while the initialization routine is the place for one-time computations.

3.10 - FreeGLUT objects
The FreeGLUT library has lots of standard 3D shapes ready to be drawn from single commands. They are awesome. You should use them.

3.12 - gluPerspective()
gluPerspective(fovy, aspect, near, far)

This statement calls a library routine that calls glFrustum(). It creates a viewing frustum like glFrustum, but the gluPerspective() frustum is defined differently.
The parameter *fovy* is the **field of view angle**. Which is the angle along the yz-plane at the apex of the pyramid. *aspect* is the **aspect ratio = width/height** for the front face of the frustum. *near* and *far* are the same as in glFrustum(). Using these 4 parameters, we are able to make a frustum that is symmetric such that top = -bottom and left = -right.
gluPerspective() is basically just a convenience that makes it easier to define the viewport of the scene. One great thing about this function is that it allows you to change the viewport according to the OpenGL window, which comes in handy when the window is resized.
gluPerspective(fovy, (float)windowWidth/(float)windowHeight, near, far) will let you change

the viewport according to the window size, which means the scene will not be distorted when you change the window size.

September 10, 2018 (class notes)
10 years since we took computer notes in class…

September 12, 2018 (class notes)
Matrix transformations are executed bottom up, because the transformations are stored in a **stack**, so they are applied in the order of the stack.

October 3, 2018 (class notes)
Quiz topics:

- Projection
- Interpolation
- Triangulation
- **No** coding

Know: bring a calculator. **Don't** give decimals. Simplify solutions.
Given: opengl calls like rotate and color and lookat and ortho. Mcam, Morth. Equations for u and v (pixels?). How to calculate vector u and vector v. scale, rotate, and translate.

Triangulation
See paper notes

October 11, 2018 (homework notes)
Orientation
Orientation is used to determine the visible side of a surface. Note that orientation in this context refers to clockwise or counter-clockwise (handedness). OpenGL wants to know which side of an object is visible so that it knows what to render. For example, if the inside of an object is green and the outside is red, opengl needs to know if the user is viewing the inside or outside to know how to color the object.

Two orders of the vertices around any given polygon are said to be *equivalent* if one can be *cyclically rotated* into the other.

For example, the following vertex orderings are all equivalent:
v0v1v2v3 v1v2v3v0 v2v3v0v1 v3v0v1v2
This is important because a viewer on one side of an object perceives equivalent orders of vertices as either all clockwise or counter-clockwise.

There are 3 different ways to calculate the orientation of a polygon. You took notes on a piece of paper on October 10th, so look for that.

Scroll down

Consistently Oriented Triangulation

Definition: Suppose an order is given of the vertices of each triangle belonging to some triangulation *T* of an object *X*. *T* is said to be **consistently oriented** if any two triangles of *T* which share an edge order the shared edge oppositely; otherwise, *T* is **inconsistently oriented**.

There is a really great image on page 326 you should look at!!!

So of course, all triangles of a consistent triangulation will be oriented the same way.

October 15, 2018 (class notes)

Lighting

Lighting models approximate light reflection on illuminated surfaces.

Compute light reflected towards the camera/eye. Factor in surface parameters including color, shininess, etc. For light, we care about 3 vectors: vector to the eye, the normal vector, and the vector to the light source (all from the object). **These vectors need to be normalized or your lighting will fail**.

We have ambient, diffuse, and specular lighting.

Diffuse lighting

Related to how the light spreads out/scatters.

Light is scattered uniformly in all directions, so the position of the observer does not matter. The surface color is the same for all viewing directions. It is defined by:

- **Lambertian shading model**
    - Energy from the light source depends on the angle to the light source
        - Max illumination - surface directly towards the light source
        - Min illumination - surface tangent to the light source

To calculate the illumination level of an object just use the dot product. The illumination level relies on the angle between the light vector (*l*) and the normal of the surface (*n*). The *v* vector is the vector from the surface to the eye.

The equation is:

$$L_d = k_d * I * \max(0, n \cdot l)$$

L_d = diffusely reflected light
k_d = diffuse coefficient
I = illumination from source

Lambertian shading gives the object a **matte appearance**. A greater k_d will increase the reflectiveness of the object.

Specular lighting

Blinn-Phong

The intensity **depends on** the view direction. This model leads to highlights.

The reflection is brightest when *v* and *l* are symmetric across the surface normal.

You want to add *l* and *v* and normalize that. This gives you the bisector *h* of the two vectors. Then you get the angle between *h* and *n*. EQUATION ON THE SLIDES

We care about the angle between n and h because it determines how bright the reflection is. It is brightest when the angle is 0, when l and v are perfectly symmetric about the normal vector.

p = phong exponent > 1

This determines how shiny the surface is. A large p gives a small shininess (a small point of high reflectivity). The opposite is true for a small p.

h = bisector(

$L_s = k_s * l * \max(0, \cos(\text{alpha}))^p$

$= k_s * l * \max(0, n \text{ dot } h)^p$

p determines the shininess or something???????????????? I DON'T GET IT STUDY THE NOTES THERE'S ANOTHER EQUATION I MISSED

Ambient lighting

This lighting is independent of **everything**. It adds a constant color since there is light from everywhere. It adds in color to the shadow areas that are normally completely black with the other lighting models.

Full lighting equation:

$L = L_a + L_d + L_s$

$L = k_a * I_a + k_d * I * \max(n \text{ dot } l) + k_s * I * \max(n \text{ dot } h)^p$

October 19, 2018 (class notes)

Texture mapping

- Objects have properties that vary across the surface
- So we make the shading parameters vary across the surface.
- Why do we care about texture mapping? Because it adds visual complexity and makes appealing images.

- Color is not the same everywhere on a surface
    - One solution is to use multiple primitives (shit ton of vertices)

- We want a way to define a function that assigns a color to each point
    - The surface is a 2D domain (an image)
    - We can represent using any image representation

Texture mapping: a technique of defining surfaces properties (Especially shading parameters) in such a way that they vary as a function of position on the surface

Mapping textures to surfaces

- Usually a texture is an image (function of *u,v*)
    - The big question: Where on the surface does the image go?
    - The answer to this is obvious for a flat rectangle that is the same shape as the image. It's just a 1:1 map, or scaled according to the size of the texture vs the surface.
- There are also 3D texture maps, which are a function of (*u, v, w*)
    - Can just evaluate texture at 3D surface point
    - Good for solid materials

Texture coordinate function

- "Putting the image on the surface"
    - Need a function *f* that tells where each point on the image goes
    - Looks a lot like a parametric surface function
- Non-parametrically defined surfaces
    - Need the **inverse** of the function *f*
- Mapping from S to D can be many-to-one
    - That is, every surface point gets only one color assigned
    - But it is ok (and useful!) for multiple surface points to be mapped to the same texture point
- Define texture image as a function:
    - $T: D \rightarrow C$
    - Where *C* is the set of colors for the diffuse component
- Diffuse color (for example) at point **p** is then:
    - $K_D(p) = T(phi(p))$

Examples:

- A globe. For a sphere: latitude-longitude coordinates
    - Phi maps point to its latitude and longitude
- A parametric surface (eg a spline patch)
    - Surface parameterization gives mapping function directly
- It's much harder for non-parametric surfaces
    - Directly use world coordinates and apply the texture map:
        - To the object in its local space before translating it
        - To the world objects (imagine just laying the texture over the world like a sheet)

Examples of coordinate functions

- Non-parametric surfaces: project to parametric surfaces
    - Imagine segmenting a person into a bunch of cylinders
- Triangles
    - Specify (*u, v*) for each vertex
    - Define (*u, v*) for interior by linear interpolation

Texture coordinates on meshes:

- Texture coordinates become per-vertex data like vertex positions
    - Can think of them as a second position: each vertex has a position in 3D space and in 2D texture space
- Use some or all of the methods mentioned above to find the vertex (*u, v*)s

We can also do **reflection mapping**

- Easy non-decal use of textures
- Appearance of shiny objects
    - Phong highlights produce blurry highlights for glossy surfaces
    - A polished object reflects a sharp image of its surroundings

Environment mapping

- Precomputed image of scene lighting from a given point
    - Take an image of what the environment around the image would look like (light sources etc) and then apply that map onto the object

Spherical environment map

- Just map the sky texture onto the inside of a dome, for example
- Look at the slide for some examples of environment maps

Cube environment map

- Same thing basically, but using a cube instead of a sphere

You can make parallax by using multiple environment maps at different depths with transparent parts so you can see the env maps further out.

<u>Normal Mapping</u>

- Defines the normal of each polygon on the mesh, and puts them in an image. In conjunction with the texture map, it uses the normal map to calculate the light of the object at each point

Displacement mapping

- Uses an image to define the translation amount for each point

Bump mapping

- This is a type of normal map that just makes the surface have a kind of bumpy texture (think of an orange skin)
- The same effect can be done with a displacement map
    - The advantage here is that you can see the bumps on the edges, but it is **more expensive**.

Better definition with our newfound knowledge:
Texture mapping: a general technique for storing and evaluating functions

October 24, 2018 (class notes)

Basic Ray Tracing - Turner Whitted

Surfaces are bumpy which affects the light reflection.

Traditional shaders

- Gouraud
- Phong
- transparency

Why "global" illumination

- Phong shading model
    - *Local* illumination
    - Simple surface
- Blinn model
    - *Local* illumination
    - torrance/sparrow *microscopic surface* model
- Blinn and newell environment maps
    - Semi-global model for illumination

First implementation of an environment map was by blinn and newell.

Ray tracing

An old idea

The concept of ray tracing is very old. People used it for art perspective in 1600s. It was used to model heat transfer in the 1970s. It was used in graphics by arthur appel at IBM in 1968 and Robert Goldstein ad MAGI commercialized what Apple did.

Basic ray tracing

Wow he showed a fucking slide with tiny code nobody could read. Send a ray from each pixel into the focal point of a pinhole camera and find the intersection points for each object the ray hits. The nearest intersection point is the object you see in the scene.

Scroll down

Recursive ray tracing

- Make the intersection routine recursive
    - Terminate the recursion when incremental contribution is less than perceptual threshold
    - Propagate intensity values up the tree. This depends on the material properties

More code nobody can fucking read.
Nice now he showed a fucking huge slide of like 50 rays and equations.

Features

- Overcome limitations of env mapping for nearby object
- Transparency with refraction is free
- Shadows are free

Auxiliary developments

- Implement ray tracing recursively
- Acceleration structures - necessary optimization required
- Adaptive sampling - emergency hack
    - To get more resolution from our render

Acceleration

Key idea: What does a ray **not** intersect?
Coherence: What do a **bundle** of rays not hit?
Don't do computations if it doesn't intersect the object or some shit ??

Anti aliasing

Done by oversampling: generate multiple rays per pixel and filter the results
Adaptive sampling: oversample only when needed. Only do oversampling when there is an abrupt changes in intensity because this signifies an edge, which is where the jaggies are going to occur (that we are trying to eliminate).

Distributed ray tracing

Paper by R. L. Cook.
Oversample randomly
Since we're already oversampling, distribute secondary rays. What this means is just do ray tracing over time, so our rays are not all hitting the same objects since the objects are moving. This gives us: soft shadows, motion blue, and non-mirrored surfaces.

The rendering equation

By James T. Kajiya. "The rendering equation" paper.

October 26, 2018 (class notes)

Images and Displays

An image is a 2D distribution of intensity or color

In graphics, we have to represent an image in a computer readable form to then display the image.
Raster image is a 2D array that stores a pixel value for each pixel
Each pixel is an approximation
$I(x,y): R \rightarrow V$
FILL IN FROM SLIDE
Display resolution

- Number of pixels (width x height)
- $I(x, y) : R \rightarrow V$
- If you have a 10 by 10 image with only black or white, you only need 100 bits.
- 1 bit for each of Red, Green, and Blue. Each can be either on or off. In the 10 by 10 image, this requires 300 bits.
- When you have 8 bits per COLOR in each pixel, you then have 24 bits per pixel. Then you need 2400 bits for the 10 by 10 image. This is 8-bit RGB (24 bits/pixel)
- Medical imaging often uses 16 bit RGB (48 bits/pixel)

Framebuffer - drives video display from 2D array of complete frame data.
There is a thing called double buffering, which is basically just an extra copy of your screen's array of pixels. It swaps between the two buffers as it renders the next frame to be displayed. If you don't have 2 buffers and your animation is playing faster than the pixels can be displayed, you get glitches/bugs/freezing.

Note that we have a color buffer **and** a depth buffer.

<u>Cathode ray tube</u>
1920s to 1930s. CRTs are the first widely used electronic displays.

<u>Raster CRT display</u>
Scan pattern fixed in hardware. The intensity is modulated to produce an image.

<u>LCD or projection display</u>
Liquid Crystal Display. The principle is: block or transmit light by twisting its polarization. Intermediate intensity levels are now possible by a partial twist. The twist is referring to the relative angle of the 2 polarized lenses. If the lenses are rotated at opposite angles, then no light gets through.

<u>LED display</u>
Light emitting diode. Not backlit.

<u>3D Displays</u>
POGCHAMP
Stereopsis is the impression of depth as seen with binocular vision. This is how we see depth.
3 different flavors of displays

- Passive stereo - 3D movies and viewfinders. You sit and you watch something else. The thing on your head is not actively doing anything. It's up to the screen to do stuff.
- Active stereo - The lens is the component that's doing something. For example, shutter glasses. Has to be synchronized with a display that alternates between left and right images.
- Auto stereoscopic - Nintendo 3DS. **LOOK AT THE SLIDE FOR THE IMAGE**. Take pixels, and cut them in half (**loses resolution**). You can then have a lens on the pixel to bend it to the left or right eye. The cheaper option is a parallax barrier. It's basically a sheet with holes in it that limits how much of the pixel you see through each hole depending on the eye.

Final project info:
Friday, November 2: Proposal due
Friday, November 9: Updated proposal
Monday, November 12: Begin working on final projects
Slides on moodle with ideas.

October 31, 2019 (class notes)
Rasterization of lines
Read section 14.3

November 7, 2018 (class notes)
Sampling and reconstruction
IMAGE PROCESSING
Look at slides on moodle!!!!!

Why do we care about sampling? Because we resize images really often, and we don't want artifacts in our images.

Sampled representations
How do we store and compute with continuous functions? We can't store continuous data in a computer since we have discrete bits.
A common representation of continuous data is samples

- Write down the function's values at many points.

Sampling digital audio:

- The recording is sound to analog to samples to disk
- Playback is disc to samples to analog to sound

Undersampling
Example: sine wave
With undersampling:

- Information is lost

- Indistinguishable from lower frequency
- Aliasing: signals “traveling in disguise” as other frequencies

Nyquist frequency: the rule of sampling. Given the frequency of the curve, sample at 2x the frequency.

Wagon wheel effect is a common artifact of sampling rates.

Filtering
This is processing done on a function. For example, smoothing by averaging.

Convolution
Basic idea: define a new function by averaging over a sliding window.
This is how you get rid of noise.
For end points, you can ignore the edges (don’t convolve over them), repeat the endpoint, mirror it across the boundry, you can wrap around to the other side of the data (image).
Equation:

$$b_{\text{smooth}}[i] = \frac{1}{2r+1} \sum_{j=i-r}^{i+r} b[j]$$

Discrete convolution
Real convolution is the same idea but with a weighted average.

$$(a \star b)[i] = \sum_{j} a[j]b[i-j]$$

Filters
A sequence of weights a[j] is called a filter. A filter is nonzero over its region of support. It is usually centered on zero, with a support radius r.

A filter is normalized so that it sums to 1.0
Most filters are symmetric about 0 since for images we usually want to treat left and right the same.

Convolution applies with any sequence of weights. A **bell curve** of weights is the **Gaussian curve**.

Scroll down, yo.

Notation: $b = c \star a$

commutative $a \star b = b \star a$

associative $a \star (b \star c) = (a \star b) \star c$

distributive $a \star (b + c) = a \star b + a \star c$

scale factors $\alpha a \star b = a \star \alpha b = \alpha(a \star b)$

identity: unit impulse e = [..., 0, 0, 1, 0, 0, ...]

$$a \star e = a$$

November 12, 2018 (class notes)

Lecture continued

Convolution and filtering

So, a way to downsample the image is to apply a Gaussian filter (average) to the image.
So we need 2D now.

For discrete filtering in 2D:

$$(a \star b)[i, j] = \sum_{i', j'} a[i', j'] b[i - i', j - j']$$

Now the filter is a rectangle and your slide around over a grid of numbers. This is used in images for blurring and sharpening, commonly.

Given an image of dimensions *n* x *m*, and a filter radius of *r*, the efficiency of this algorithm is $n*m*(2r+1)^2$. That is very slow.

Optimization: separable filters

Definition: $a_2(x, y)$ is separable if it can be written as: $a_2[i, j] = a_1[i]a_1[j]$

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

Convolve image with this filter

0	0	0	0	0
0	0	0	0	0
1	4	6	4	1
0	0	0	0	0
0	0	0	0	0

0	0	1	0	0
0	0	4	0	0
0	0	6	0	0
0	0	4	0	0
0	0	1	0	0

Convolve image with these two filters separably

This is an efficiency of 2(n*m*(2r+1)) = nm(2r+1).

Details and issues

What about edges? There are a few ways to deal with this:

- Clip filter - just cut off edges of the image
- Wrap around - use values from the other end of the image
- Copy edge - extend the last pixel of the image r times into the empty space
- Reflect across the edge
- Vary the filter near the edge - ignore edges and don't do convolution

Why?

This is how we reduce and enlarge images

Issues: devices have different resolutions and applications have different memory/quality tradeoffs.

The poor approach to resizing is to drop pixels from the image.

The good approach is to use resampling.

November 14, 2018 (class notes)

Quiz format:

- Take home quiz. Turn in on monday
- Problems similar to previous quiz
- Open book, slides, notes. NOT open-previous-quiz

$k_a I$ = ambient color