Lesson 1: Introduction

Motivation and a trip down the graphics pipeline, laying out the fundamental processes

Lesson: Introduction

[text inside brackets are “stage directions” or other things, vs. what I’ll be saying.]

[video: head shot. Perhaps the italicized phrases below are put on the screen for emphasis as I talk.]

I’m Eric Haines and I’ll be teaching about something I love: computer graphics. What’s not to like? In this area of computer science you write code to create cool pictures. I’ve been programming computers since 1974 and have been working on computer graphics since 1983. With computer chips constantly getting faster, and with graphics accelerators in particular evolving at a rapid clip, the rules about what’s possible change time and time again. There are always new things to learn. I guarantee that the computer languages and hardware we use *will* change. The good news for you as a student is that I’m going to focus on what makes computer graphics tick. The underlying principles I’ll cover here will give you a firm foundation in the field.

I’m particularly excited that I’m able to teach this class now. A few years ago such a class would have been extremely difficult. With the rise of a web technology called WebGL you can simply click on a link and immediately run a 3D demo yourself.

In some trivial sense, everything you see on your computer screen can be called “computer graphics”. Word processors, videos such as this one of me, paint programs, and much else result in images on your monitor.

[Show phrase “***Interactive 3D Rendering***”, which I will hand-draw - images come in as each word is described]

I’m going to focus on *interactive 3D rendering* in particular. Let’s break that phrase down, “interactive 3D rendering”, and start at the end. The last word, “*rendering*”, means “depiction”.

[draw little pictures of translating, and making chicken in, sausages out for the word rendering.]

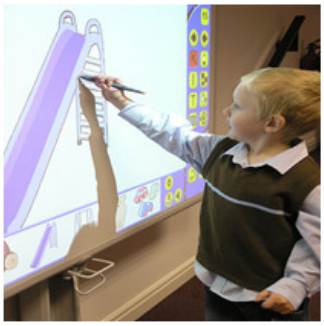
Well, it can also mean “translating to another language”, or “turning animals into meat”, but let’s ignore those definitions.

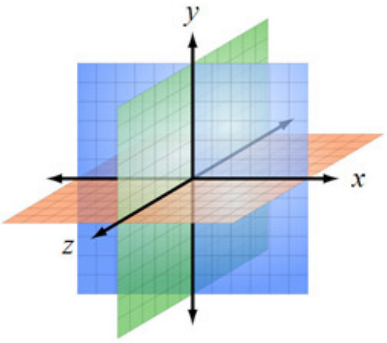
[erase, and replace with image of building, as shown below - progress from right to left]

An artist can render a scene by painting it, an architect can render a building in perspective, and so on. To “render” means to create an image.

[building: **http://commons.wikimedia.org/wiki/File:Boston_building_historic_drawing.jpg**
XYZ http://commons.wikimedia.org/wiki/File:3D_coordinate_system.svg
Interactive http://commons.wikimedia.org/wiki/File:Interactive_whiteboard1.jpg]

Interactive 3D Rendering





For the phrase “*3D*”, this means defining things in a three dimensional world. In the field of 3D computer graphics, you mathematically define *objects*, *materials*, *lights*, and a *camera*, then using these together to render onto a two-dimensional screen. Your monitor becomes a window into this world that you have created. Even newer smartphones have 3D graphics accelerators in them. You’ve undoubtedly seen videogames and special effects in movies - these are two of the most obvious uses of 3D computer rendering techniques.

[screenshot of teapot running on my phone. Better yet, record it live!]

There are a vast number of other uses of 3D rendering that affect how the world works. Architects commonly create 3D graphics presentations for their own benefit and for clients. Engineers use rendering to help show potential problems with their designs. Doctors use 3D

reconstructions from medical scanning devices to understand patients' problems. Even lawyers will employ 3D animations to help explain accidents or other events.

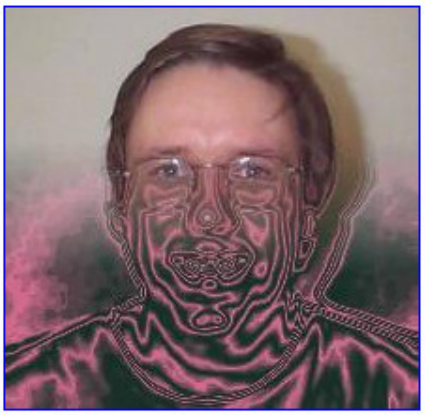
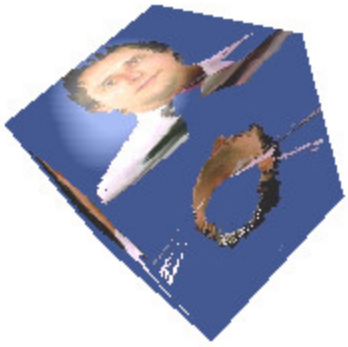
"*Interactive*" means that you can instantly influence what you see on the screen. Videogames give us obvious examples: you turn the wheel left in a racing game and go that direction. Compare this to, say, playing a DVD. [draw controls] There's perhaps some minimal control that could be considered interactive - pause, fast forward, and so on - but you are not affecting the world itself in any way. Drawing these lines is interactive [draw a doodle, as shown], I'm affecting what I see, but it's two dimensional. By "interactive 3D" I mean that the user's actions affects something in a 3D *virtual world* - at the least, you move the camera through the world, possibly affecting objects and lighting as you go.

Here's an example of an interactive 3D application [show http://gallantlab.org/semanticmovies/]. [I have clearance from Jack Gallant: "Thanks for your interest. We're thrilled that you find our site interesting and we're happy to grant whatever permission Udacity requires. (I'm on your side with fair use!)

"Just FYI we'll be releasing the server and client software for the brain viewer in the coming months. But its not yet ready for public consumption."
]

This program by James Gao shows MRI data of how the brain is stimulated when exposed to different objects and actions. On the right is a 2D system for picking an object. On the left you then see the areas stimulated. You can also change the view, and with the slider control how the data is represented. I asked the creator of this program, Jack Gallant, about how much of the brain is dedicated to processing what our eyes see. The answer is that more than a quarter of our brain is involved in interpreting visual signals.

[at this point we could go back to me talking on say the left side of the screen, put up the various images at http://erich.realtimerendering.com/ of my face (move your cursor over the image to change it), jumping from one to another (or cross-fading?) to show image processing - or, better yet, just run a bunch of video effects on my head as it talks, then end by adding the animated spinning cube of my head. Might as well end with a video flourish, much of the rest is going to be me writing.]

It's worth noting that there are other areas of computer graphics having to do with images but that are not truly 3D rendering. For example, there's a complementary field called "image processing", where you use the computer to analyze photographs or other images and try to extract a mathematical description of the world from them. Though graphics hardware can definitely accelerate image processing operations, we'll mostly be focused on 3D rendering in this course.

[Instructor Comments: You can try the brain program by going to this link http://gallantlab.org/semanticmovies/ . Chrome is the best browser for it. If it doesn't work for you, don't worry - the next lesson will help guide you through setting up WebGL on your machine.]

Question

So, your first quiz: **which of these can be described as using interactive 3D rendering?** Feel free to search the internet for any of these titles and terms to learn what they are.

[Checkboxes used.]

[] The computer games *Call of Duty* and *World of Warcraft.*
[] The *Toy Story* and *Ice Age* movies
[] Recording a 3D image for medical purposes, such as a CT (Computerized Tomography) scan of a patient's head
[] The mobile phone games *Angry Birds* and *Pacman*

Answer

Only the first answer is correct. In Call of Duty, World of Warcraft, and many other videogames you move through a 3D world and interact with it.

[I'd love to show examples from each source, I think it's Fair Use - we're not competing with the items shown, it's for educational purposes, etc. - but let's avoid those who like to overreach on copyright. So, self-censorship.]

Pixar's "Toy Story" was the first film made completely with computer graphics, back in 1995. It and many other animated films such as the *Ice Age* series use 3D rendering techniques. However, the final product is not interactive, it's a linear film that you have no real control over. Each image in a computer animated movie can take from seconds to minutes to hours to compute, depending on the complexity of the scene. That said, artists and technical directors working on these films depend on interactive rendering techniques to set up lighting, design materials, and create animations.

Computerized Tomography gathers slices of data. However, gathering the data by performing the scan is not the same as rendering this data.

Pacman is interactive, and is rendered, but the scene described is entirely in 2D. Games such as *Angry Birds* are also in this category, where an illusion of 3D is achieved by having 2D layers that move at different speeds. See the Instructor Comments below for more information. Mobile phones and tablets can certainly perform 3D rendering, but these particular games don't need such capabilities.

[NOTE: Instructor Comments: Games such as Angry Birds use a form of 2.5D rendering, which you can read more about on Wikipedia: http://en.wikipedia.org/wiki/2.5D]

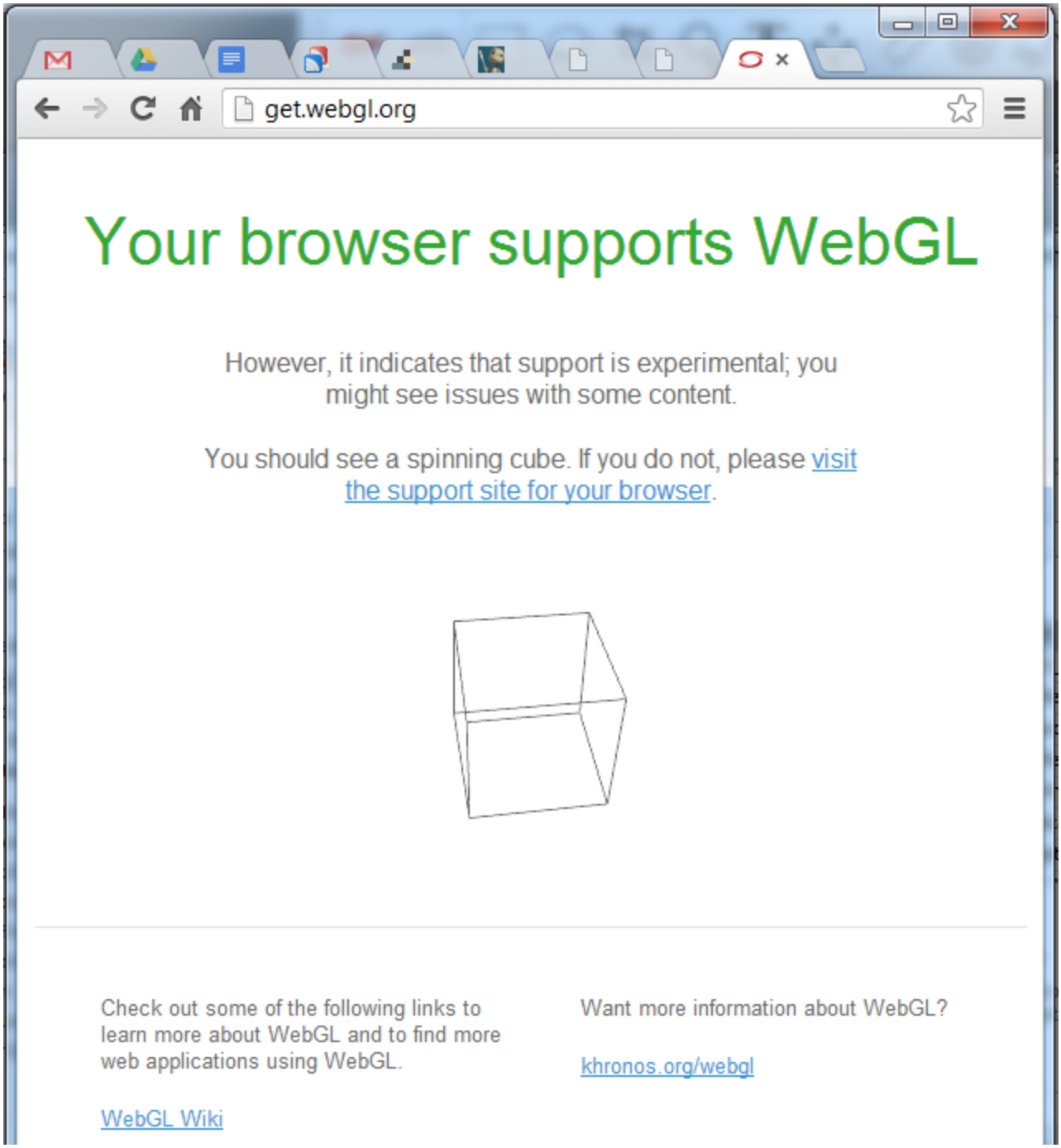
Lesson: WebGL Setup

Now, before we go any further, you should make sure your computer can run WebGL programs. "GL" stands for "*Graphics Library*", and it's what this course uses to teach 3D graphics.

WebGL is built into a number of web browsers, so you don't need to install anything. Unfortunately, there's one major exception: Internet Explorer. So, you'll need to use a different browser, such as Chrome or Firefox. On Macs, Safari supports WebGL, but you'll have to do a little twiddle to enable it. Linux has its own issues. You may also have problems with older graphics cards and drivers. If you get stuck, see the Instructor Comments below this video.

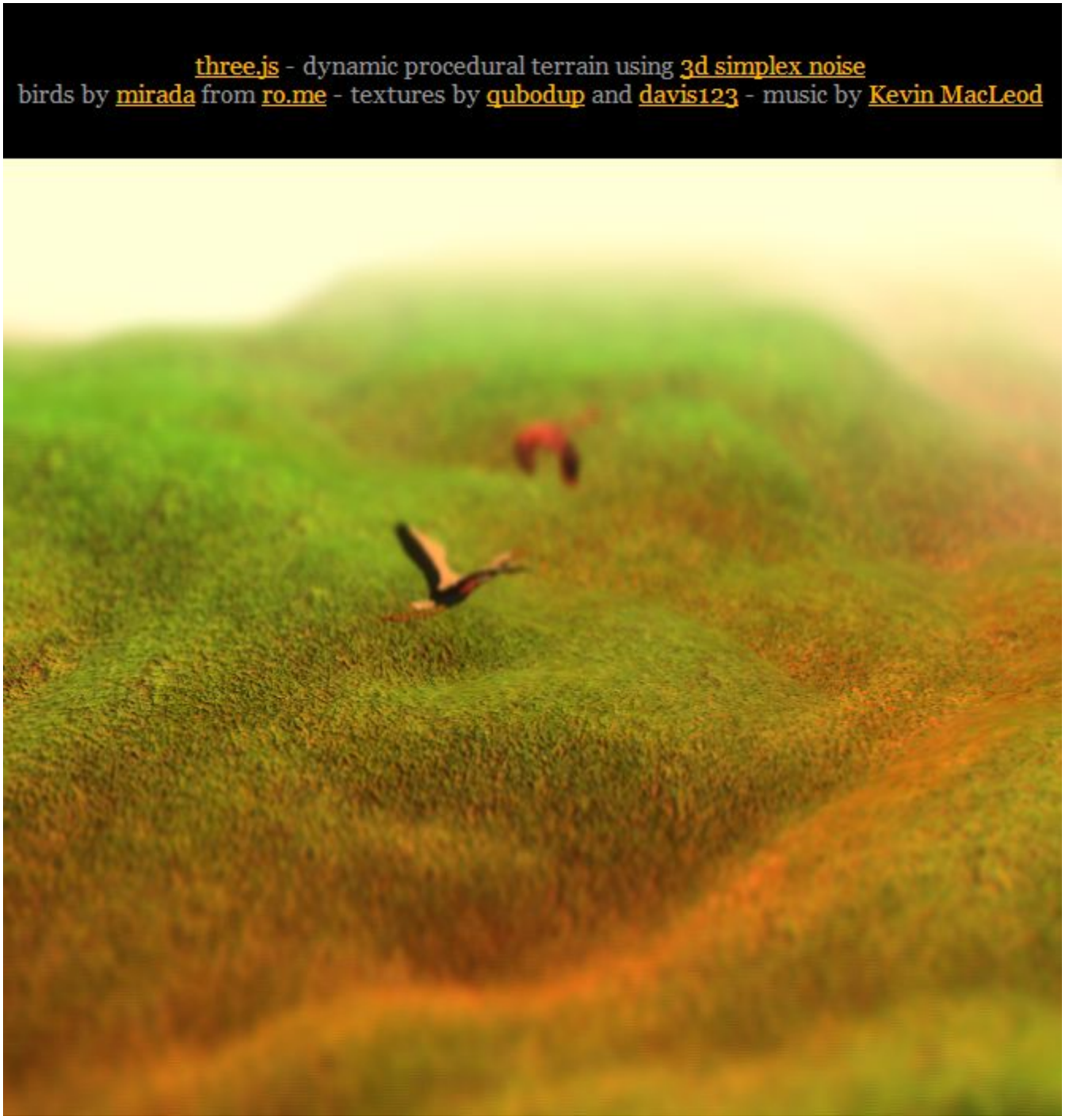
The easiest way to test and debug your computer is to visit the following URL: get.webgl.org.

[put URL http://get.webgl.org on screen as background and show the website running].



If you see a simple rotating cube on the screen, you're all set. If not, follow the instructions on that web page to get your computer properly set up. If one browser doesn't work for you, try another. My personal preference is Google's Chrome, as it has a JavaScript debugger built in. If you get stuck, see the Instructor Comments for more tips and suggestions. Ultimately, you have to get WebGL to work in order to fully enjoy this course.

[Switch to video of WebGL program running in browser, e.g. webgl_terrain_dynamic.html]



Once you have it set up, you can go to a huge number of web pages that have graphical demos running on them. Look at the links provided for some of the better programs out there.

[Instructor Comments: To see if your machine is set up right, go to http://get.webgl.org/. If your machine doesn't show a spinning cube, go to http://www.khronos.org/webgl/wiki/Main_Page and

look under “Implementations”. For Safari on the Mac, follow these instructions http://www.khronos.org/webgl/wiki/Implementations/WebKit.

An excellent summary of what supports WebGL is here: http://codeflow.org/entries/2013/feb/02/why-you-should-use-webgl/

Some graphics cards are blacklisted because their drivers are old, never to be updated, and won’t work with WebGL. See this page http://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists for more information. It’s possible to override the blacklisting in Firefox, see this article: http://www.sitepoint.com/firefox-enable-webgl-blacklisted-graphics-card/ - this might be an acceptable “if all else fails” solution, since this course will be using fairly vanilla WebGL features. Google Chrome has blacklisted XP, so there’s a similar workaround: http://code.google.com/p/chromium/issues/detail?id=72975. If all else fails, try different browsers, as they have different limitations.

[We can take information from here, too, about how to update drivers, with the author’s permission: http://cesium.agi.com/tracksanta.html]

[Once you’ve set up, try a more interesting demo, like this http://mrdoob.github.com/three.js/examples/webgl_terrain_dynamic.html]

[Supplementary material: Here are some demos worth trying out.

- three.js - we’ll be using this library in the course. At the top of this page are a number of interesting projects based on it.
- Zygote Body: human atlas, uses webgl-loader to speed download. Try searching on body parts.
- Car demo (mute your sound) and another car demo.
- Jellyfish - lovely and hypnotic
- WebGL Water: go ahead, touch the water.
- potree: point cloud rendering and technologies. Interesting to watch the data stream in.
- St. Jean's Cathedral: beautiful lighting. Left mouse changes light direction, right and WASD move. Many options.
- SnappyTree: a cool little tree builder.
- Progressive loading: showing bounding boxes and then part by part, with an interesting fade effect.
- Volume rendering: recommended by David Larsson, doesn't load for me.
- Fluid: nice interactive 2D fluid simulator.
- Brain: MRIs of brain and what areas are stimulated by different words
- 3D print preview: Sculpteo uses WebGL throughout its site. These can take awhile to load. Another good one here.

]

Unit 1: Introduction

In this unit we're going to look at some of the core ideas behind interactive rendering. First I'll discuss interactivity and explain concepts such as the refresh rate and frames per second.

I'll next talk about how the human eye and how a camera works. These both relate to the idea of defining a view frustum as a way to see into a virtual world. I'll discuss how light travels through the real world, and how we simplify this physical process in our virtual world in order to rapidly make pictures of it.

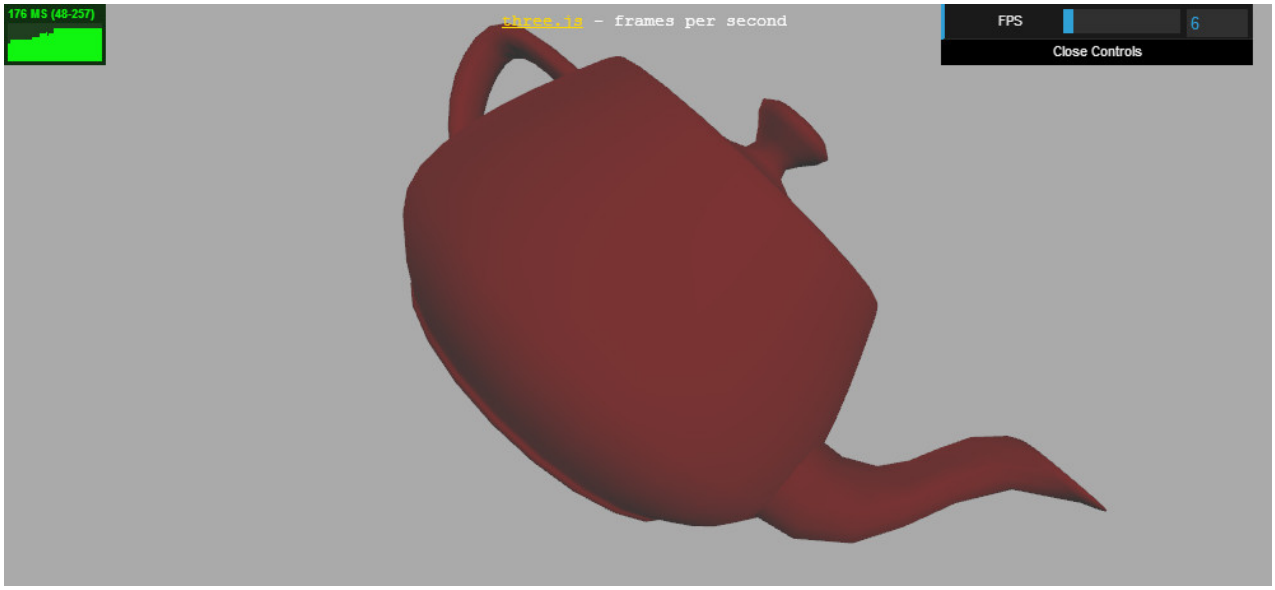
In the latter half of this unit I'll focus on what the graphics pipeline is and how graphics hardware is able to interactively render complex scenes. I'll end by discussing two rendering algorithms, the Painter's algorithm and the Z-buffer.

By the end of this unit you should have a general sense of how rendering is done with a computer and how graphics hardware accelerates this process.

[List out:
* Refresh rate, frames per second
* The eye, pinhole camera, view frustum
* Light: physical and virtual
* The graphics pipeline: how it works
* Painter's algorithm and Z-buffer
]

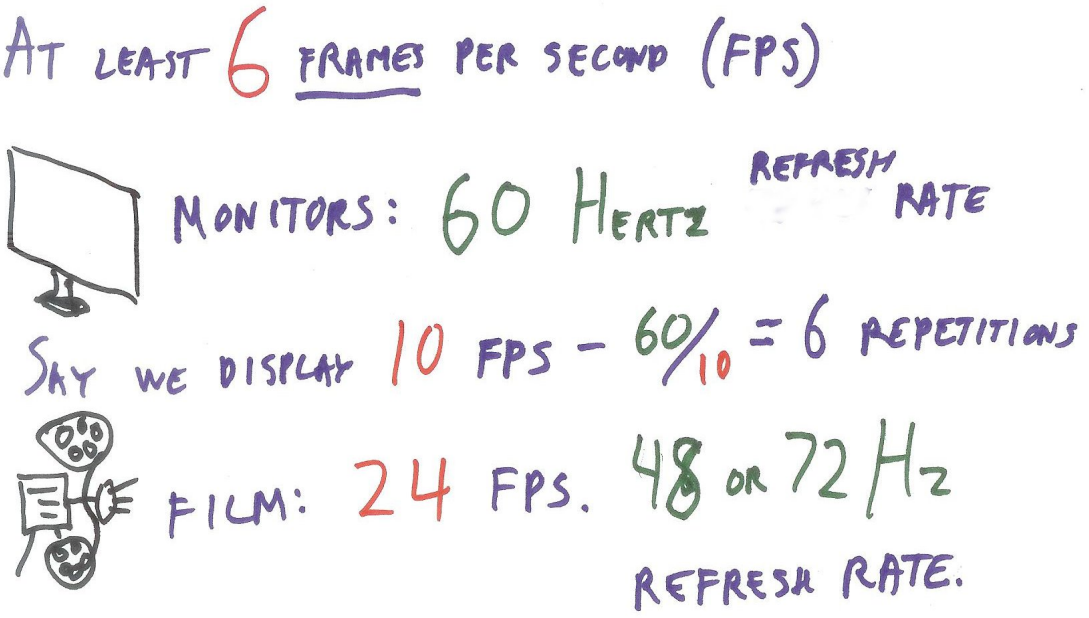
Lesson: Interactivity and FPS

[Start with video of http://www.realtimerendering.com/udacity/?load=demo/unit1-fps.js - Slide slider from 60 down to 1 FPS.]



If we want to play a videogame, or design a new tennis shoe, or try out different colored paints on a building, we would ideally like the experience to be interactive. When we take an action, we would like a response from the computer quickly enough that we don't feel slowed down. How fast is fast enough? I'll give you a real number: it is about six frames per second.

[cut to 6 Frames Per Second - > 6 FPS]



What we mean by this is that 6 different images are displayed each second. This sounds a bit arbitrary, but testing has shown that anything less than this rate feels unresponsive, like you're wading through mud. Instead of a feeling of working with the computer, you get the impression that you're constantly waiting for it to catch up.

In fact, for videogames the minimum rate is often 30 frames per second, or even 60 frames per second. These numbers are *not* arbitrary, but are tied to the hardware itself. Most monitors these days, as least in America, refresh at 60 Hertz, which means "display the image 60 times a second". This is called the *refresh rate*. This value is part of the monitor itself, not something that varies.
[60 Hertz refresh rate]

This value of 60 Hertz gives an upper limit on the *frame rate*, the number of *new* images per second. However, an application can certainly have fewer frames per second. So, for example, say it takes us a tenth of a second to compute and display each new image. This would mean our frame rate is 10 frames per second. The display itself may refresh, that is, send its stored image to the screen 60 times a second, but most of those images will be the same.

In the film industry, the frame rate is 24 frames per second, but the refresh rate can be 48 or 72 Hertz, with each frame repeatedly projected 2 or 3 times.

It surprises people that the frame rate of film is so low compared to computer games with rates of 60 frames per second. I won't go into the details in this course, but the key reason that film

can have such a comparatively low rate is that film captures what is called **motion blur**. When a frame is captured by the camera, objects moving will be slightly blurred. This gives a smoother experience and so a lower frame rate is possible. A fair bit of research has been done in the field of computer graphics to rapidly create frames with motion blur, since they look more realistic. See the Instructor Comments for more on this topic.

However, a film is not interactive. A faster frame rate in an application also allows a faster reaction time, giving a more immersive feel.

[Instructor Comments: You can give the FPS demo a try yourself, find it here http://www.realtimerendering.com/udacity/?load=demo/unit1-fps.js - most graphics accelerators should be able to run this demo at 60 frames per second.

The Wikipedia page on motion blur http://en.wikipedia.org/wiki/Motion_blur gives a start on the topic. You can see a little bit of some motion blur correction in this demo http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_dynamic.html, Go to view 4 (hit the 4 key) and toggle motion blur on an off with the “b” key. The ground will be blurrier as you move when motion blur is on.]

Question

In some parts of Europe and over much of the world, televisions use the PAL standard. PAL stands for Phase Alternating Line. Maybe I shouldn’t show the European Union flag, as many of these countries started with PAL but have moved or are moving to higher quality standards. A frame is made up of two fields, and a field is made of the alternating horizontal lines on the television. Field updating is not used on computer monitors.

How many milliseconds go by between each field update, if fields are updated at 50 Hertz?

[ ] milliseconds

Answer

20 milliseconds. This is an incredibly small amount of time, and isn’t even the fastest time strived for by games. If you have a CPU running at say 2.8 Gigahertz, that’s 2.8 billion instruction cycles a second. During 20 milliseconds this comes out to be 56 million cycles for that single CPU. This sounds sizeable, but if there are a million pixels on the screen, this leaves just 56 cycles per pixel per frame - not a lot of time. Later in this unit we’ll see that the CPU usually takes care of creating and positioning objects in the scene. These objects are then sent to a graphics processor, which is particularly tuned to rapidly display them.

[**20 ms**
2.8 GHz = 2.8 billion * 20 ms = 56 million per frame / 1 million pixels = 56 cycles per pixel

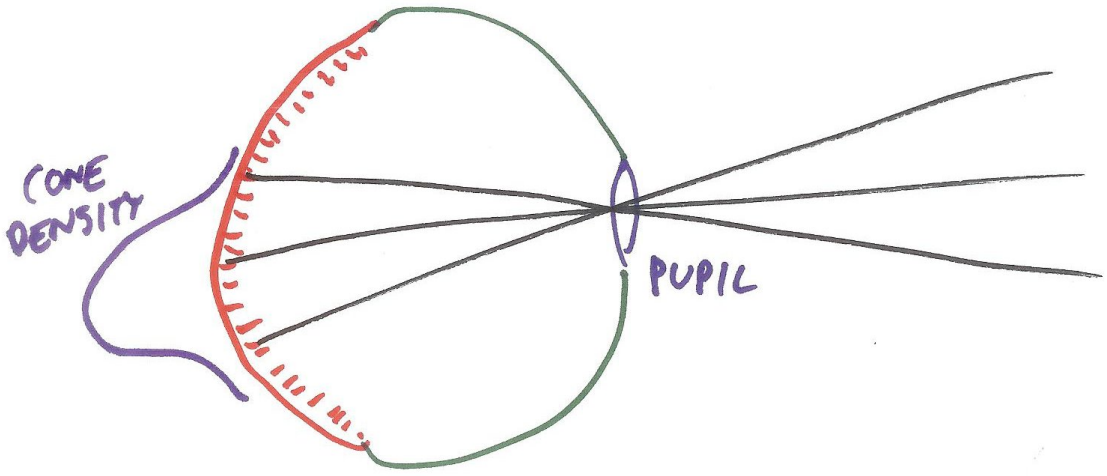
per frame]

A game running at 60 frames per second allows just 16.6 milliseconds per frame. What's important to realize is that this 16.6 millisecond number is usually a hard limit for some games. Other games aim for a rate of 30 FPS, giving themselves more time to generate higher quality images. This rate still is only 33.3 milliseconds per frame. Often considerable time is spent to ensure that every part of a game runs at the minimum designated FPS goal. This is done by improving code efficiency and by trimming back on content such as geometry and lighting in a scene.

[Supplemental Material: Some applications will aim to avoid a rate between 30 and 60 FPS, since then the frame rate doesn't align with the refresh rate. This video https://www.youtube.com/watch?v=hAzhayTnhEI explains in detail how this mismatch can cause a sense that the interaction with the application is not smooth. That said, many games simply strive for as fast a rate as possible.]

Lesson: The Eye

So, how do we see anything? Well, clearly, we use our eyes, but let's talk a bit more about that. [drawing here of eye looking to the right - give a bit of room to the left for the distribution function, which is added later.]



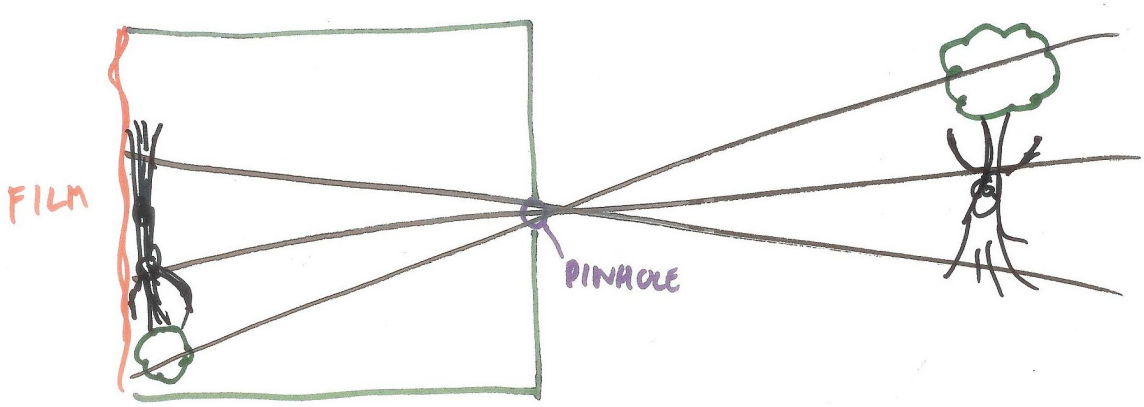
[draw black lines]
Light travels through the *pupil*, through the eye's interior, and makes contact with the retina in the rear, which is covered with *rods* and *cones*. The rods are for low-level lighting, such as during

nighttime, and also for peripheral vision and motion detection. The cones provide us with color vision. In the lesson on materials we'll talk more about our eyes' three types of cones and how they help us perceive color. For now, the key idea here is that light coming from a particular direction is detected by cones in a particular spot in the eye.

This description barely touches on the complexity of the eye. The iris around the pupil opens and closes to let more or less light in. The back of the eye is a curved surface. The density of cones is highest at the fovea, which is where our eyes focus for reading or other activities. [draw distribution curve behind retina.] And we haven't even touched on what happens to these signals when they're processed by the brain. [draw neural output line in blue]

Lesson: Pinhole Camera

An easier model for rendering is the pinhole camera.
[pinhole camera drawing here]



With a pinhole camera you put a piece of photographic film on one end of a box and poke a hole in the other end, then cover up this hole. You do this in the dark so that the film is not exposed to light. Of course, just about no one uses film any more, but I'll assume you have some idea how camera film works.

To take a photo, you point the camera, open the hole for a short amount of time, then go back to a darkroom and develop the film. With a pinhole camera the light coming in from one direction is detected by a single location on the flat rectangle of the film. Put these differing light contributions together and you get a picture.

Most cameras work the same way: they may have more complex lens systems, but they gather light in a similar manner, with the image projected behind the lens. We often simplify and think of our eyes as elaborate cameras that send images directly to our brains. It's better to think about the eye as a part of our visual system, since ultimately it's the brain's interpretation of the data that matters.

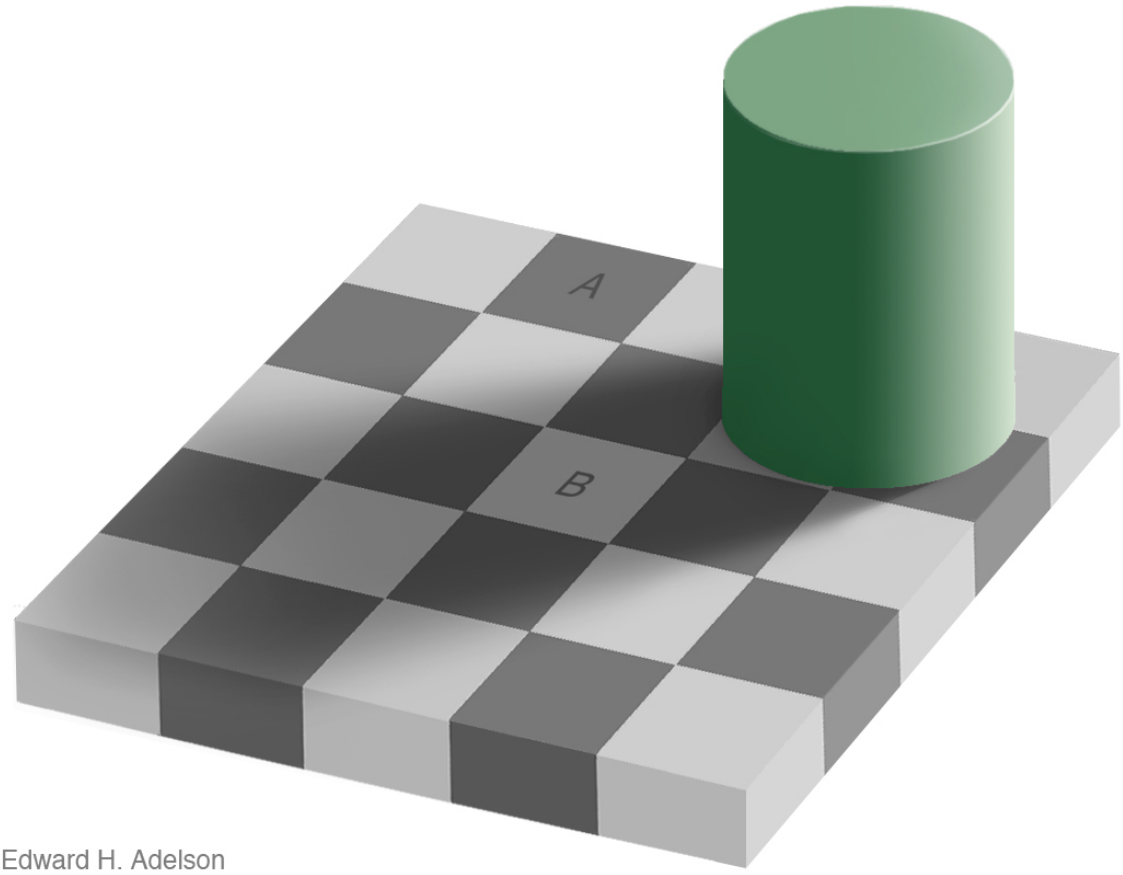
Question: Seeing is Believing

[Weird, the text seems to be missing! Maybe I accidentally deleted it at some point.]

Answer

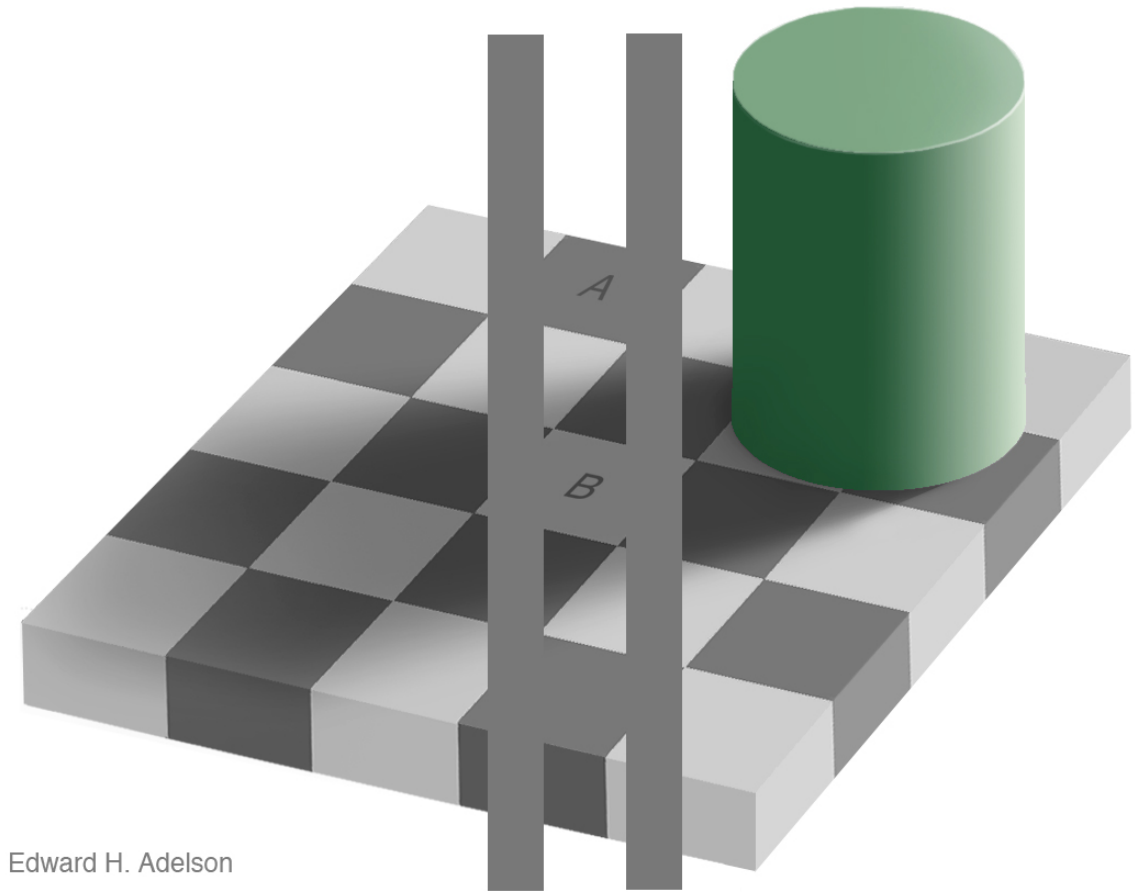
Here is one of my favorite optical illusions, by Edward Adelson at MIT.

[http://persci.mit.edu/gallery/checkershadow - can use freely!]



Edward H. Adelson

So the shocking truth is that the squares marked A and B are the same shade of gray. If you don't believe it, here's a bit of proof, an image where two vertical bands with the same shade of gray are overlaid on the image:



If you still don't believe it, you can print out the original image and cut out squares A and B and hold them next to each other. See the Instructor Comments section for where to get these images.

As Professor Adelson notes, this illusion proves that the visual system is not a very good light meter, but that's not its purpose. Our visual system tries to compensate for shadows and other distractions so that we can properly perceive the nature of the objects in view.

[Instructor Comments: You can find the original images for the illusion here http://persci.mit.edu/gallery/checkershadow, along with a full explanation of how it works. There's also a clever video http://www.youtube.com/watch?v=z9Sen1HTu5o showing the effect. I love optical illusions; my favorite sites include Michael Bach's collection http://www.michaelbach.de/ot/, Kitaoka's works

http://www.ritsumei.ac.jp/~akitaoka/saishin-e.html, and these wallpapers http://www.flickr.com/photos/w00kie/sets/180637/show/. You can also lose a day wandering through the “Mighty Optical Illusions” blog http://www.moillusions.com/ .]

Question: Eye vs. Camera

[Write out the following:] Of the following, what is the biggest difference between how our visual system works and a camera’s mechanism?

[( ) means “radio button”, only one answer is correct.]

( ) By putting lenses in front of our eyes we can change the sharpness of what we see.
( ) If the light is too bright, we become temporarily blinded.
( ) We can change our focus, such that parts of what we see become fuzzy.
( ) We see images “correct side up”, not upside-down.

Answer

Higher-quality lenses can be used to improve clarity or change the focus of a camera, so the first and third answers are not very strong. Too much light will also “blind” a typical camera, as the image produced will be overexposed, often appearing all white.

The answer is that we see images right side up. Look at our pinhole camera model: the image is actually formed upside down.

[put pinhole camera image next to eye image.]

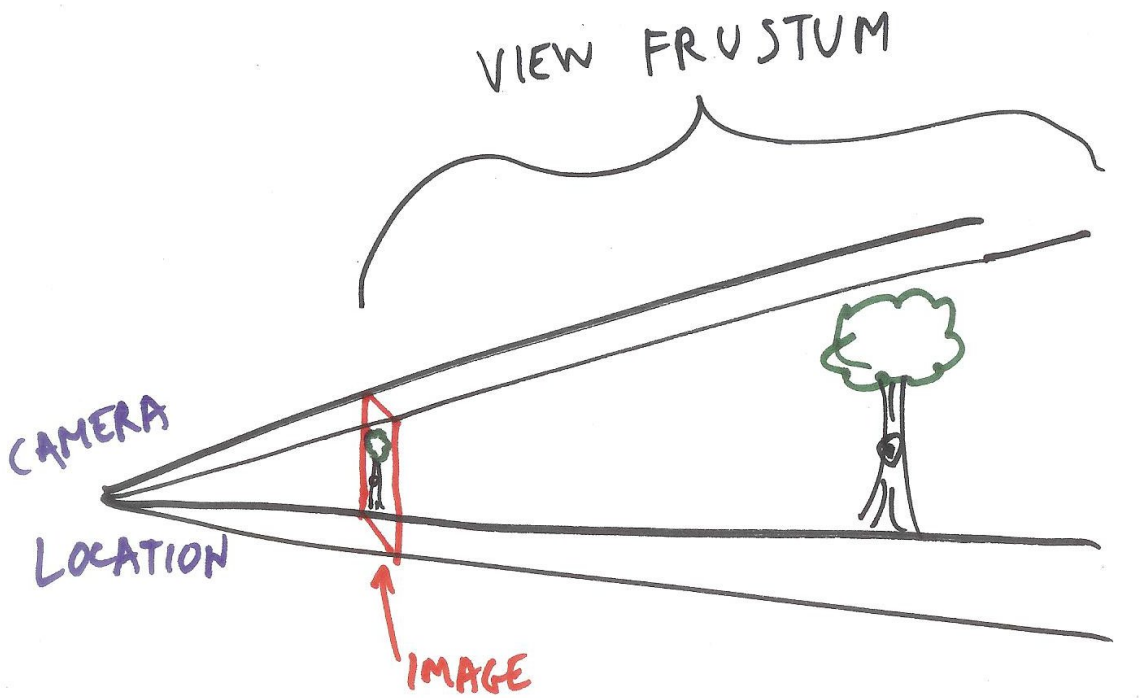
So the simplistic model that the image on our retina goes directly to our brain in that form is clearly false. There is in fact a considerable amount of processing that occurs in the brain beyond just rotating the image 180 degrees. For example, our eyes have blind spots where the optic nerve is attached. Our brains fill in this area with the surrounding elements in the scene.

[Instructor’s Comment: eyes are fascinating organs, especially since there are a wide range of designs. See the Wikipedia article on the eye http://en.wikipedia.org/wiki/Eye for some truly amazing knowledge.]

Lesson: View Frustum

A pinhole camera captures an image upside down. In computer graphics we think of the camera a bit differently, with the image being formed *in front* of the viewer. This is a more convenient way to consider the task of image generation. For starters, now the object projects onto our screen right-side up.

[show pinhole camera view, then scroll to the view frustum - USE STICK FIGURE, and draw the frustum line. Also point to which part is the view frustum, as shown in first image.]



This pyramid-shaped view of the world is called the *view frustum*. “Frustum” means a pyramid with its top chopped off. It is one of the most frequently misspelled words in computer graphics, so lock it in your brain now. There’s no letter “R” in the last half of that word - I suspect it sneaks in there because of the “R” in the word “frustrate”, or maybe the word “fulcrum”. Enough about that.

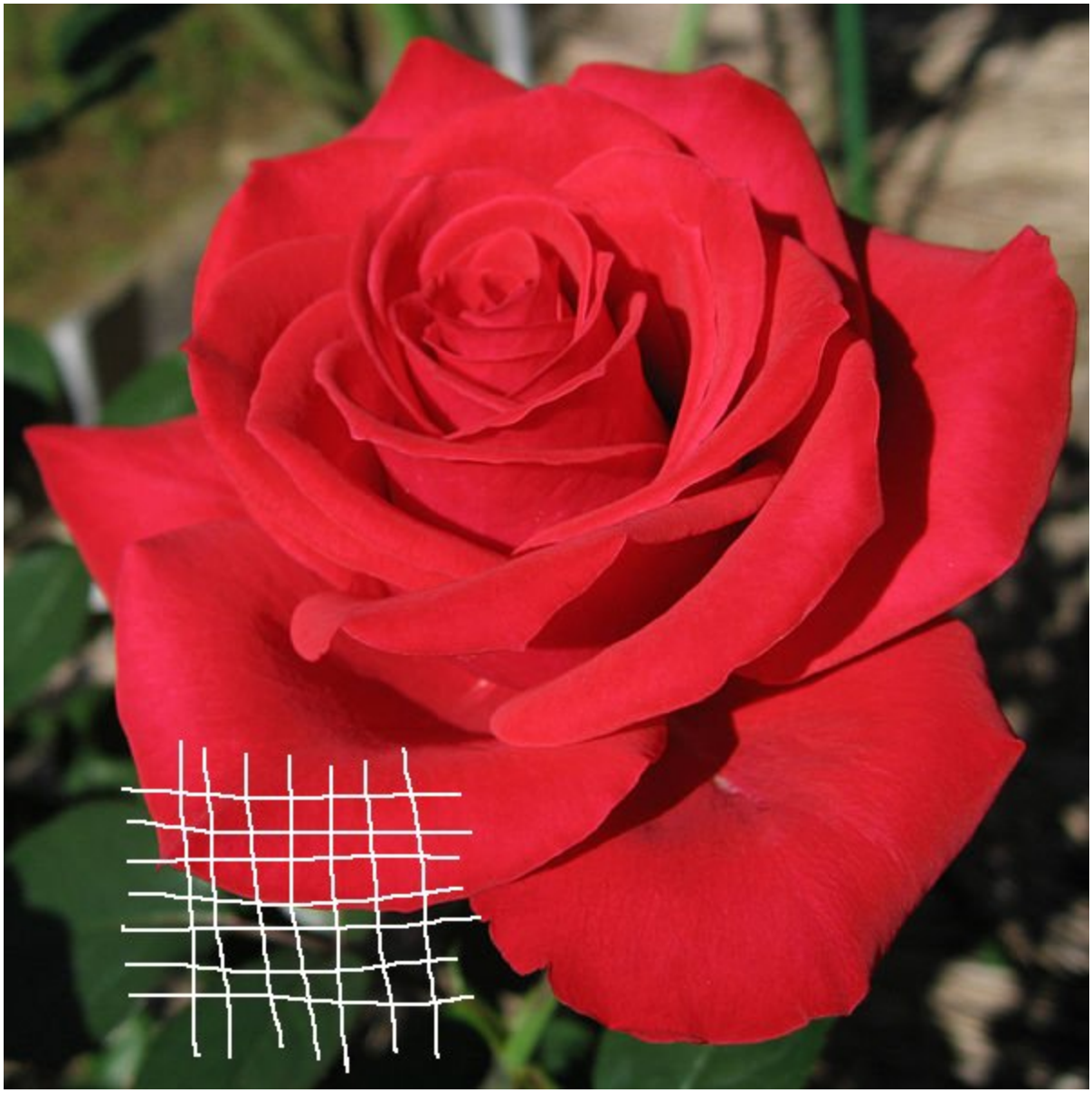
The point here is that we want to know how much light is coming from each direction. Each light emits photons, which bounce around the world and may get absorbed or otherwise changed. In a camera we gather photons from the world and these create a photograph. In computer graphics we simulate the real-world as best we can and try to determine how many photons are

coming from each direction. Whatever amount of light we gather from a direction we record at a pixel on the screen.

Lesson: Screen Door

Another way to think about rendering is as if we're looking through a screen door or a wire screen in a window. The screen is there to keep out flies, but for my purposes each little square on the wire screen is like a pixel on a computer screen. Through some screen squares we see a bright red, through others a deep green, putting them all together we get a rose. When we look at our computer screen, we can think of it as a window into a 3D world. However, moving our head in the real world normally does not change what we see through this window. Our task in 3D computer graphics is to define a world and then figure out as best we can what light comes from each direction and so forms an image.

[wikimedia: http://commons.wikimedia.org/wiki/File:Rosa_Red_Chateau01.jpg]
[This image should have a grid of white lines on it, showing pixels:]

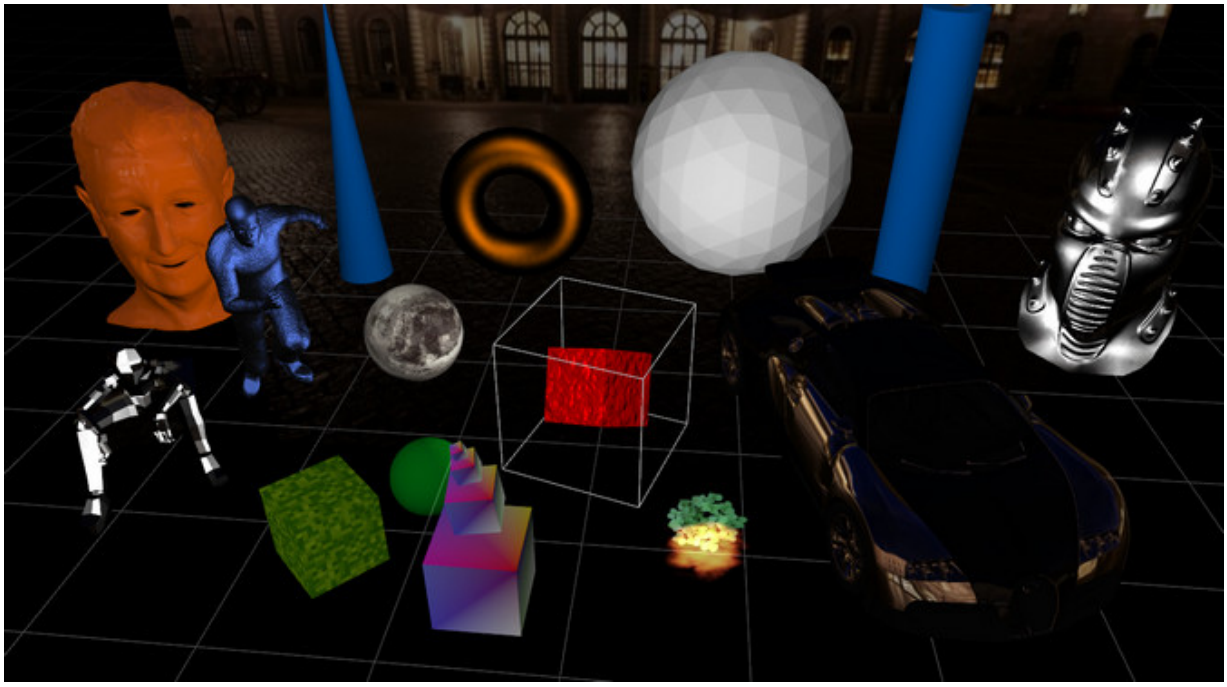


[Instructor Comments: See my blog entry http://www.realtimerendering.com/blog/do-you-spell-these-two-words-correctly/ for way too much information on spelling the word “frustum”.]

Lesson: 3D

The computer renders a 3D scene by determining how much light comes from each direction. There are a few elements that go into this calculation. First, we have to define a world. For example, take this scene:

[webgl_loader_scene.html - show it interactively]



Here there are many different objects, defined in different ways. Each object is described by some 3D geometry. It also has some material description saying how light interacts with it. Objects can also be animated over time - the red cube is obviously spinning, and if you take a closer look at the metallic sitting man on the left you'll notice he moves.

There are also lights in this world, which in this case are not animated. As the view changes, some materials look different as the light reflects off them at a different angle, while others don't change.

[move mouse to show interactive effect.]

Finally, a camera is also defined for the scene, and interactively changes under the user's control.

[Gundega: is this repetitive? Not sure how to show this summary while also doing the demo.]

To sum up, a scene consists of objects, which typically are described by geometry and material information, along with lights and a camera.

[**Scene**
  Objects
    3D Geometry & Material

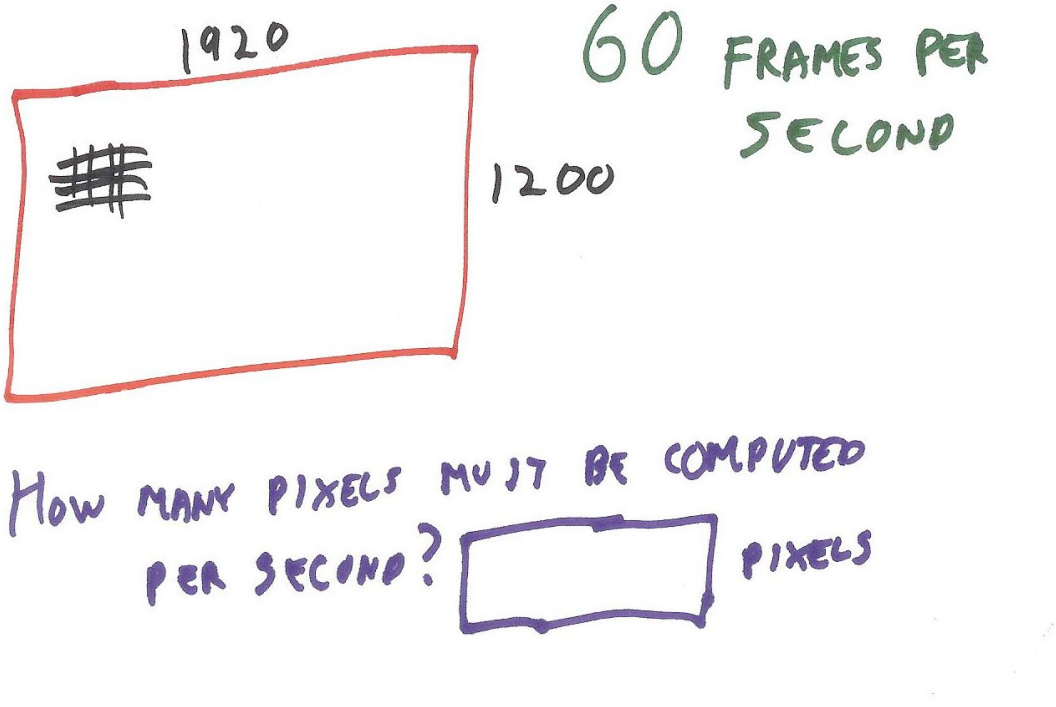
Lights
Camera
]

[Instructor Comments: you can look at the demo shown in the video by going to this link, http://mrdoob.github.com/three.js/examples/webgl_loader_scene.html [we should really host at Udacity], letting it load and then clicking “Start”. For an in-depth overview of how three.js labels elements in a scene, see this page http://ushiroad.com/3j/]

Question: How Many Pixels?

Guess what: computers are fast! A graphics processor consists of specialized computers working in parallel. Let’s get some idea of how much processing is involved in rendering a full screen.

My monitor is 1920x1200 pixels in size. I would like to have a frame rate of 60 frames per second. How many pixels must be computed each second to attain this rate?



[ ] pixels

Answer

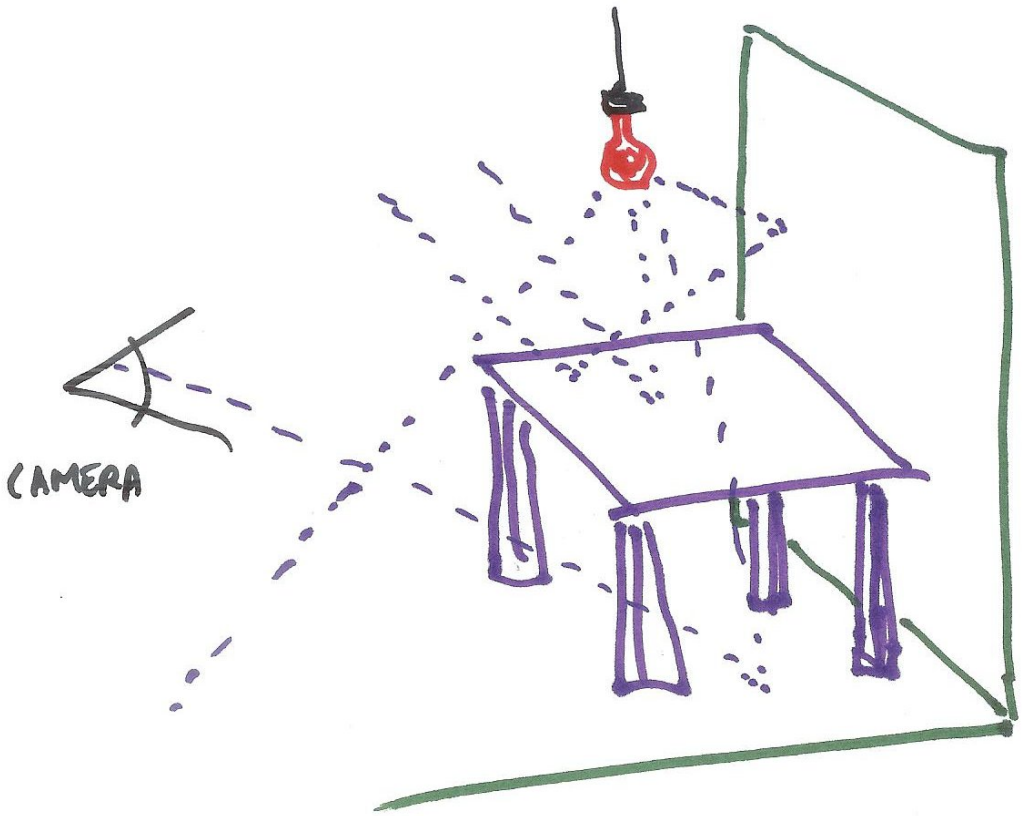
[138240000] One hundred and thirty eight million, 240 thousand pixels. By the way, the word “pixel” is short for the phrase “**picture element**”. Note that this value is the *minimum* number of pixels to compute.

This huge number helps explain why all computer made nowadays have some sort of 3D graphics acceleration hardware inside them. This can take the form of a $2 integrated chip on up to two or more graphics cards, each costing hundreds or even thousands of dollars.

Lesson: Light and Rendering

There are any number of ways to figure out how much light arrives at each pixel to form an image. In theory, you could fully simulate the world around you: photons start from a light source, hit atoms and are reflected, absorbed, re-emitted, and so on. You would put a camera in the scene and whatever very few photons actually happened to enter the camera’s lens would then determine the image.

[side-view picture of a light over a table, light hitting surfaces and so on, some reaching the eye. Draw rays last, in a separate layer, so we can reuse.]



As you might guess, doing so would take a huge amount of computation for even the most modest scene. A back of the envelope computation for a 40 watt lightbulb shows that it emits about 10 to the nineteenth photons per second.

[put 10^{19} on screen next to bulb.]

Question: How Many Computers?

To get a sense of how involved it would be to track all the photons in a scene, I want you to do a rough estimate. We have ten 40 watt light bulbs in a room. As mentioned before, each light bulb provides 10^{19} photons per second. About a billion computers, 10^9th, are currently in use world-wide. Say each computer can trace a million photons through a scene per second. This is an unrealistically high rate, since we're talking about tracing each photon until it is absorbed by an object or it reaches the camera. Your grandmother's computer likely cannot do this. How many Earths worth of computers would we need to trace all the photons in a scene and so form an image?

Ten 40 Watt bulbs
A light bulb emits about 10^{19} photons per second.
About a billion (10^9) computers in use world-wide.
Computer determines paths of a million photons per second.

How many Earths worth of computers are needed to compute all the photon paths in our room?

() We need just one Earth's worth of computers.
() We need ten Earths worth.
() We need one hundred thousand Earths worth.
() We need ten million Earths worth.

Answer

The answer is that we need 100,000 Earths worth of computers. I may even be off by a factor of 100,000, but you get the point.

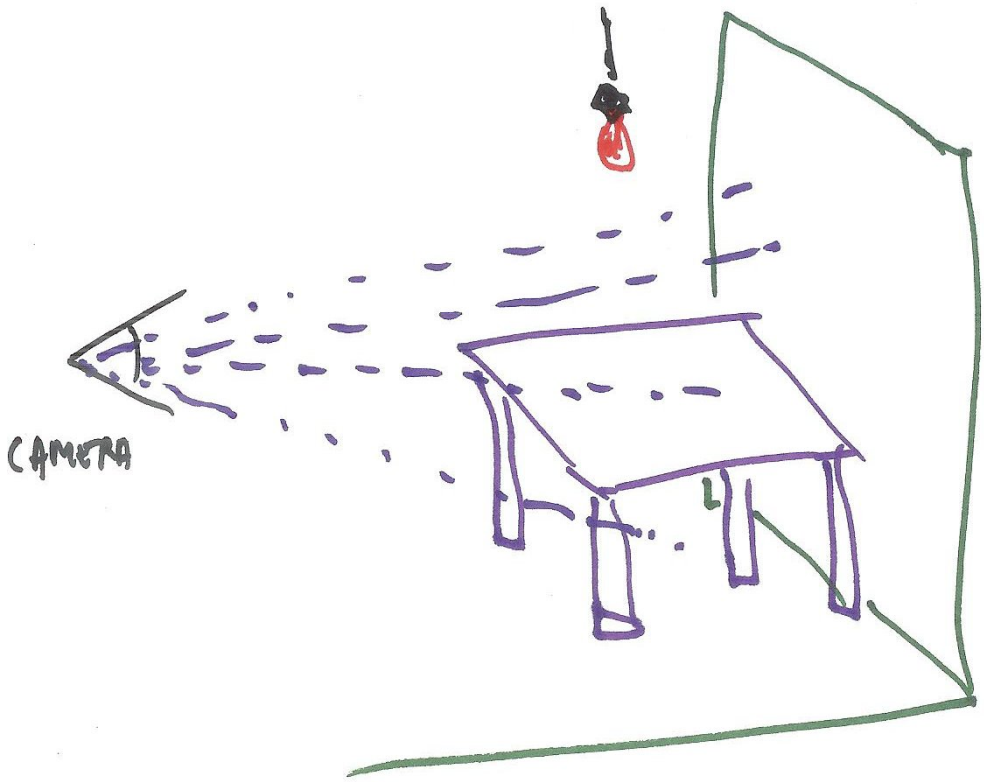
Tracking all those photons is ridiculously expensive. To make the problem more reasonable, *all* rendering algorithms make some simplifications. To have frames computed at interactive rates, we streamline the process even more.

[Instructor Comments: Here's a rough back of the envelope computation of photons per second, http://boards.straightdope.com/sdmb/showthread.php?t=336101]

Lesson: Reversing the Process

We've seen how expensive it would be to track all the photons in a scene. To avoid this level of computation, in computer graphics we make a number of simplifying assumptions.

First, only those photons that reach the camera are the one needed to make an image. So these are the only ones we would like to compute. We know which objects we can see. But, how do we know in advance which photons will matter? Each photon is generated by a light source but we don't know where each ends up. So, we reverse the process: instead of sending photons from the light, we essentially cast a ray from the eye through each pixel and see what is out there.

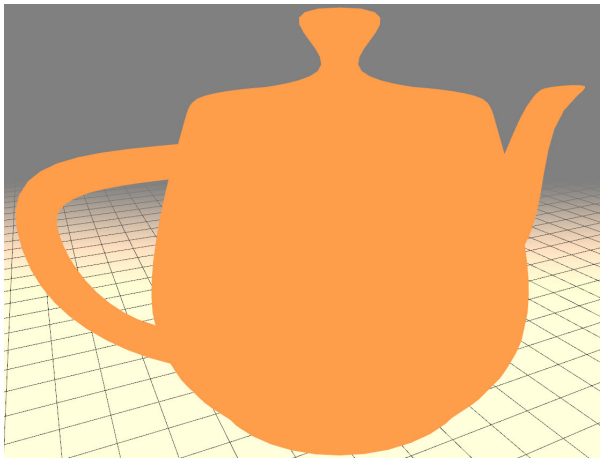


When a surface is seen at a pixel, we then compute the direct effect of each light on that surface. Add up all the light contributions and you have a reasonable approximation of what the surface looks like.

One more approximation can be used for these lights: nothing blocks them. Every surface is considered visible to a light, unless it faces away from that light source. In other words, no objects cast shadows.

Rendering can get even simpler than this. For example, you could do away with lights altogether and just draw each object with a solid color. However, I'm drawing the line there, as lights add considerable realism and perception of 3D to a scene.

[pictures without and with lighting. I could make these "fairer" by making each textured, for example.]



So, there are two extremes. At one end photons are carefully traced from light emitters through a scene, potentially giving a highly realistic image. At the other end the camera determines what gets viewed and a rough approximation of what light reaches each surface is determined. This process can be unrealistic, but can be computed extremely fast. We will start at this second extreme and in later lessons show how we can remove or modify various simplifications so that more photon-based computations are performed, such as casting shadows.

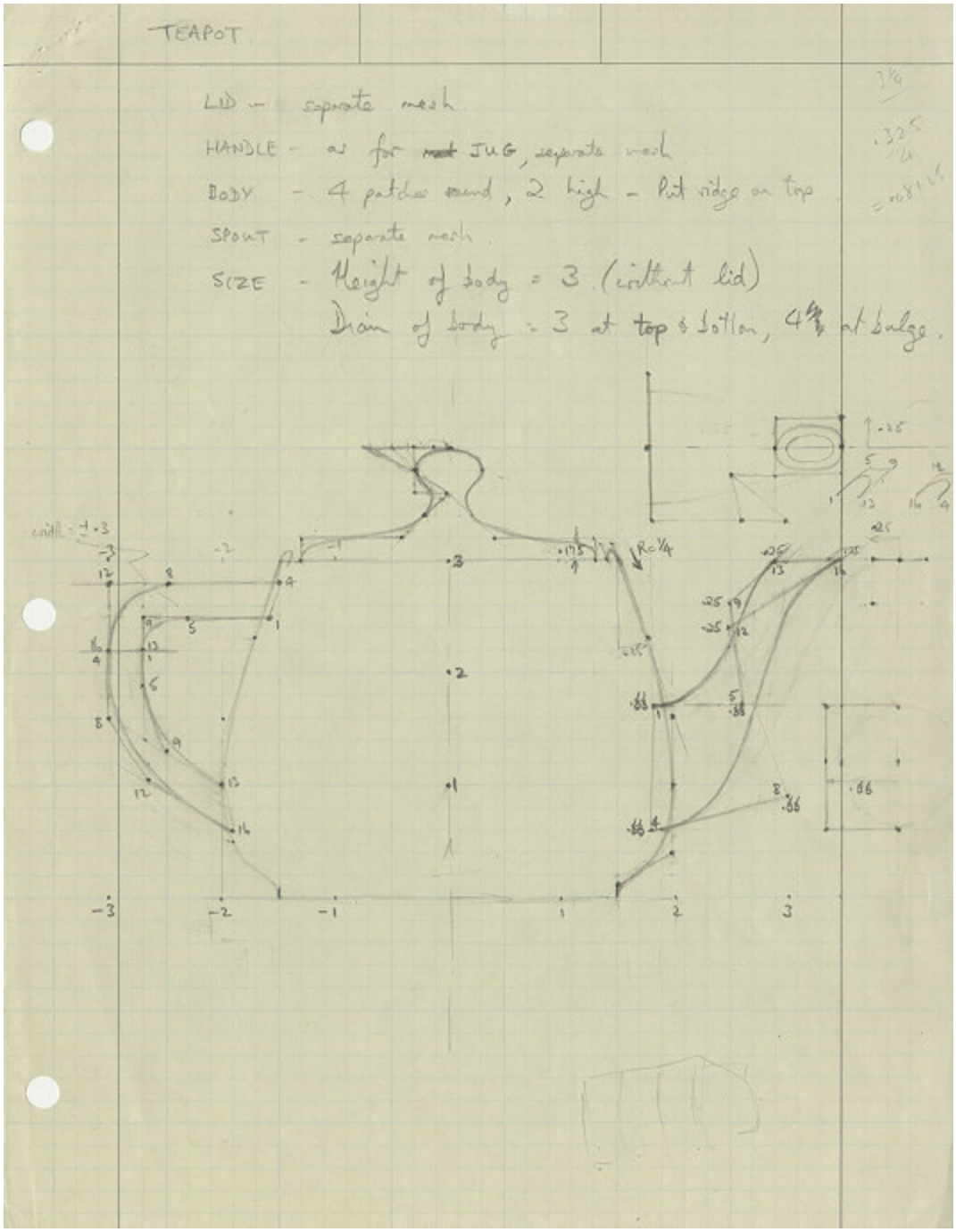
Lesson: The Teapot

You may have noticed there have been a few teapots in these lessons. I hope you like them, because you'll be seeing lots more of them as we go. The teapot is the classic demonstration object of 3D graphics. It's like "Hello World" for programming.

The teapot model was created by Martin Newell, one of the pioneers of computer graphics. Among other accomplishments, he co-developed an early rendering method called the Painter's Algorithm, and did seminal work in the area of procedural modeling. He also founded the company Ashlar, Inc., a computer-aided design software firm.

The model was inspired by a real-live Melitta teapot, manufactured in 1974. Back then there weren't many good models to use for rendering, especially for displaying materials. Objects made of cubes, cylinders, spheres, and so on only take you so far. In 1975 Martin Newell at the University of Utah was talking about this problem with his wife Sandra. They were just sitting down for tea, so she suggested that he model a tea set.

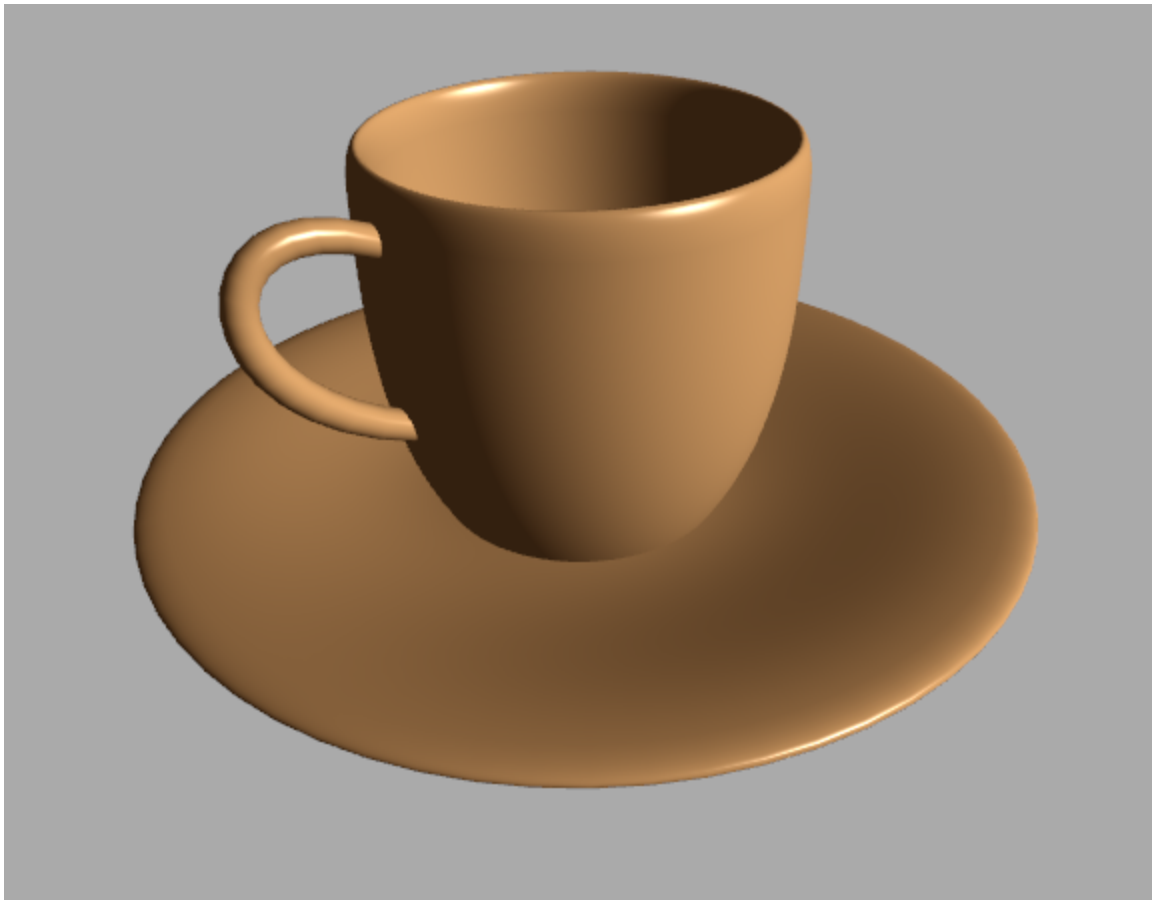
[Computer history museum,
http://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206]



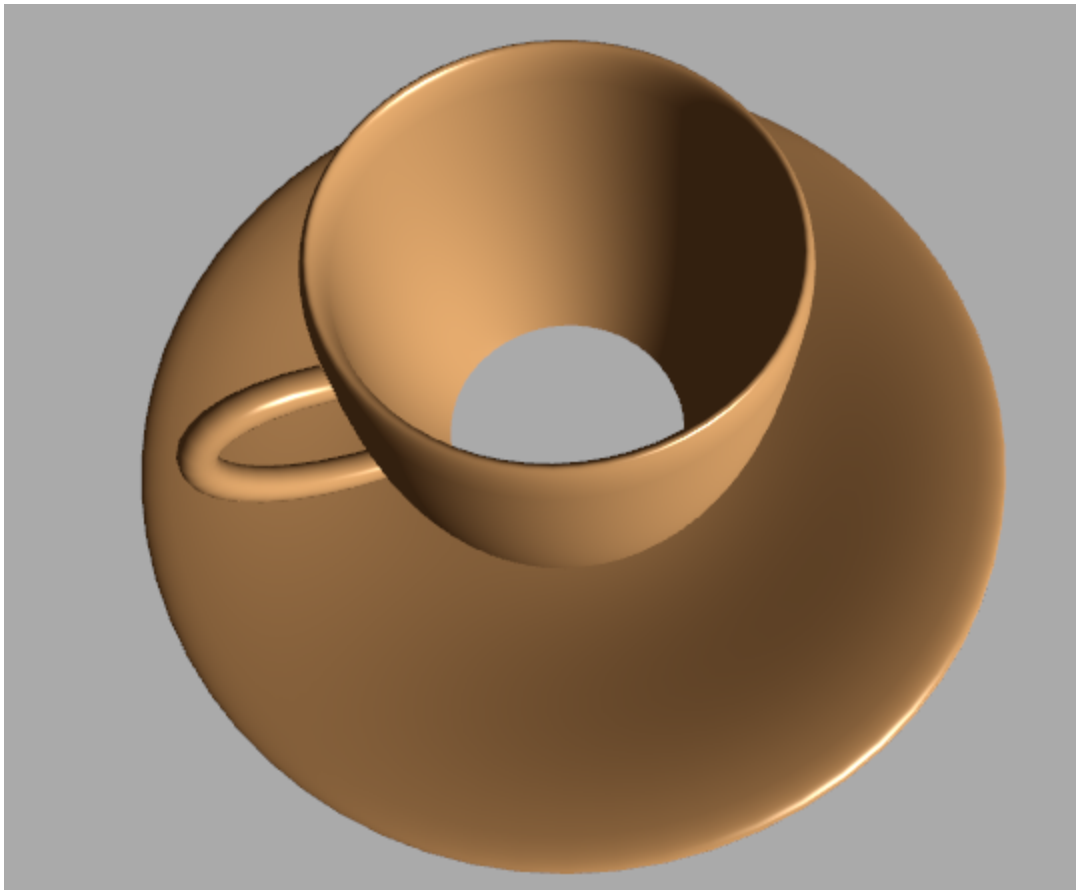
He made a sketch of the silhouette on graph paper. The drawing represented a recognizable profile of a teapot. It doesn't match the real-world teapot – the spout and the knob on the lid are a bit different, for example. He described these curves by using what are called cubic Bezier splines. A "spline" is a fancy word for a type of curve. These spline curves are formed by carefully placing a few points, called control points. [point to sketch] A surface is then formed by using spline curves in two different directions to define a patch - see the additional course

materials to learn more about this process. The original teapot is defined with just 28 patches. To render these patches interactively we often convert them into grids of triangles.

[show teacup demo]



This type of modeling is useful for creating all sorts of curved surfaces, such as car bodies. When Newell created the teapot, he also made some other models to create a whole set. Not half as well-known, the teacup and saucer make up a pretty nice little model. At least, they look good from this angle. Well, not great, though - there’s no shadow, so the cup looks like it’s floating over the saucer.



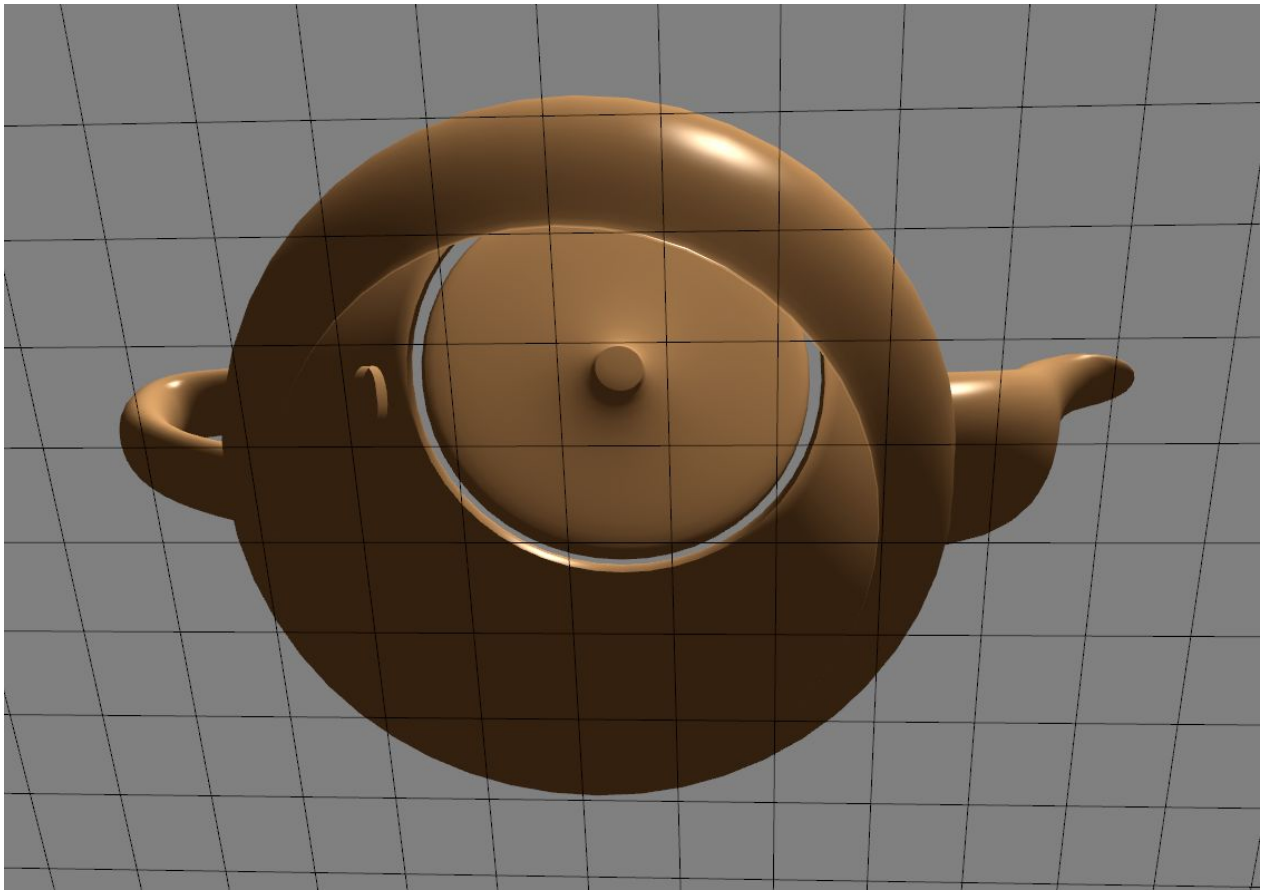
If you rotate the view, its secret is revealed: there’s a large hole in the middle of it.

[show teaspoon demo]

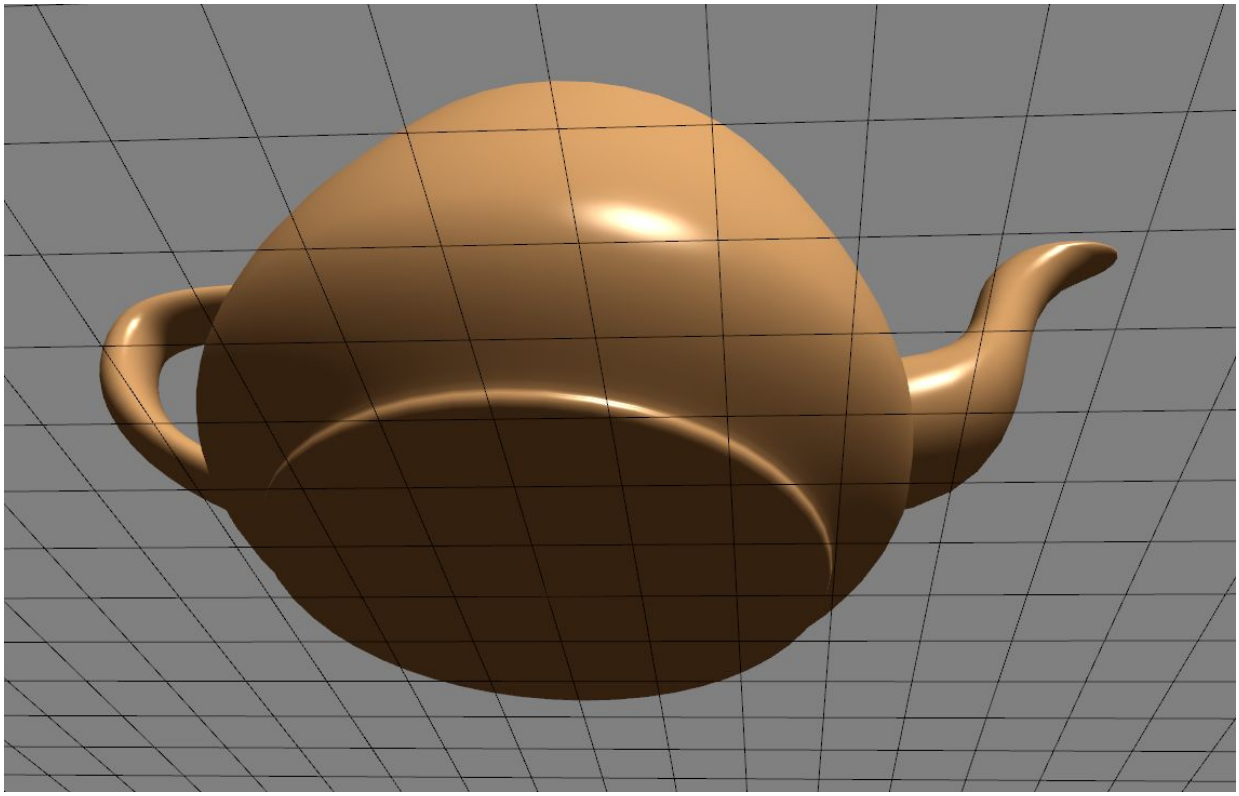
There's also a teaspoon model, which is pleasing from any view. I was surprised by how easy it was to display these models, once I had my teapot demo working. The text files of patch data for the teacup and teaspoon are available on the web. The time stamp on these files is from 1991. Digital data doesn't rot!

A nice feature of using a cubic Beziers for the data description is that the text files were each around just 8000 bytes. For a lot of applications nowadays, that's less than the size of an empty save file.

[My demo – show the bottomless teapot, then add bottom. Show scale on and off. Show gap and fitted lid. http://www.realtimerendering.com/teapot/ and
http://www.realtimerendering.com/udacity/?load=demo/unit1-teapot-demo.js
http://www.realtimerendering.com/udacity/?load=demo/unit1-teacup-demo.js
http://www.realtimerendering.com/udacity/?load=demo/unit1-teaspoon-demo.js]

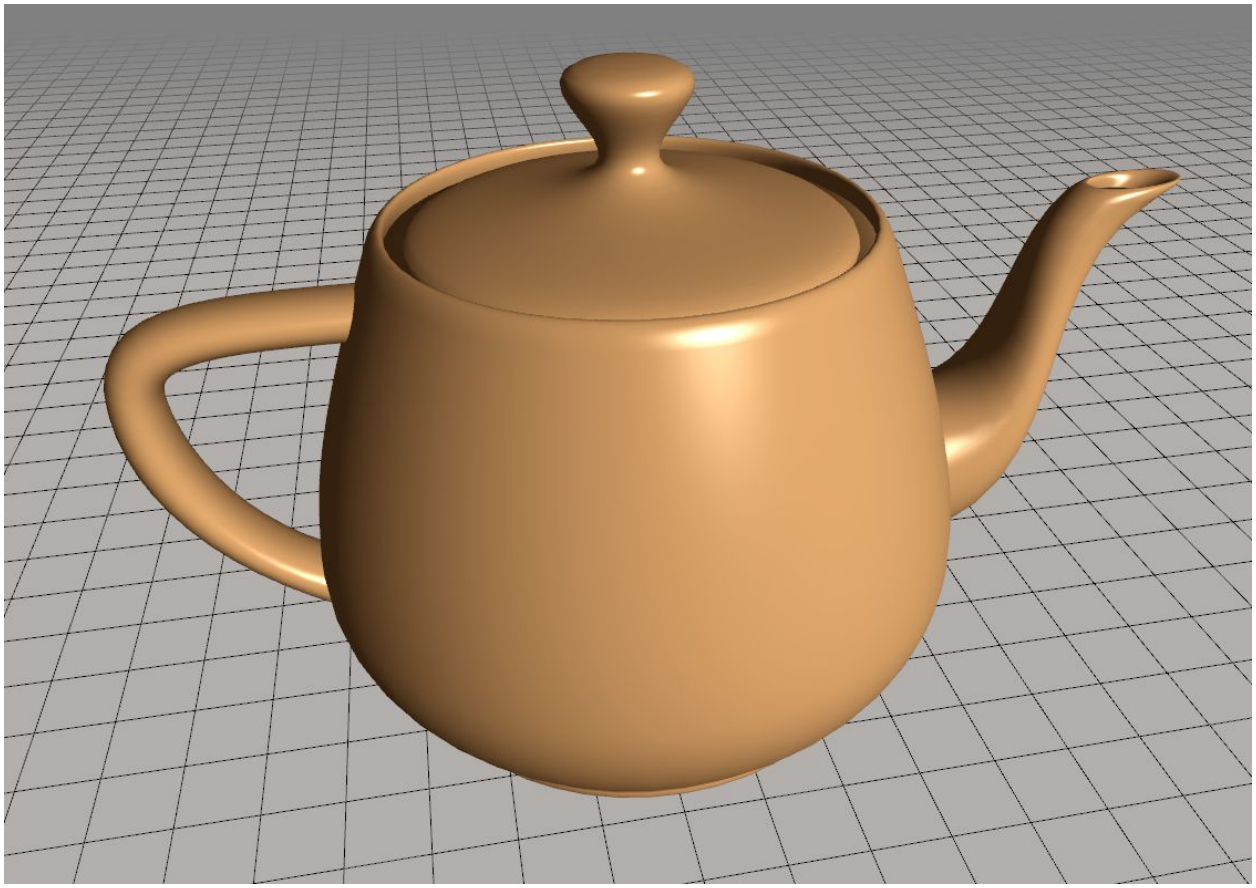


Let’s get back to the teapot. Like the teacup, the teapot also didn’t originally have a bottom. After all, a teapot is usually sitting on some tabletop, so doesn’t really need one most of the time. 4 patches defining a bottom were added later, by Frank Crow in 1987. Some graphics practitioners consider adding a bottom to be heresy, so be careful about what choices you make and whom you show your teapots.

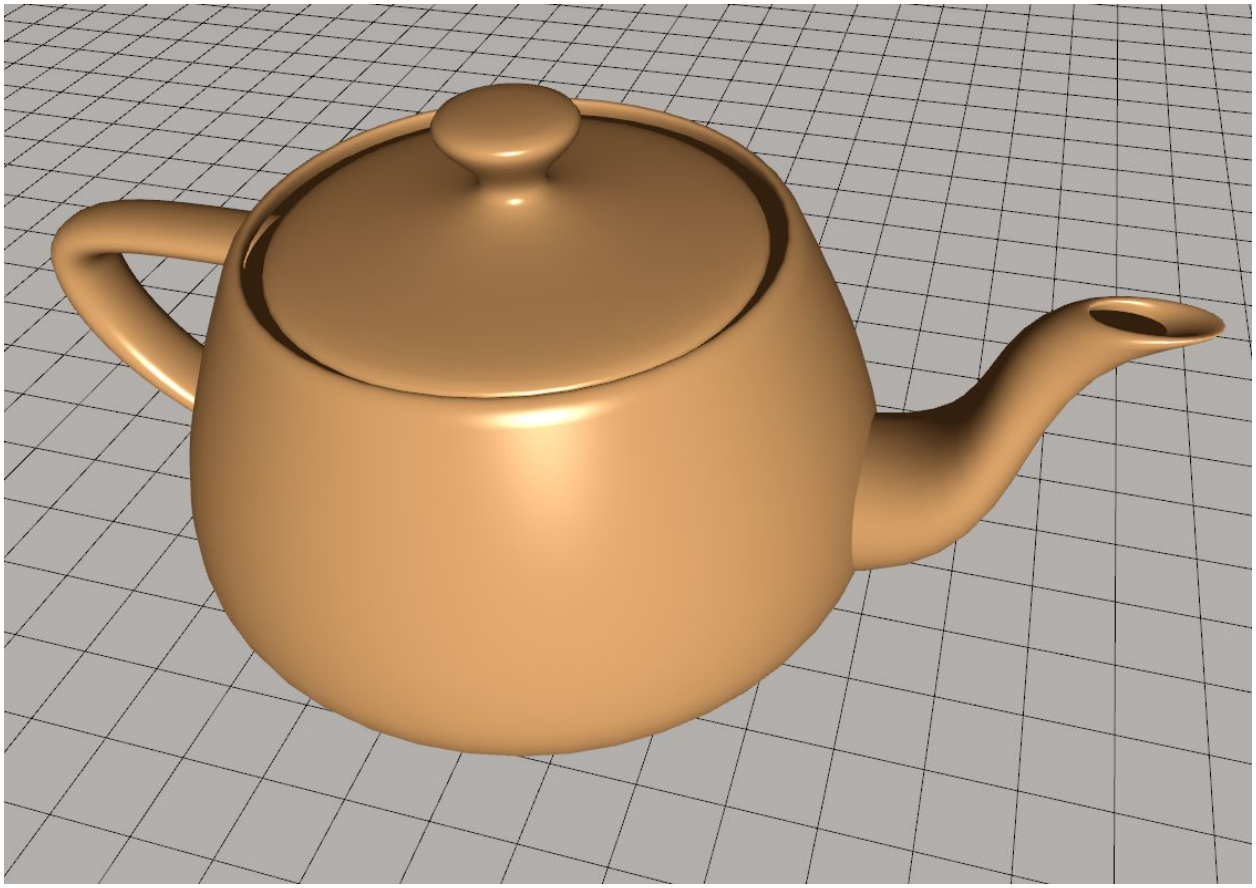


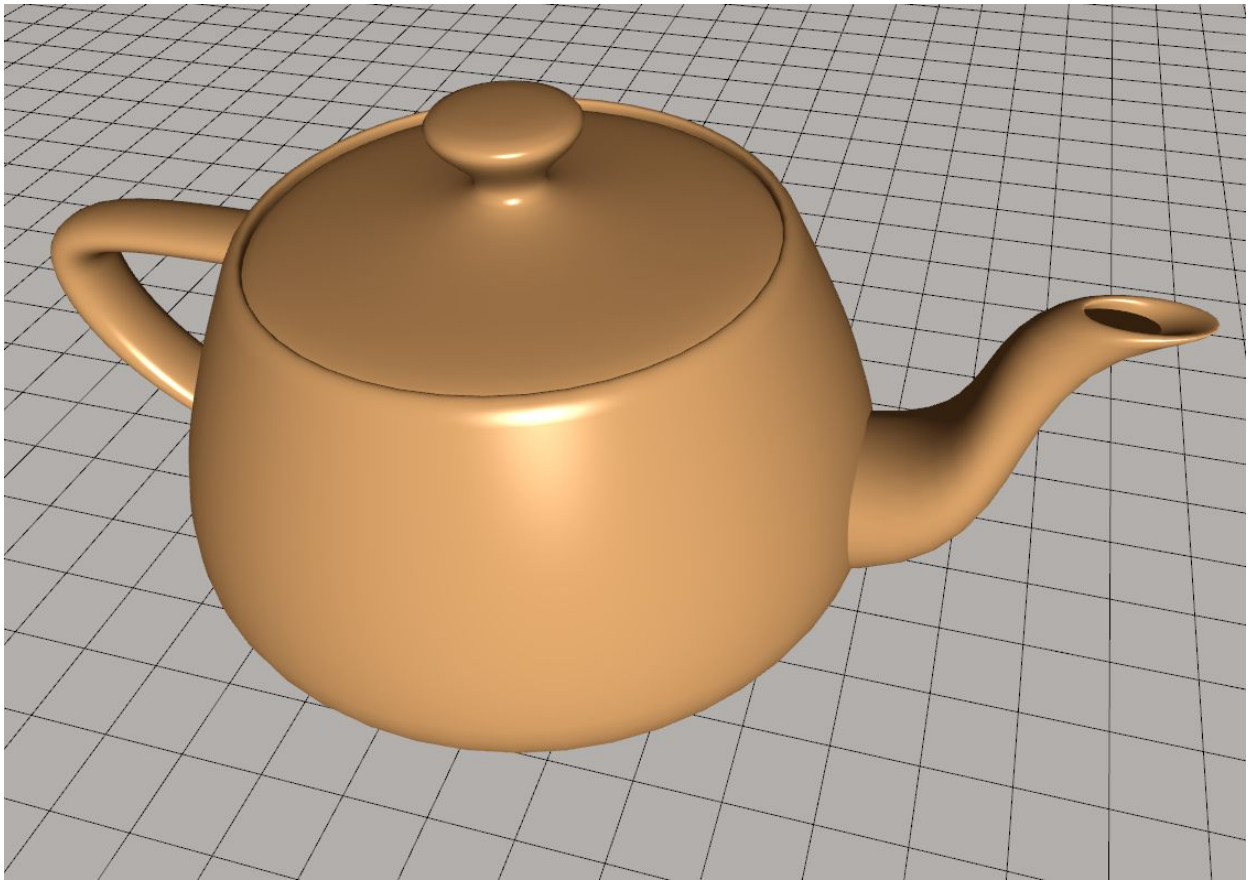
One other obvious difference between the real-world and virtual teapots is the vertical scale. Martin Newell shared his teapot dataset with other researchers. His original sketch's scale matches the real-world teapot, but the computer graphics model evolved to be squatter. I've seen two stories in print about how this happened - and being in print makes it true, right?

The more reputable one is that Jim Blinn, one of the early users of the teapot dataset, decided that the teapot simply looked more pleasing when squished. The other story I've seen is that the display Blinn was working on didn't have square pixels. Rather than mess with the camera's transform, he decided to scale down the teapot vertically to compensate. I like this version a lot, since Blinn has a well-known quote of, "all it takes is for the rendered image to look right". I asked Martin Newell about these two stories and he thinks the non-square pixels may have been the original reason, but that Blinn's aesthetic sense could well have kept the model that way. Martin himself prefers the squat version.



Speaking of looking right, there’s a noticeable gap between the lid and the body of the teapot. I’ve taken the liberty in this course of widening the lid a bit so that this gap is not noticeable. Shhhh, don’t tell anyone.





Blinn's use of the teapot in his research popularized it and the size change stuck. The teapot is a great object to render for a number of reasons. First and foremost, we have a good mental model of what the teapot *should* look like, so can easily identify bugs. The curvature of the surface varies considerably, especially on the spout and handle, causing the shading to change in interesting ways.

[change tessellation value to be lower, and show blocky version]

Some of the spline patches form a dense set of triangles, others less so, which gives some rendering algorithms a workout.

[SPD image]

Since the teapot is not some simple convex form such as a block or cylinder, it casts a shadow on itself – another good test case. My small contribution to the cause was an open-source program that generated the teapot. This was back in 1987, long before the term open-source existed. I ported this code to three.js for this course.

[Teapotahedron]

[Cleared rights with Erin Shaw and Dave Kirk]

Many people consider the teapot a standard model. Back in 1987 Jim Arvo created a classic image showing the teapot as the sixth Platonic solid, next to the tetrahedron, cube, and so on. Really, I've seen a lot more teapots rendered than I have dodecahedra.

[http://en.wikipedia.org/wiki/File:Original_Utah_Teapot.jpg]

[Nice to end with the real teapot, then have real Pixar teapots walk in]

The teapot is an iconic object that is a symbol of 3D computer graphics. There are other famous models out there – the VW bug, the Stanford bunny, the Sponza palace, happy Buddha, the dragon – but the teapot is by far the most celebrated. Keep an eye out for it in the first “Toy Story” movie and in the Simpson’s 3D episode. If you want to see the real teapot in all its glory, it’s on display at the Computer History Museum in Mountain View, California. Now that you’re aware of the teapot, you’ll start to notice it popping up when you least expect it.

[An article by Frank Crow about the teapot: http://origin-www-ca.computer.org/csdl/mags/cg/1987/01/mcg1987010008.pdf SJ Baker has a good teapot history page http://www.sjbaker.org/wiki/index.php?title=The_History_of_The_Teapot. Wikipedia also has some additional facts and stories: http://en.wikipedia.org/wiki/Utah_teapot Some interesting images: sketch of the teapot, http://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206

A good overview of the iconic images of computer graphics is here:

http://design.osu.edu/carlson/history/lesson20.html. Various well-known models can be downloaded: http://graphics.stanford.edu/data/3Dscanrep/ has the Stanford bunny, happy buddha, and dragon, http://graphics.cs.williams.edu/data/meshes.xml has various forms of the Sponza palace. The teapot program I wrote is at http://tog.acm.org/resources/SPD/ . You can see the VW bug model and others on this quirky timeline: http://sophia.javeriana.edu.co/~ochavarr/computer_graphics_history/historia/

The teapotahedron even makes it into demoscene programs: http://youtu.be/3PXz421gXTQ

The walking teapots at the end have their own webpage http://renderman.pixar.com/products/tools/renderman-teapots.html and Facebook fan club https://www.facebook.com/pages/Pixars-RenderMan-Walking-Teapot-Official-Fan-Club/114826081870108]

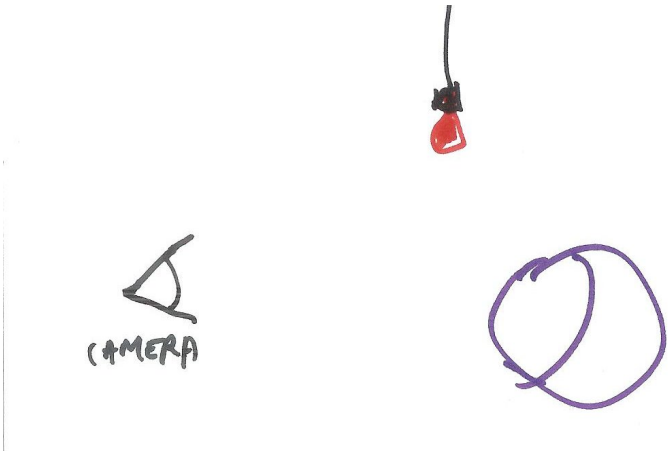
[image rights should be given in supplemental materials - may use more of these...
http://commons.wikimedia.org/wiki/File:Utah_teapot_simple_2.png
http://commons.wikimedia.org/wiki/File:Celshading_teapot_large.png
http://commons.wikimedia.org/wiki/File:Original_Utah_Teapot.jpg
http://commons.wikimedia.org/wiki/File:Utah_teapot_3dsmax.png
http://commons.wikimedia.org/wiki/File:Melitta_teapot.png

Additional sources, may not be cleared:

http://excelsior.biosci.ohio-state.edu/~carlson/history/images/teapot8000.jpg
http://excelsior.biosci.ohio-state.edu/~carlson/history/images/utah_teapot.jpg - heh, my own teapot image is in this photo.
]

Question: Simple Materials

Say you render an image by the extreme of finding what object is visible at each pixel and then finding how a light affects the surface.
[draw a sphere, light, and camera - there's a reason I've made this small.]



Which of the following objects can be rendered with this simple system? Here's a hint: exactly two answers are correct.

[image http://commons.wikimedia.org/wiki/File:Ball3_1_2.jpg and
http://commons.wikimedia.org/wiki/File:Caustics.jpg and
http://commons.wikimedia.org/wiki/File:Incandescent_light_bulb_on_db.jpg]

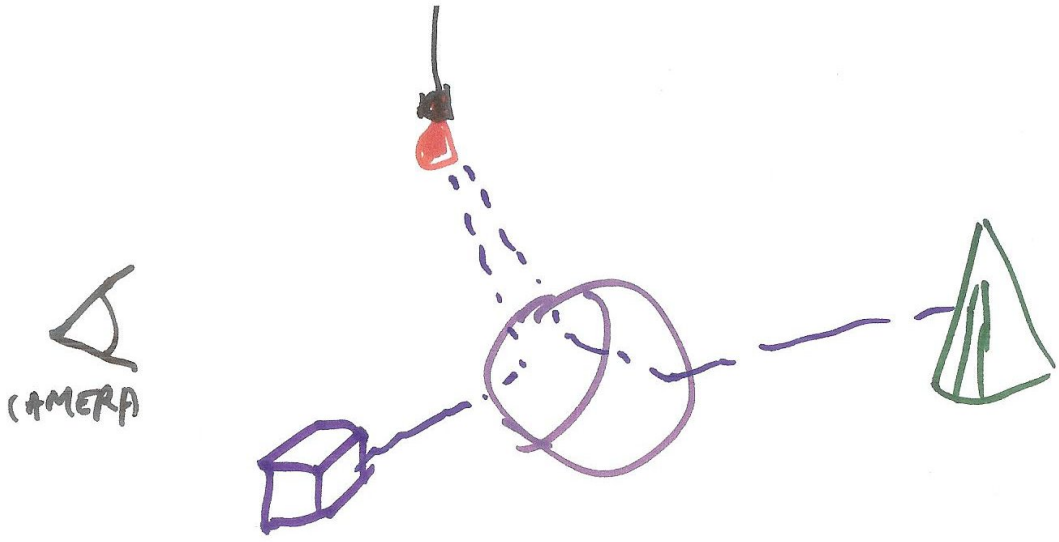
The sphere could be made to look like:
[ ] a polished metal ball, reflecting everything around it.
[ ] unpolished wood, showing a grain.
[ ] it is a light bulb.
[ ] it is made of glass, refracting light through it.

Answer

The two correct answers are that you could make a sphere look like wood or even a light bulb. In both cases you just need the sphere itself and, optionally, a light source to shade it. For the light

bulb you might do nothing at all, just filling in the sphere as bright white.

The problem with a highly reflective ball or a crystal glass material is that you need the sphere to reflect or refract light from elsewhere in order to give any sort of convincing effect.

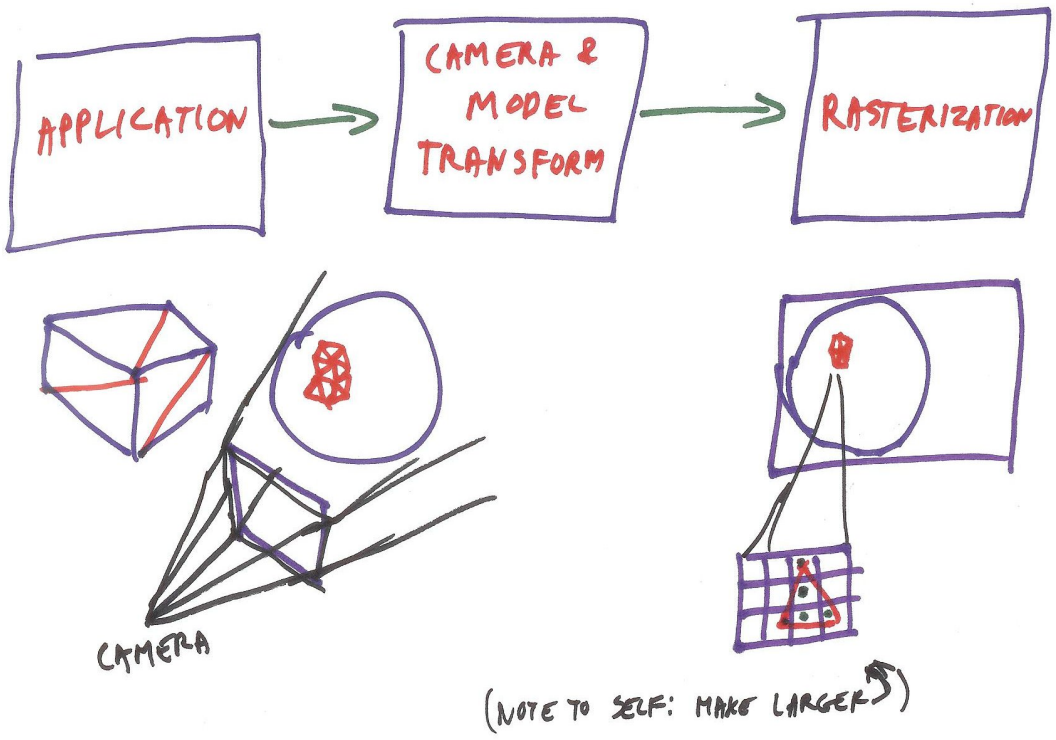


There are ways to create such effects using additional information about the scene, which we'll deal with in future lessons. For now, try out the WebGL demo in the Instructor Comments to see a variety of materials applied to a set of spheres.

Lesson: A Jog Down the Pipeline

We now have some sense of how we can make lighting calculations not take an incredible amount of time. In this lesson we're going to take an extremely high level view of how objects get rendered by a graphics accelerator. This graphics hardware is usually called the **GPU**, for **Graphics Processing Unit**. A GPU uses a rendering process called "rasterization" or "scan conversion", which is optimized to use the simplifications I've outlined.

Let's look at one rendering pipeline from end to end:
[Pipeline here: application -> camera and model transform -> rasterization]



The idea of the pipeline is to treat each object separately. The first step is simply that the application sends objects to the GPU. What objects? Typically, 3D triangles! Each triangle is defined by the locations of its three points.

An application converts a cube into a few triangles. A sphere gets turned into a bunch of triangles. So, in the first step the application decides what triangles to send down the pipeline.
[draw a cube and ball getting triangulated - how do I give instructions that I'd like the video to show me actually drawing the triangulations, not just the final results?]

In the second stage of the pipeline these triangles are modified by the camera's view of the world, along with whatever modeling transform is applied. A modeling transform is a way to modify the location, orientation, and even the size of a part. For example, if you were rendering a bouncing ball, the ball's modeling transform each frame would move the ball to a different location along its path. The effect of the camera view is clear enough: after the object is moved to its location for the frame, is it still in view of the camera? That is, is the object inside the view frustum? If not, then we're done with this object, since it won't affect any pixels on the screen. In this case we say the object was ***fully clipped***. So here the cube is fully clipped.
[view frustum drawn, viewing ball]

The camera and modeling transforms compute the location of each triangle on the screen. If the triangle is partially or fully inside the frustum, the three points of the triangle on the screen are then used in a process called *rasterization*.

[draw pixel grid and show triangle on it.]

This process identifies all the pixels whose centers are inside the triangle. In other words, it fills in the triangle. These locations then are used to show the image of the triangle on the screen.

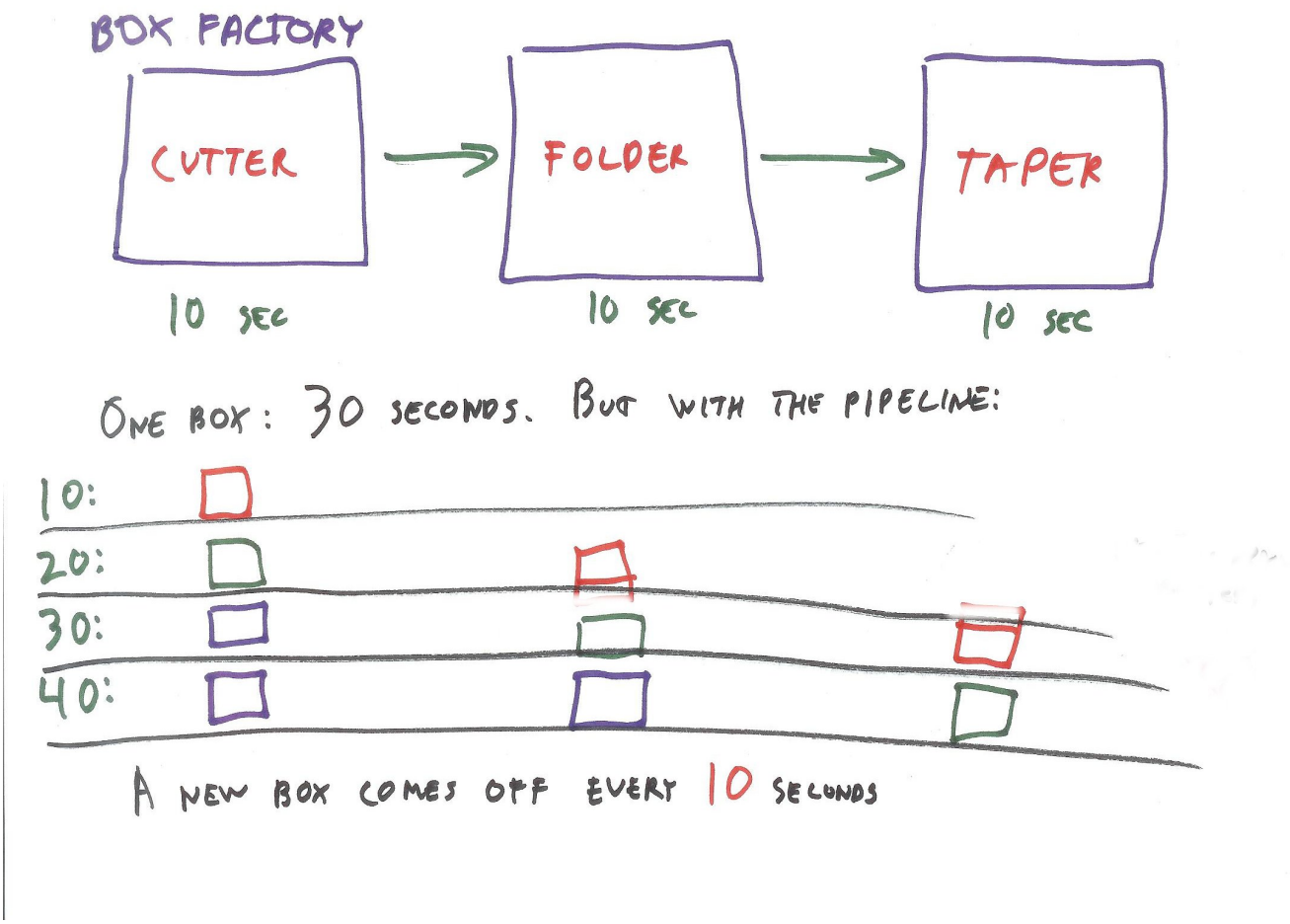
[Instructor Comment: A book I helped write covers the pipeline in depth, and the chapter is free online here: http://books.google.com/ebooks/reader?id=V1k1V9Ra1FoC&printsec=frontcover&output=reader&source=gbs_atb&pg=GBS.PA11 . For an even more thorough walk down the pipeline, read this set of articles, http://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/]

Lesson: Pipeline Parallelism

The idea of a pipeline is that every object is dealt with once. Say we have a list of 52 parts of a car, each part represented by a set of triangles. Each of these parts is sent down the graphics pipeline and is rendered to the screen. Once an object goes through this process we're done with it for the frame. If we stopped the process halfway through and looked at the image formed, we would see only the first 26 car parts in the image.

The advantage of a pipeline is that each part of the pipeline can be working on a separate task at the same time. Say we take a trip to the local cardboard box factory. There are three work areas: the box cutter, the box folder, and the box taper.

[cutter -> folder -> taper]



I won't describe what each step is, because it's a box factory, that's pretty boring! All you need to know is that each step takes 10 seconds. If you want a single box, it takes 30 seconds from start to finish to make it. However, if you want a lot of boxes, a new box will arrive at the end of the assembly line every 10 seconds.

[show timeline of boxes, red, blue, green and what time it is]

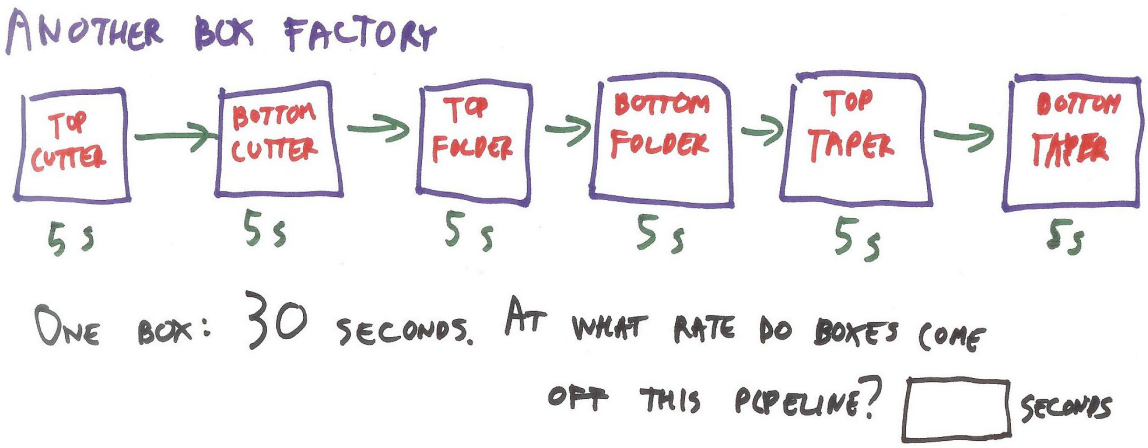
The red box gets cut, that's 10 seconds
It goes on to the folder station and a green box follows behind and gets cut at the same time, 20 seconds.
In the next ten seconds the red box is taped, the green box is folded, and the blue is cut; 30 seconds.
After this point, you can see another box will come off the assembly line every 10 seconds.

Question

Our previous pipeline has just three stages. Say we are able to split each of these stages in half, and each takes 5 seconds.

[top cutter -> bottom cutter -> top folder -> bottom folder -> top taper -> bottom taper]

The time to make a single box is still 30 seconds. At what rate do boxes come off this pipeline?

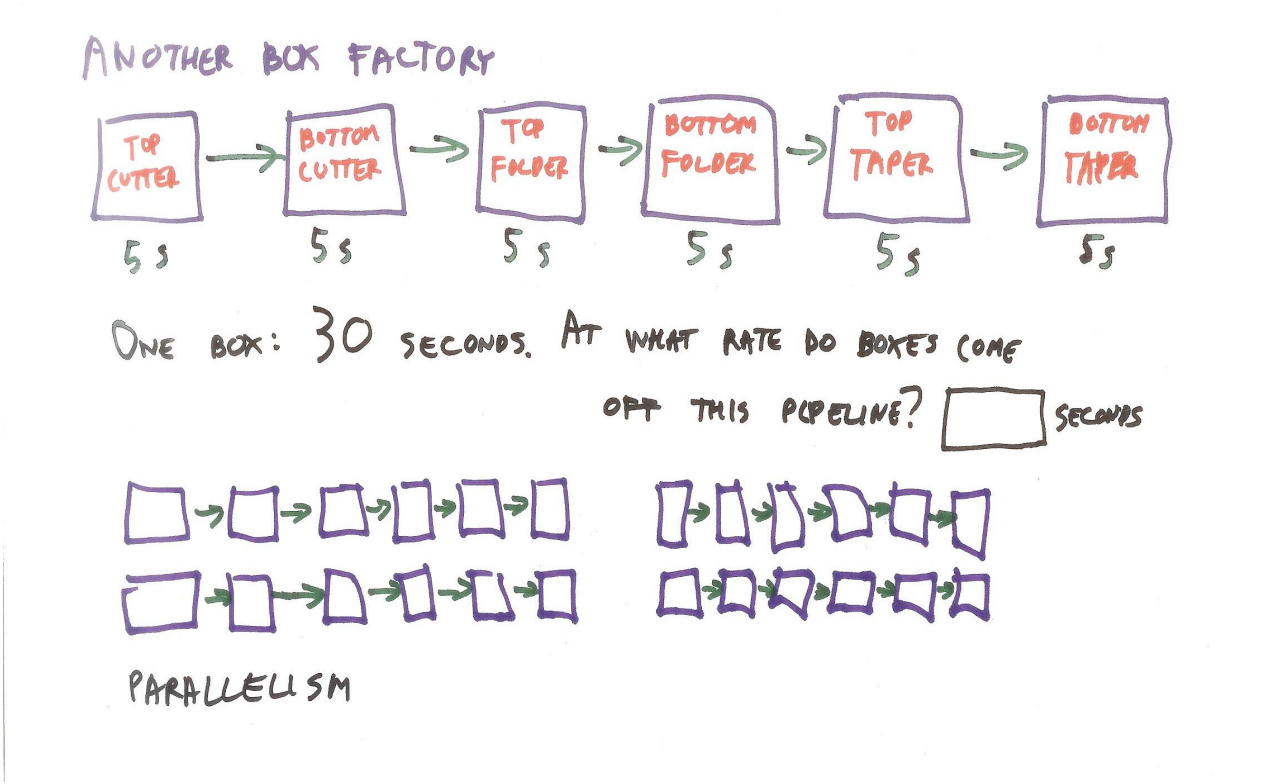


[ ] seconds

Answer

Boxes will come off the assembly line at a rate of one every 5 seconds! This is because we have 6 separate stages, all of which can operate at the same time.

We can of course produce even more boxes by buying more factory space and creating more assembly lines.
[draw more pipelines in parallel]

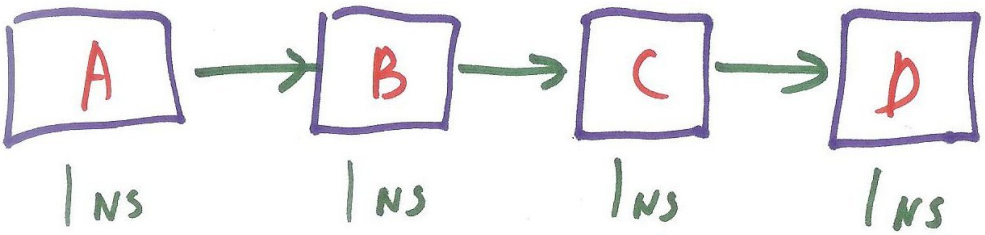


GPUs are designed using these techniques of pipelines and parallelism to give extremely fast rendering speeds.

[Instructor Comments: I should note that what I'm describing is a typical desktop or laptop computer's GPU. Portable devices such as smart phones and tablets will usually use tile-based rendering instead. There's a brief explanation of this algorithm here, http://www.onepc.net/reviews/0026/page2.shtml. The good news is that even this type of architecture can still be controlled by WebGL.]

Lesson: Bottleneck

In a perfect world every step of the pipeline would take the same amount of time and all stations would always be busy. In reality, there is always some part of the pipeline that is the bottleneck at any given moment. For example, say you have a pipeline of four stages:
[boxes A B C D]

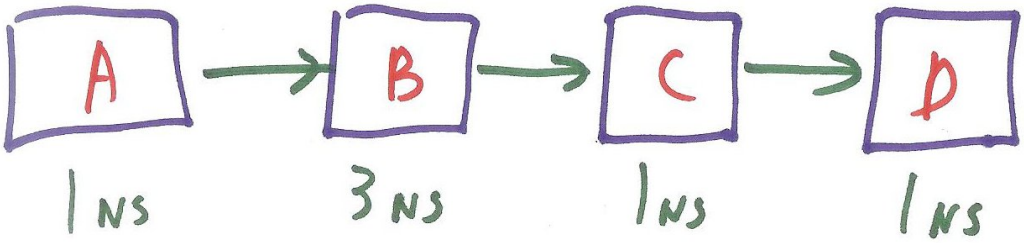


Normally each stage takes, say, a nanosecond. This means an object will take 4 nanoseconds to travel through the pipeline, and the rate of the full pipeline will be one object being produced every nanosecond.

Question

I'm going to let you figure this one out. What happens if stage B suddenly starts to take 3 nanoseconds to do its function? I'll give you the easy answer: an object will take a total of 1-(2,3,4)-5-6 - 6 seconds to travel through the pipeline.

The harder question is: how often will an object come out of the pipeline being fed with objects? Our original pipeline had one object per nanosecond. Think through how often objects will come out of this pipeline, once it has settled down under these new conditions.



An object will come off the pipeline every [ ] nanosecond(s).

Answer

The answer is **three nanoseconds**. Imagine you have objects entering each stage in the pipeline at the same time and suddenly stage B takes 3 nanoseconds. After 1 nanosecond the object entering stage A is done processing, but stage B is still busy. The object coming out of stage A is blocked and has to wait. When the pipeline is blocked this way, a **stall** has occurred.

With stages C and D the objects move on through just fine. However, the object in stage B does not enter stage C. Stage C is said to be **starved** at this point. It is available to do work but nothing is being sent to it.

[In the original figure above, show objects entering the pipeline and getting stalled, etc. I'm game to get tips from Katy on how best to do this effect.]

If we continue this analysis, we'll see that the pipeline moves only as fast as its slowest stage. In this case it's stage B. Clearly, if an object takes three nanoseconds to move through a single stage, it cannot come out any faster than this out of the pipeline as a whole. The other, faster stages don't slow things down, they simply wait for stage B to finish its work.

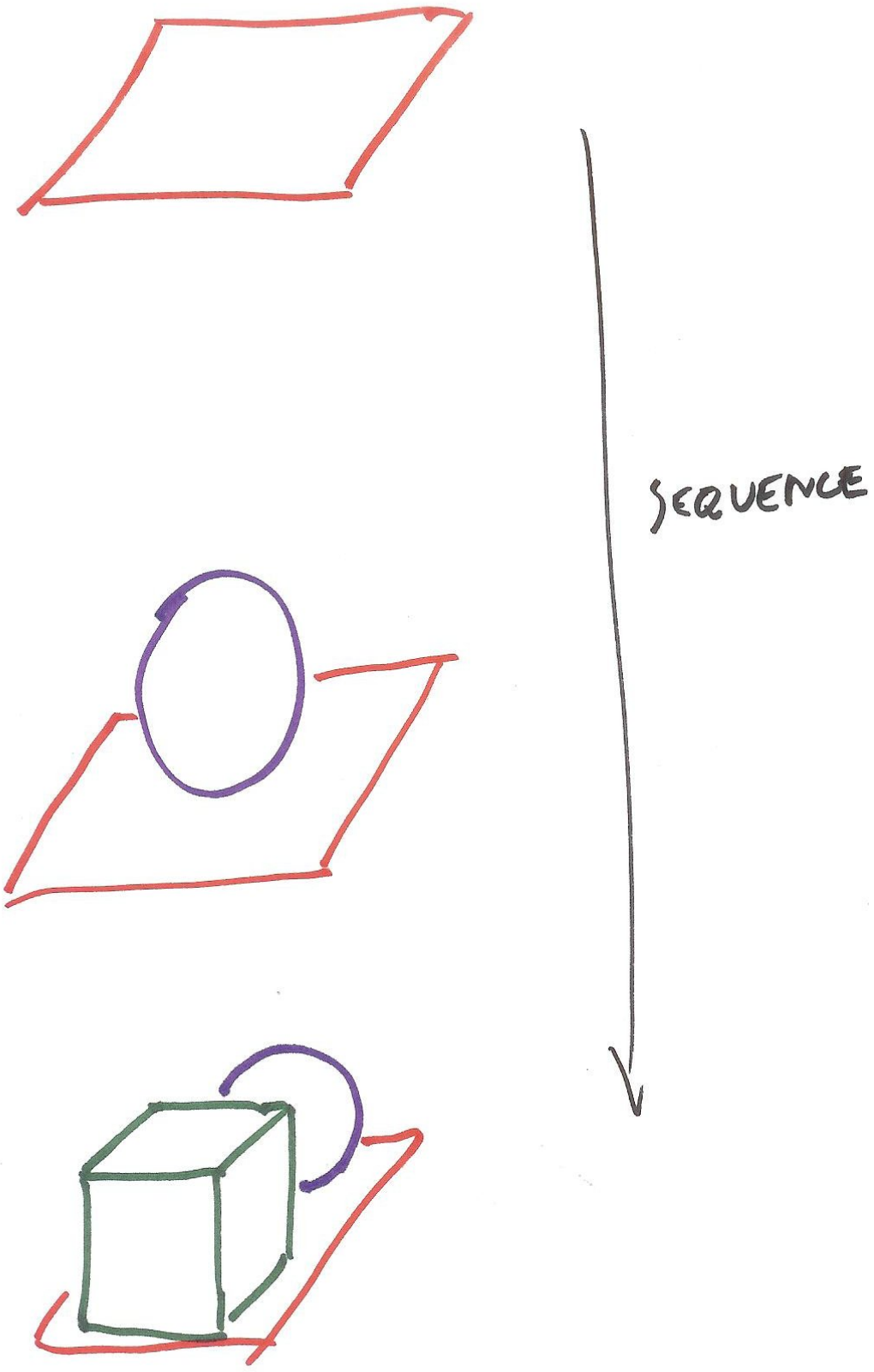
With the rendering pipeline, these same sorts of principles apply. Having the application run efficiently and keep the GPU fed with data is a common bottleneck, for example. The bottleneck will change over time, with sometimes one stage being the slowest, sometimes another. GPU designers use different techniques to perform load balancing, such as first-in-first out queues and unified shaders. We won't be describing these techniques in this course, but it's good to be aware that the GPU has a lot going on inside it. Various performance debugging tools let you see what parts of the pipeline are bottlenecks and so can help you improve the performance of your application.

Lesson: Painter's Algorithm

With our simplified pipeline, the application sends down triangles to be transformed to screen locations and then filled in. What happens if two triangles overlap on the screen? How is it decided which one is visible?

One way to render is what is called the *Painter's Algorithm*. The idea is to draw each object, one on top of the other. To make this work, you sort objects based on their distance from the camera, back to front. You the render the most distant object first, render the next closest object, and so on. The closest object is drawn last, so it covers up the objects behind it. This isn't really how most painters actually paint, but it could work in theory for, say, oil paints. I guess it makes some sense for computers to use it, since it's a somewhat mindless way to paint a painting.

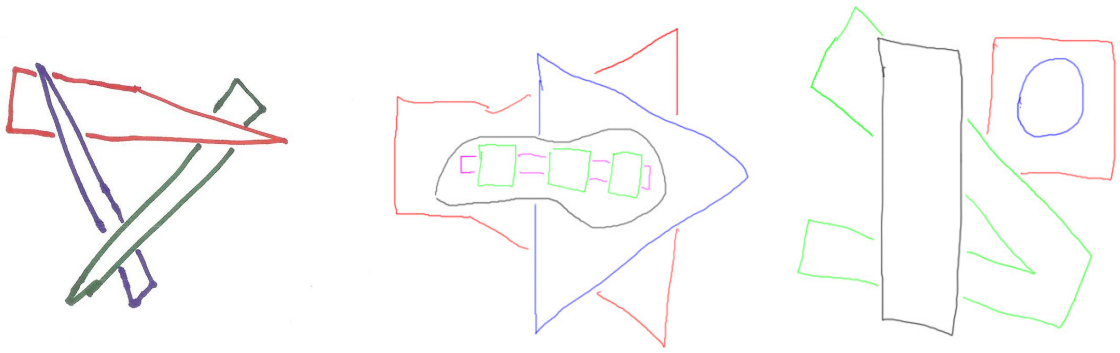
[Draw some objects, say a plane, cube, cone, and sphere, each in front of the next. **Save final drawing with separate layers, for reuse with Z-buffer!**]


SEQUENCE

Question

It turns out the Painter's Algorithm has a serious flaw. Below are three drawings. Which of these three scenes cannot be rendered by painting one object on top of the other?

[Add checkboxes (not radio buttons) under each. Make the first drawing **SECOND** so that a bit more thought is needed.]



Answer

The second drawing cannot be drawn properly. The problem with the Painter's algorithm is that you can get into cycles, where each object is in front of the next. Here, the red triangle is in front of the green, which is in front of the blue, which is in front of the red. There's no way to fully draw these objects in any order and produce the proper image.

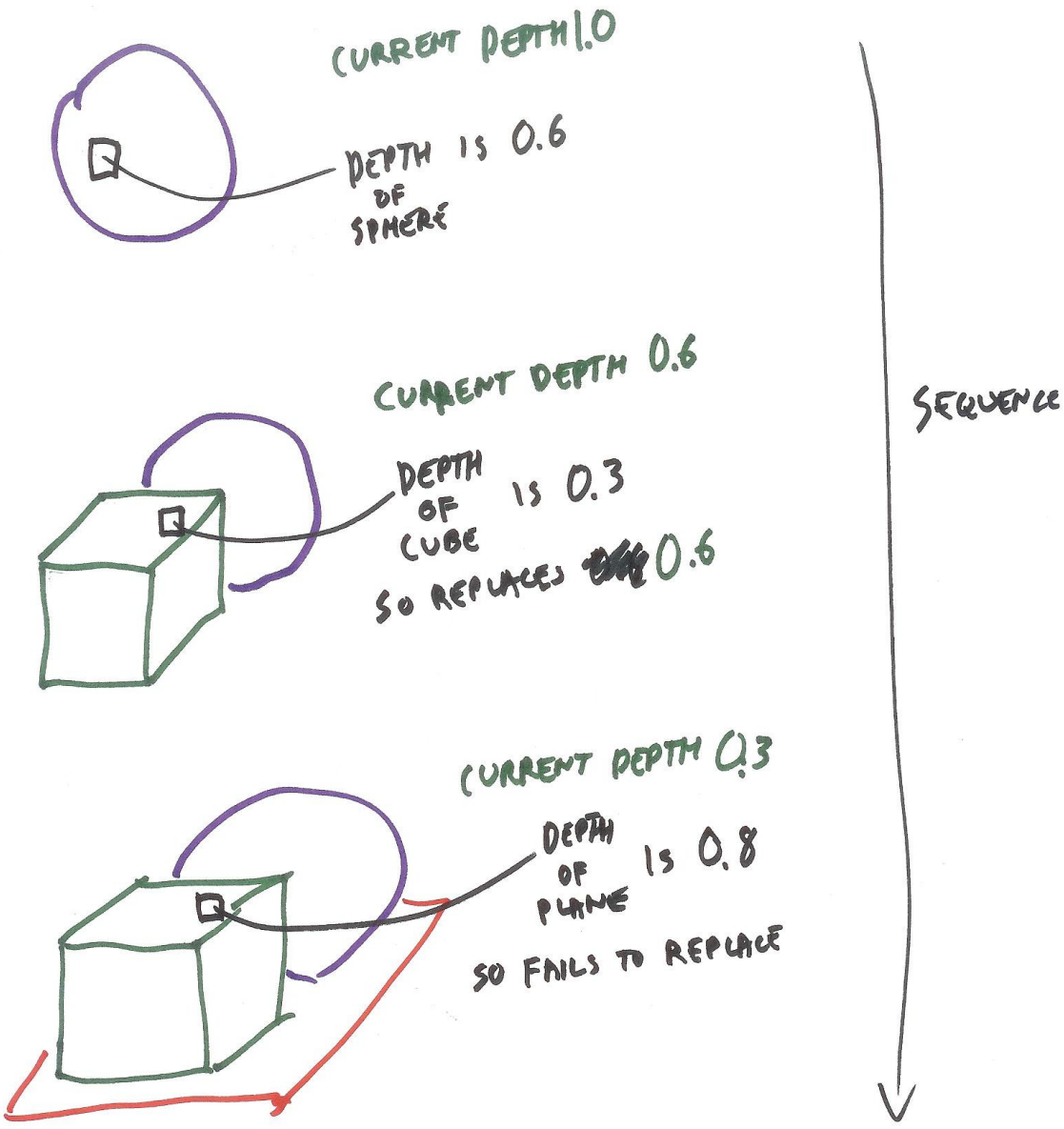
Lesson: Z-Buffer

The way the GPU usually solves the visibility problem is by using what is called a ***Z-buffer***. Or, if you have British leanings, it's pronounced *zed-buffer*.

The artist Lennart Anderson once said, **"Landscape painting is really just a box of air with little marks in it telling you how far back in that air things are."** This is a pretty good description of the Z-buffer algorithm. The "Z" in the Z-buffer stands for distance from the camera. "Buffer" is another word for a data array, an image.

So, for the image, in addition to storing a color at each pixel we also store a distance, which is called the z-depth. The z-depth is often considered as a floating point number that ranges from 0.0, meaning close to the eye, to 1.0, meaning the maximum distance from the eye. The Z-buffer is initially cleared by filling it with 1's.

[Draw objects again, this time using the Z-buffer idea. Also track the depths.]



When an object is rendered into an image, at each pixel the distance from the object to the camera is used to compute its z-depth, a value from 0 to 1. This value is then checked against the current value stored in the Z-buffer. If the object's distance is lower than the z-depth value stored, the object is closer to the camera and so the color of the object is then stored in the image's color buffer. If the object's distance is greater, the pixel's color is not touched. The Z-buffer works by keeping track of which object is closest to the camera at each pixel. In the end,

the closest object's color at each pixel is stored and so is visible in the final image.

Question

Say it costs one cycle to read the z-depth in a pixel in the z-buffer, one cycle to write a new value into the z-buffer, and one cycle to write a new color into the image itself. Ignoring all other costs, what is the fastest way to draw objects?

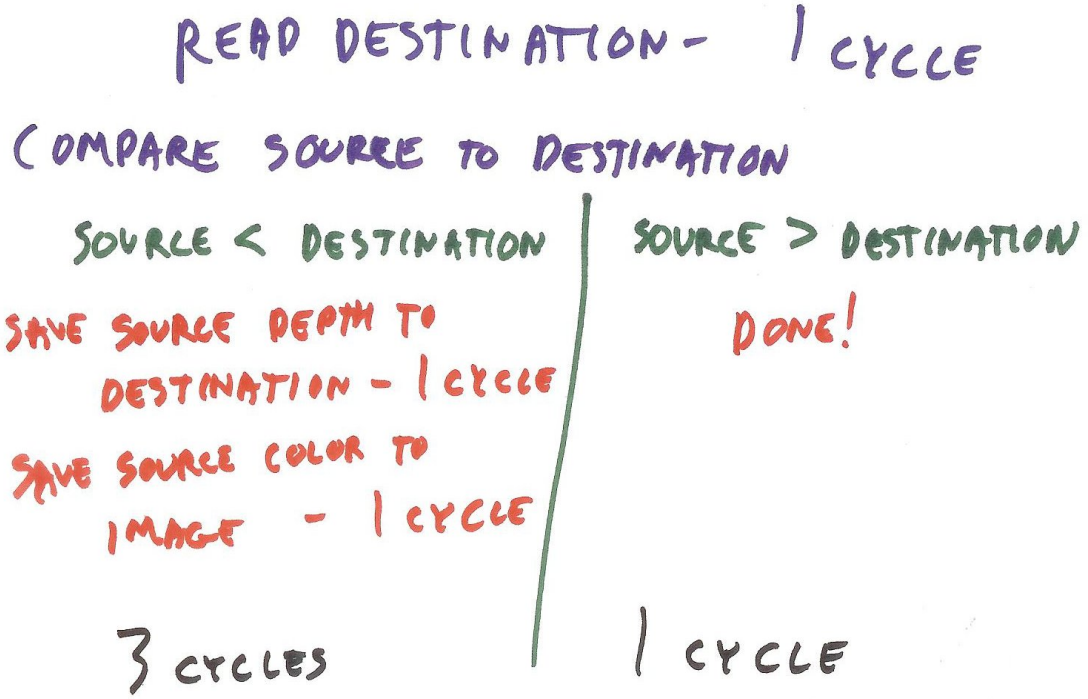
[1 cycle to read z-buffer,
1 cycle to write z-buffer,
1 cycle to write color
ignore all other costs]

( ) Draw the objects from front to back
( ) Draw the objects from largest to smallest
( ) Randomize the draw order, to avoid worst-case behavior
( ) Draw the objects from back to front

Answer

The answer is going to depend on how each z-depth comparison goes. When the object is closer than whatever is already in the Z-buffer, then the cost is 3 cycles: read old z-depth, write new z-depth, write new color. We call the new value coming in the "source", and the old value the "destination".

[table of these three operations shown here.]



When the object is farther away, only one cycle is needed, to read the old z-depth. To optimize the number of objects hidden by other objects, we want to draw those objects closer to the camera first.

Largest to smallest has no predictable effect, as the testing process is happening per pixel, so size doesn't matter. Randomizing might help avoid worst-case behavior, but it's not the fastest way. Back to front order is, in fact, the worst-case possible, since each object in turn is the closest object so far, so all values read in the Z-buffer will also be written. Roughly sorting and drawing from front to back is indeed a way in which applications optimize GPU performance.

Lesson: WebGL and three.js

What follows is a program using WebGL and a developer's library I chose for this course called three.js. WebGL is an Application Program Interface - "API" for short - that controls the GPU. It is based on OpenGL ES, an API used in mobile devices, which in turn is based on OpenGL, an API for desktop computers that dates back to 1992.

The other major competing API is called DirectX or Direct3D, which is commonly used for games on the XBox console and Windows PC's. This is almost the last time I'll mention DirectX. It's not all that different from WebGL - in fact, Google Chrome on the PC turns WebGL commands into DirectX calls. See the supplementary materials section for more information.

[Show WebGL comes from OpenGL ES from OpenGL. Show DirectX and Direct3D.]

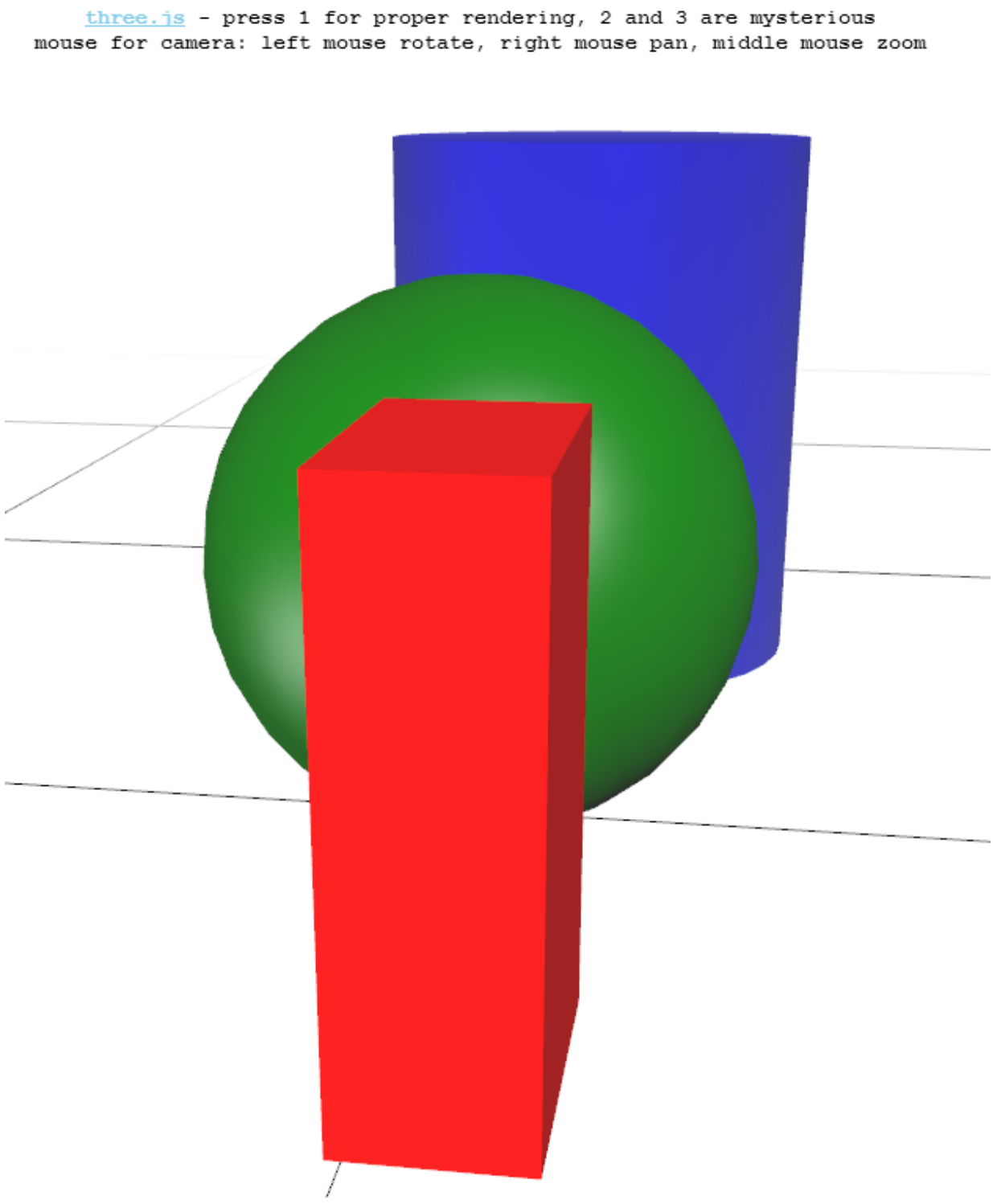
WebGL is a state-based API: you set up exactly how you want the GPU to do something, then send it geometry to render under these conditions. This gives a pretty fine-grained control over the GPU, but can result in hundreds of lines of code to draw even the simplest scenes. This is why we'll be using three.js, a free development library with many useful features. Just a few lines of code can create objects with materials, set up lighting and a camera, and even perform animation.

[Additonal course materials page: Three.js's homepage is http://mrdoob.github.com/three.js/ and is well worth exploring. I interviewed the creator of three.js here: http://www.realtimerendering.com/blog/interview-with-three-js-creator/ - I found his background inspiring. WebGL's homepage is http://www.khronos.org/webgl/ .To track the latest WebGL eye candy and other cool news, see http://www.webgl.com/]

Old stuff follows

To get your feet wet, we'll put a fairly simple program in the browser window. Push the "run" button and you should see three objects on a grid.

[Question is: should we put the code in the browser window, or just have them run a demo?]
[Show the program running]



You can use the mouse to rotate, pan, and zoom - see the instructions on the screen. The trackball control is quite powerful but a bit confusing at times. Here's my one tip: clicking and dragging along the left or right edge of the screen and moving up and down will rotate the whole scene.

In this demo program you can see the effect of turning off the z-buffer. To turn off the Z-buffer, press the 2 key, and push 1 to turn the Z-buffer back on.

[Instructor Comments: Run this program to try out turning the Z-buffer off and on.]

Back to New Stuff

Narration

Unit 1

WebGL setup - (if WebGL support is detected, will replace the video with this)
If you hear this, and can interact with a demo on your screen, it means that your browser supports WebGL. If you ever run into problems with our demos or programming exercises, try refreshing the page or restarting the browser. This technology is very new and not completely stable yet. The good news is that you will be learning both the very basics of what makes interactive computer rendering tick, and the bleeding edge technology that implements these principles.

FPS demo narration -
Congratulations! If you hear my voice and see a teapot on your screen it means that you can run our interactive demos. On the top right corner you can see a menu that allows you to change settings for this demo. These can be different for different demos and exercises. In this case you have a slider that allows you to change the FPS rate. Change FPS, then try to move the camera and see how it feels. Left click and drag will rotate the scene. On the left corner you can see how many miliseconds it takes to draw a frame. If the frame rate does not match the set FPS, it means that your videocard is not powerful enough, or the drivers are old. Remember, this is bleeding edge technology! Something like this was not possible even couple of years ago! Don’t worry however, even if you can not reach 60 FPS on this demo, most of our examples and programs will still run on your computer.

This is rendering option number 0. You see three objects on a scene. You can rotate the camera around the scene by clicking and dragging with your left mouse button. Note that the object and

light position does not change, what changes is the position from which you look at the objects.

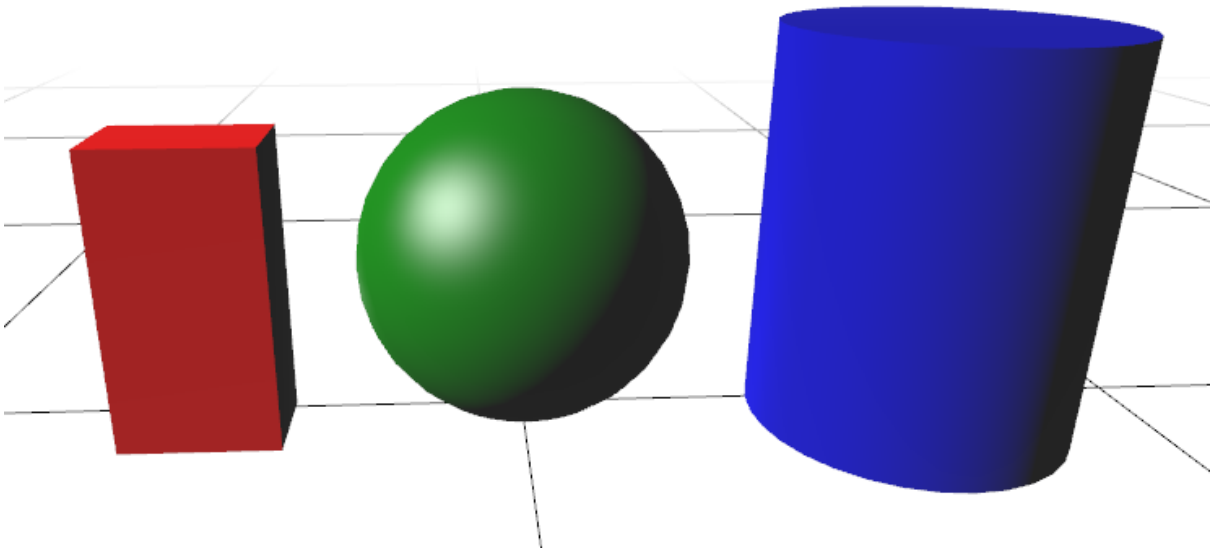
This is rendering option number 1. The scene - object and light positions - is exactly as in the previous demo, but as you can see, there is something wrong with the rendering, it does not look quite right. Look from different angles and try to think what is wrong!

This is rendering option number 2. The scene - object and light positions - is exactly as in the demos before, but as you can see, there is something wrong with rendering, it does not look quite right. Look from different angles and try to think what is wrong!

Quiz Intro: Draw Order vs. Sorted Introduction

You'll now run three demo programs that show different rendering modes. In true computer science fashion we'll call them #0, #1, and #2. When you move the mouse you'll change your view of the world - the objects themselves don't change position. Look at each demo carefully as you will have to answer a tricky question about the last two demos.

[Show the three object screen shot]



[More intro?]

Demos: Demos 1,2,3

Quiz: Draw Order vs. Sorted

Rerecorded 3/25

You just ran three demo programs. The first program gave the correct view of the scene. The next two demos were a bit more mysterious - call them mystery demos #1 and #2. The question to you is which of the following is true:

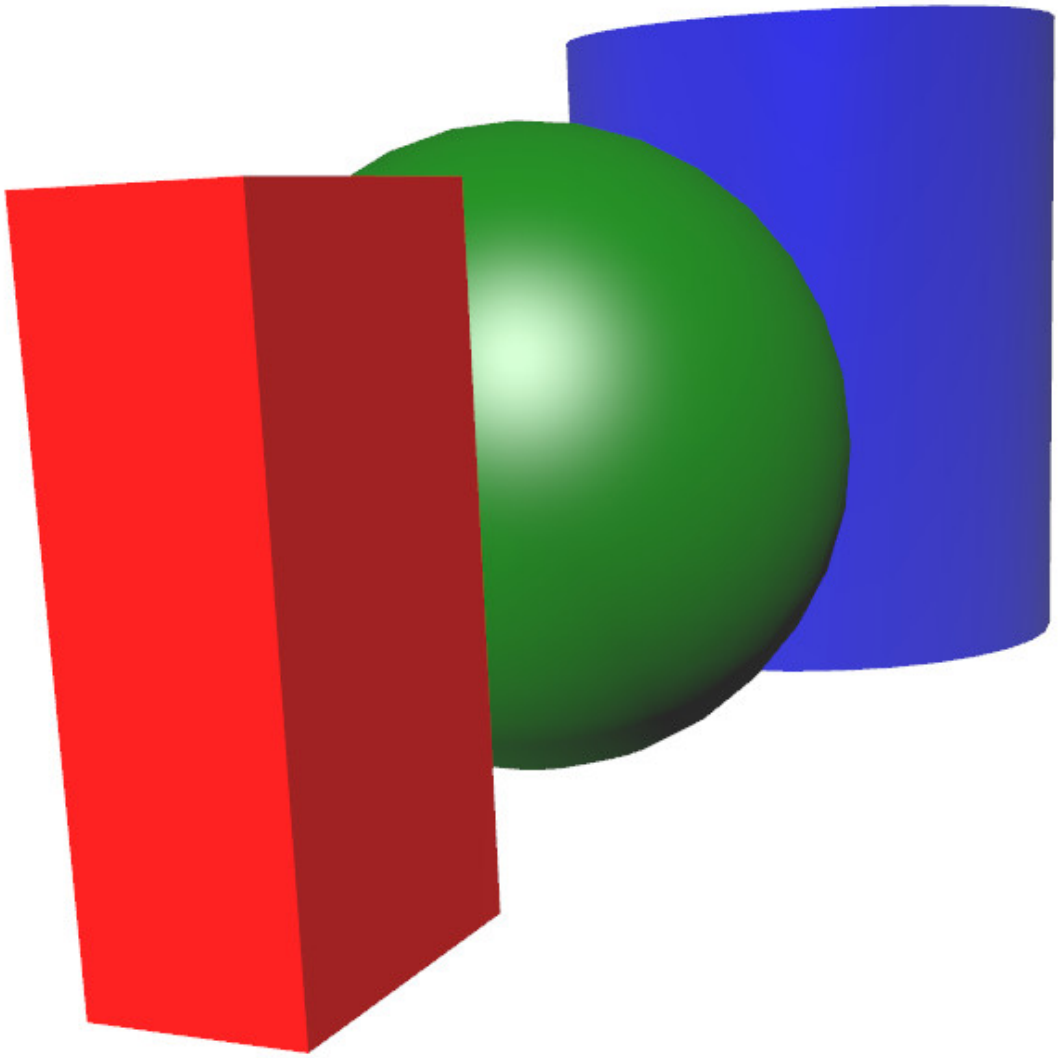
( ) #1 draws objects in a fixed order, #2 draws objects front to back.
( ) #1 draws objects front to back, #2 draws objects back to front.
( ) #1 draws objects front to back, #2 draws objects in a fixed order.
( ) #1 draws objects in a fixed order, #2 draws objects back to front.

[Change: Put demos in separate programs so they swap correctly. I also tried this, doesn't work. I think separate demos is probably the right answer, with some labelling to show which is which (not sure how to do that).

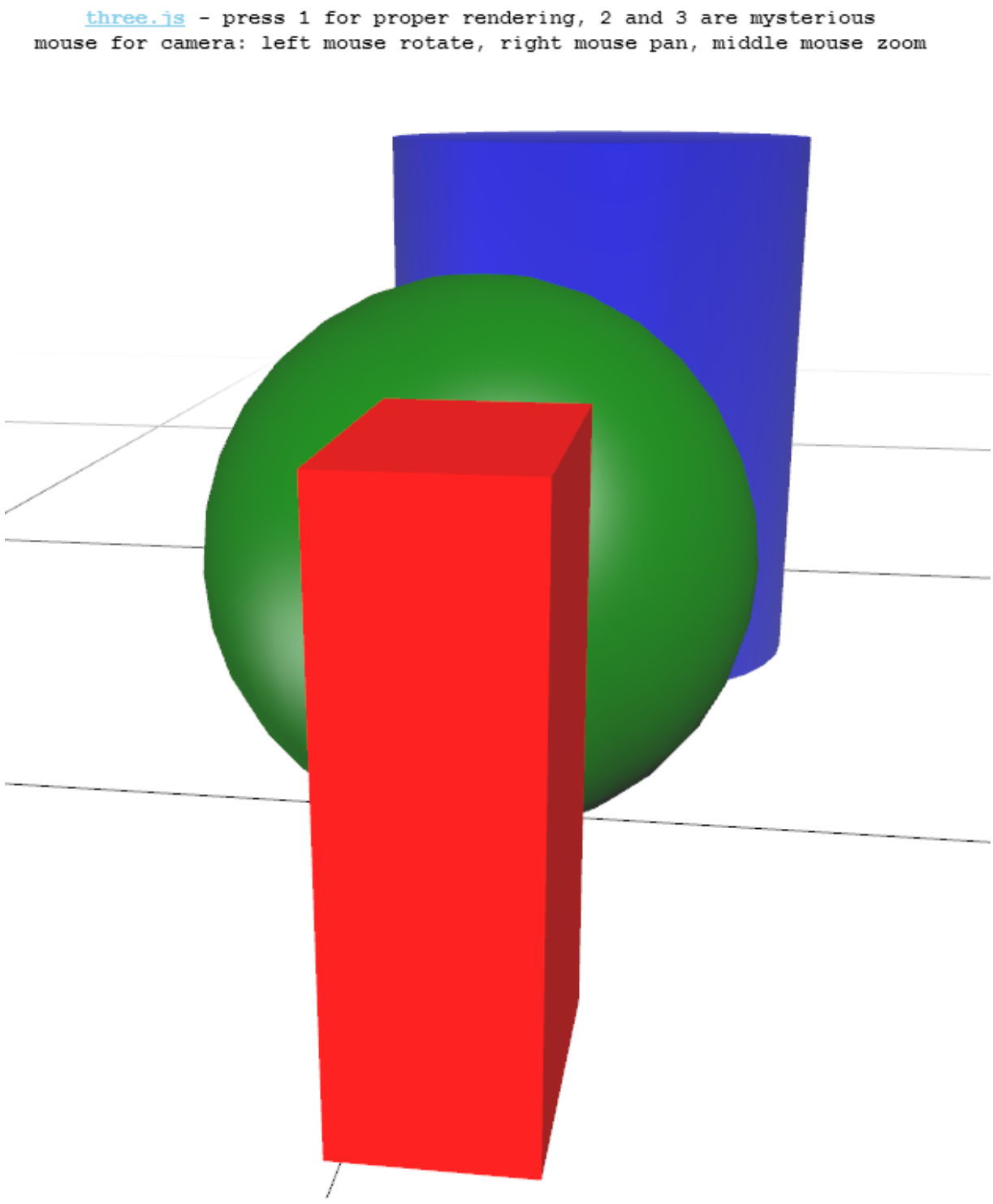
.depthTest
Whether to have depth test enabled when rendering this material. Default is true.
.depthWrite
Whether rendering this material has any effect on the depth buffer. Default is true.
When drawing 2D overlays it can be useful to disable the depth writing in order to layer several things together without creating z-index artifacts.

Answer

[This should just be a video clip of me running the program]



The correct answer is the first one. The first mystery demo gives a precise draw order: first the blue object, then red, then green.



The second mystery program sorts the objects in the opposite order that we'd want to use for the Painter's algorithm. It orders them so that the closest are drawn first. While this order is good when the Z-buffer is on, in that less shading computations are then needed, it gives a consistently wrong look.

[Instructor Comments: There are many three.js demos here, http://mrdoob.github.com/three.js/, along with links to a basic “getting started” guide, documentation, and more. Definitely explore, there are some interesting resources here, such as an alternate documentation guide that gives a different view of things: http://threejsdoc.appspot.com/doc/index.html

The three.js distribution itself can be downloaded from here, https://github.com/mrdoob/three.js/, if you know how to use github. You can develop WebGL programs locally on your machine, though this course does not require it. To set up a Windows machine for development, you’ll want to read this article I wrote: http://www.realtimerendering.com/blog/setting-up-a-windows-server-for-webgl/; there are also tips for Mac and Linux here, https://github.com/mrdoob/three.js/wiki/How-to-run-things-locally. If you do work locally on your own computer, you will eventually want to use the WebGL Inspector for debugging http://benvanik.github.com/WebGL-Inspector/, though this tool assumes a fair amount of knowledge of WebGL itself.]

[Supplementary materials: The ANGLE project is what converts WebGL calls into DirectX, you can find the code here http://code.google.com/p/angleproject/. You can read about the ANGLE project in the book “OpenGL Insights” http://www.amazon.com/OpenGL-Insights-Patrick-Cozzi/dp/1439893764. This chapter is also online for free - http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-ANGLE.pdf]

Conclusion

[show demo if we don’t do headshot then use http://mrdoob.github.com/three.js/examples/webgl_shader2.html]

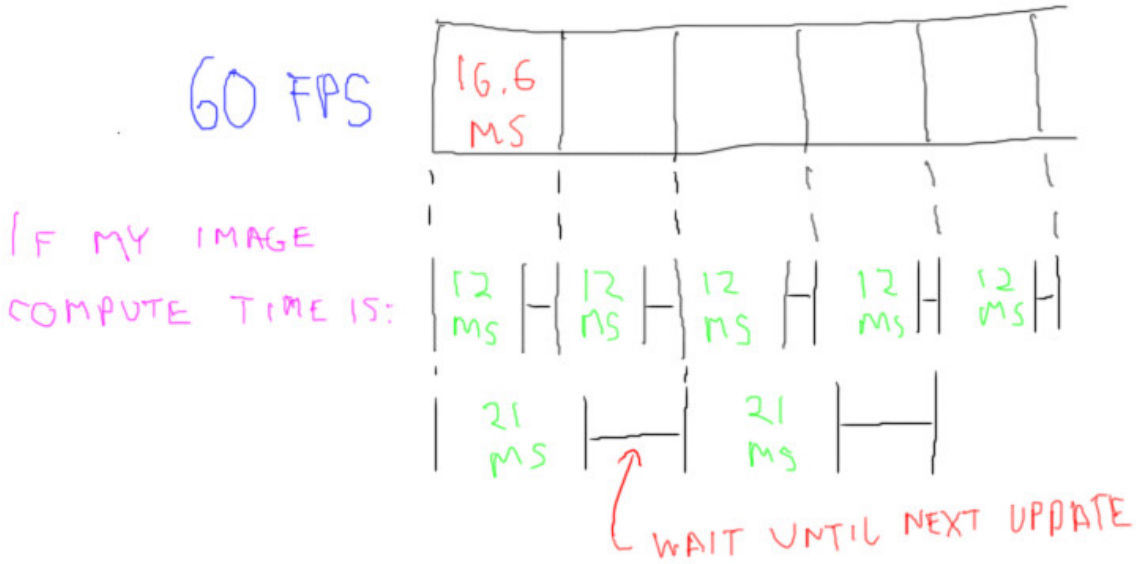
"You now know about the fundamentals of interactive rendering. We've covered a lot of ground: the basics of lighting and cameras; pipelines and bottlenecks; pixels and the Z-buffer. I'd estimate you now know more than 99.44% of the population about what makes interactive rendering tick. In the lessons ahead we’ll learn more about these subjects and related fields. You'll also begin to program and create virtual worlds yourself as we delve further into the key elements of interactive 3D graphics."

Problem Set 1

[Recorded 2/25]

Problem 1.1 Frame Skipping

[show frame miss]



If you experimented with the frames per second demo, you may have noticed something odd. If you slid the slider from 30 frames per second up to 59 frames per second, the frame rate shown in the upper left is a pretty solid 32 milliseconds per frame. If you go all the way to 60 FPS this drops to 16 or 17 milliseconds per frame. This happens because a new frame is not computed until the previous frame is displayed.

For example, say an image takes 12 milliseconds to compute. This is well within 16.6 milliseconds per frame, so each frame will display a new image. If the image takes say 21 milliseconds, this is longer than the 60 frames per second rate. It's sort of like trains leaving the station at a constant rate: if you don't get your work done in time to catch the current frame, you have to wait until the next one comes around.

The fastest we can display frames is 60 frames per second. So, as soon as an image takes more than 1/60th of a second to compute, it will miss the first frame. This makes the effective rate 30 frames per second, since only every other frame is updated with a new image.

I should say this doesn't always happen with every application, it depends on the way updates are done. This problem assumes that we don't begin to compute a new frame until the old one is displayed.

The question to you, then, is what is the effective frame rate if you just miss 30 frames per second. In other words

Say your image compute time is a tiny bit greater than 1/30th of a second, at what rate can you display these images?

[ ] FPS

[Instructor Comments: Feel free to run the FPS demo again to see the phenomenon: http://www.realtimerendering.com/udacity/?load=demo/unit1-fps.js]

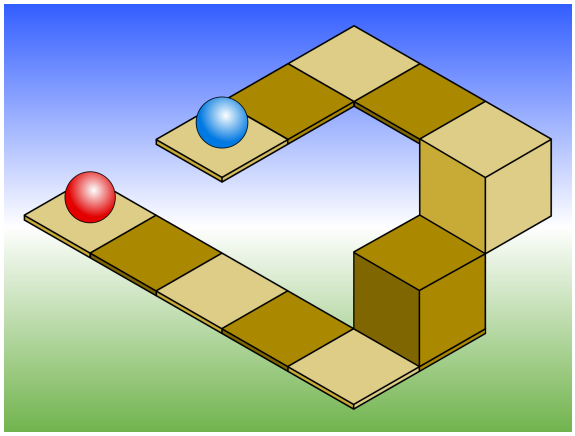
Answer

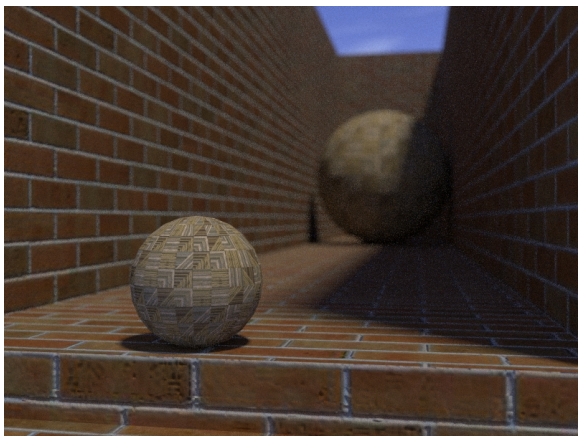
If it takes you slightly more than 1/30th of a second and the monitor updates at 60 frames per second, this means you'll miss two frame updates and be able to update only every third frame. 60 divided by 3 is 20, so 20 is the answer. I'm fine with either using logic or experimentation to figure this out, but logic gives you a deeper understanding.

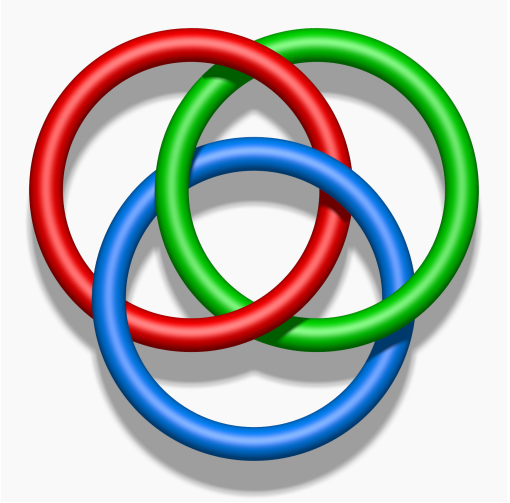
Problem 1.2: What is Not Paintable?

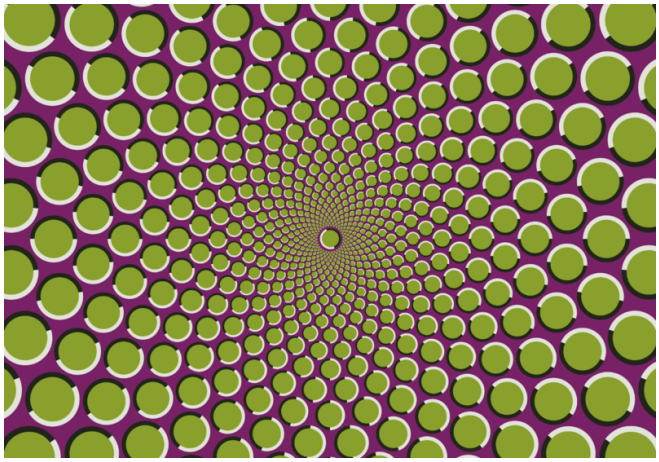
re-recorded 3/25

[Add a radio button next to each one, put circles on far left and far right]









[***Which cannot use the Painter's Algorithm?***]

I have a rendering algorithm question for you. Here are four different optical illusions. Assume each object in each of these scenes is drawn separately. For example, each block or square, each brick, each ring, and each dot is a separate object. Which of these scene most definitely could not be rendered with the basic Painter's Algorithm?

Answer

The Painter's Algorithm works by drawing each object fully, in back to front order. It depends on there not being any cycles, where one object is in front of another, which is in front of a third, which in turn is in front of the first. This image has a red ring that is in front of the green, the green is in front of the blue, and the blue is in front of the red. GIven this cycle, it certainly cannot be rendered with the Painter's algorithm if each ring is drawn as a whole. It's possible to process a scene and break the objects into pieces that could be drawn in the proper order, but the basic Painter's Algorithm clearly fails here, so this is the answer.

By the way, these are called Borromean rings: no two rings are linked to each other, but still all three are attached.

[Additional Course Materials: Borromean Rings are discussed here http://en.wikipedia.org/wiki/Borromean_rings.

Image credits:
http://commons.wikimedia.org/wiki/File:IsometricFlaw_2.svg,
http://commons.wikimedia.org/wiki/File:Opsphere3.jpg
http://commons.wikimedia.org/wiki/File:Borromean_Rings_Illusion.png
http://en.wikipedia.org/wiki/File:Motion_illusion_in_star_arrangement.png]

Problem 1.3: FPS vs Milliseconds

[***12 FPS vs. 24 FPS - twice as fast***

inverse: milliseconds per frame.

30 FPS == 33.3 ms per frame
60 FPS == 16.7 ms per frame

-10 FPS - compared to what?

]

[12 FPS vs. 24 FPS - twice as fast]

Measuring frames per second is a common way to talk about an application as a whole. If an application runs at 12 FPS on one system and 24 FPS on another, we can truthfully say the second system is about twice as fast in this instance.

[ms per frame]

However, graphics programmers themselves have been moving towards using the inverse of frames per second: milliseconds per frame. There are a number of good reasons for this. The most appealing one to me is that milliseconds gives you a solid number to use for comparison of different algorithms.

[30 FPS = 33.3 ms per frame]

If you talk about an algorithm's implementation in terms of milliseconds per frame, you can then treat the effect on speed as a single number. "This user interface is costing us 5 milliseconds per frame" is a useful fact in its own right. "This user interface slows us down by 10 FPS" is not all that helpful, since we don't know what the frame rate was before. Rating individual parts of the rendering process by frames per second doesn't make a lot of sense, since it's the sum of the costs that's important.

You can add millisecond costs together to see how much a frame will typically take. Games, for example, will often have a budget, say 33.3 ms per frame, whichs means 30 Frames per second. If different effects cost various amounts of time, it's easier to add these up and see if we're over budget

[60 FPS = 16.7 ms per frame]

If an application aims for 60 frames per second, the budget's halved.

Another reason to avoid Frames per second is that they're difficult to work with mathematically. So, of course, that's what I'm going to ask you to do, to get a feel for this sor of problem.

You measure your application, a walkthrough of a building. Here are the rates you find:

Walkthrough benchmark:

25 FPS for 100 frames
50 FPS for 100 frames
25 FPS for 100 frames
10 FPS for 100 frames of your benchmark.

What is the average rate of the application in frames per second? _____ FPS

I'll give you a hint: simply averaging the FPS values is not going to work. Think about how many frames are rendered and how many seconds pass during the entire benchmark.

Walkthrough benchmark:

25 FPS for 100 frames
50 FPS for 100 frames
25 FPS for 100 frames
10 FPS for 100 frames

What is the average rate of the application in frames per second? _____ FPS

Answer

Walkthrough benchmark: 1000 ms/25 FPS = 40 ms

25 FPS for 100 frames - 40 ms * 100 frames = 4 s
50 FPS for 100 frames - 20 ms * 100 frames = 2 s
25 FPS for 100 frames - 40 ms * 100 frames = 4 s
10 FPS for 100 frames - 100 ms * 100 frames = 10 s - total 20 s

What is the average rate of frames per second? _____ FPS

[first, then erase]

25 + 50 + 25 + 10 / 4 = 27.5 FPS - WRONG!

[second]
400 frames / 20 seconds = 20 FPS

Usually we just add numbers up and take the average. This doesn't work with frames per second, and that's the problem. If you add the four values together and divide by four you get 27.5 frames per second. That's not correct, however. Say we think about this question in terms of miles per hour. Instead of 400 frames, say you drive 400 miles. If you drive 25 miles per hour for the first 100 miles, 50 miles per hour for the next 100, and so on, how long will the trip take you? Divide 400 miles by this number of hours and you'll get your average number of miles per hour.

[4 2 4 10 to right]

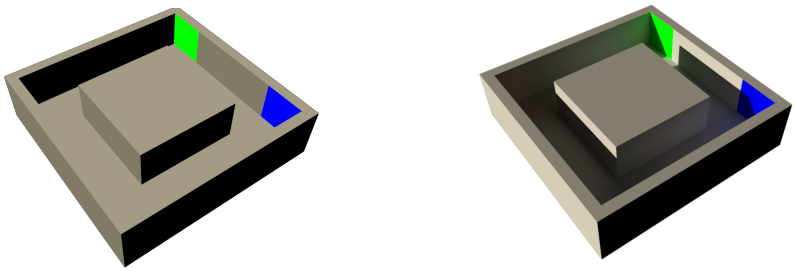
Using this method of computing frames per second, we see that the first set of frames takes 4 seconds to view, the second 2, third 4, and fourth 10, a total of 20 seconds. 400 frames divided by 20 seconds is 20 frames per second. This answer is a little surprising, in that it's lower than three of the four frame rates measured. The duration of the 10 frames per second portion of our test is so long that it pulls the average down.

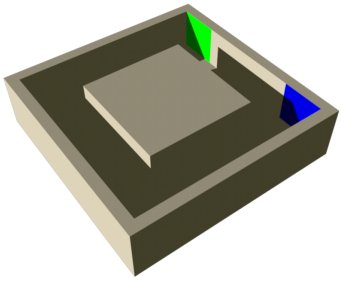
[Additional Course Materials: this blog post talks about the FPS problem in general http://renderingpipeline.com/2013/02/fps-vs-msecframe. This research articles talks about it for mobile devices: http://graphics.cs.williams.edu/jcgt/published/0001/01/03/]

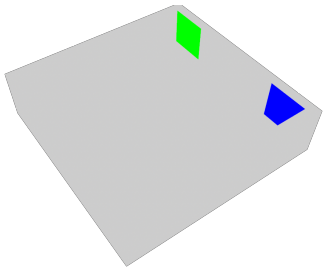
Problem 1.4: Rendering Costs

Here are four renderings of the same scene. One is trivial to render, the others take more computation. It’s up to you to think about how expensive each is to render. Put on each line a number from 1 to 4: 1 means the quickest to render, 4 means most complex.

[On here label A B C D in normal order]







[Additional Course Materials: these four images should be available for display on their own.]

Answer

[D is 1, A is 2, C is 3, B is 4]
[upper left is noshadows.png, upper right is global_illum.png, lower left is ambient_and light.png, lower right is ambient.png]

The quickest scene to render is the one without any lighting at all, rendering D. These objects just use a solid color, and so little detail is visible.

The second fastest to render is rendering A, as the objects do not cast any shadows. Notice how details get lost near the front edges. It's hard to tell where the floor ends and top of the outer wall begins.

The next fastest is the rendering C. This one has shadows, which take longer to render. Shadows help us a bit in telling how surfaces are related to each other. However, everything in shadow looks the same, since we're computing only light that directly hits these surfaces.

The slowest rendering is rendering B. It's also the highest quality, as light bounces off the walls and into the shadowed areas, so giving them some definition.

Problem 1.5: Firefly

Here's a thought experiment. Imagine you're a firefly.

[http://commons.wikimedia.org/wiki/File:Photinus_pyralis_Firefly_3.jpg]

Now imagine you're flying around in a closet with the door closed - you're the only light source. Finally, imagine your light is quite bright and that you have human eyes. Horrified yet?

An easier way to say it is: imagine there's a single light in the scene, and you're viewing everything from its location.

Which of the follow are true:

[ ] Objects are visible, but get dimmer the farther you are from them
[ ] Objects are visible, but only in shades of gray
[ ] Objects are visible, but are only solid colors without shading
[ ] Objects are visible, but no shadows can be seen

Answer

The first choice is true: objects definitely get dimmer the farther they get from a light source. The

number of photons hitting a surface drops off proportional to the square of the distance from the light.

The second choice is false, there's no reason that the objects wouldn't have colors. Well, maybe if all the objects in the closet were gray, but that's not a fair assumption.

The third choice is also false. Even though you're at the light source's location, the objects can receive different amounts of photons depending on their distance from and angle to the light.

The fourth choice is true. This is the main point of this question: what the light "sees" is, by definition, not in shadow. We'll use this fact later in the course to generate shadows on objects.

Problem 1.6: Light Field Dimensions

[http://commons.wikimedia.org/wiki/File:USMC-070914-M-7696M-010.jpg and http://commons.wikimedia.org/wiki/File:Tower_illusion_-_Tamil_Nadu,_India.jpg]

[***Radiance*** later add ***Light Field*** **-** see below for how the final thing should look, leave room for the question!

How many numbers do you have to set?
Radiance **=** ***LightField****(____ variables)*]

When we look at a scene, we can't always tell how far away objects are. Our eyes, well, at least my eyes, are not laser scanners, they don't return z-depths directly to my brain. Maybe another way to say it is that we're not bats and don't use sonar.

Our eyes detects what is called the “radiance” coming from each direction. Our brain figures out distances from a variety of visual cues. We’re not even particularly aware of these cues. For example, it wasn’t until the Renaissance that painters figured out that distant mountains had a blue tinge, and so started to paint them that way. However, the actual amount of light from those distant mountains is a given amount of radiance - we don’t store a depth.

One way to think about radiance is in terms a function called the “Light Field”. Say we have a pinhole camera. We can move the camera around and point it in different directions. This camera gathers the radiance values for a bunch of similar directions and makes an image from all of these.

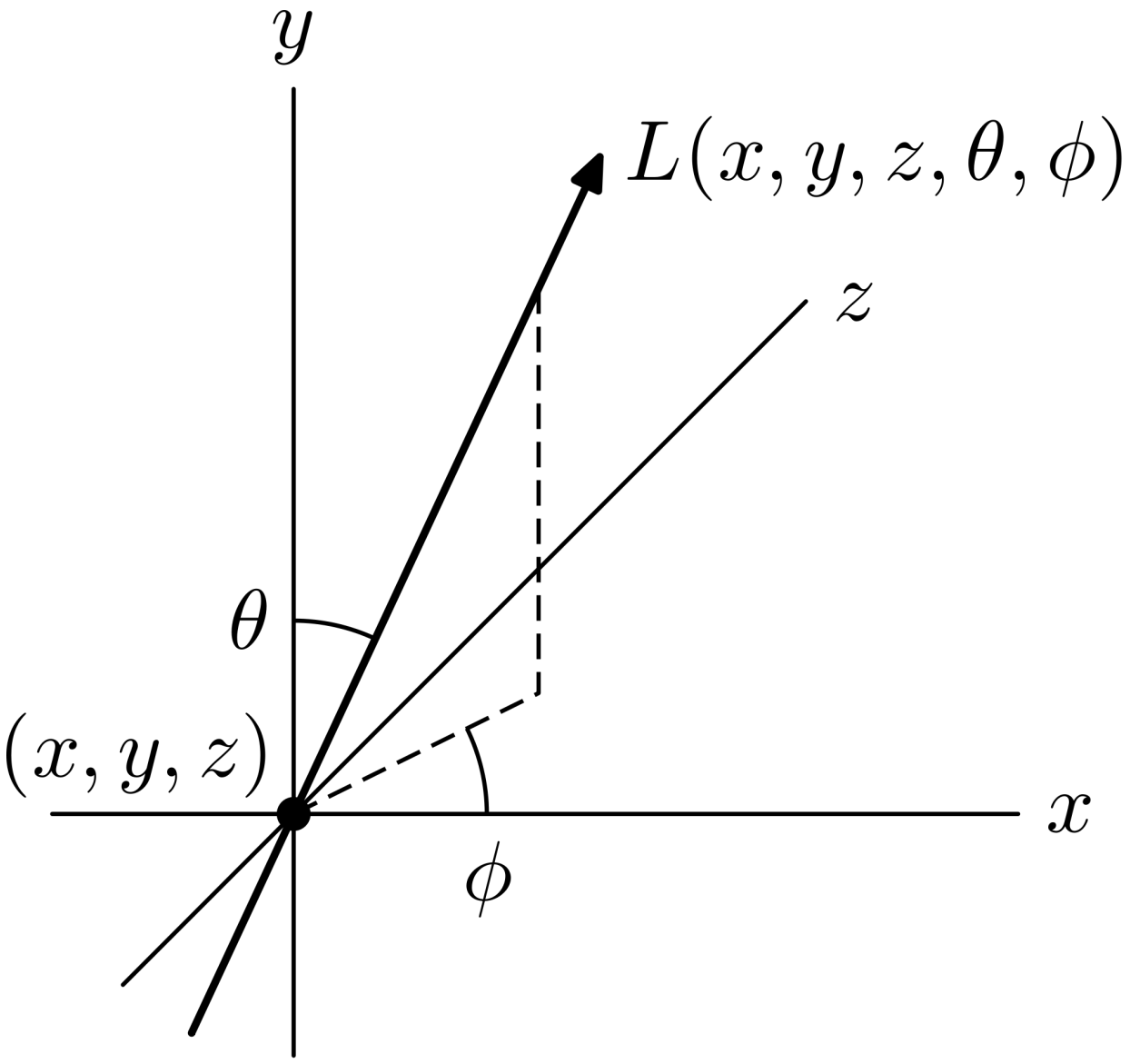
Now simplify the camera model even more: imagine you’re looking through a long narrow tube. Whatever color you see through the tube is essentially the radiance value from that direction.

The question to you is, how many numbers, how many variables, do you have to give in order to specify any radiance value you want to gather? Don’t worry about the fact that light has different wavelengths, or that time could be involved.

[Additional Course Materials: photo credit Wikimedia Commons
http://commons.wikimedia.org/wiki/File:USMC-070914-M-7696M-010.jpg and
http://commons.wikimedia.org/wiki/File:Tower_illusion_-_Tamil_Nadu,_India.jpg]

Answer

[Leave the question, swap out the image with this]



Here's the way the light field is expressed mathematically. The answer is 5 variables: an XYZ position in space and an altitude and azimuth. For me, it's just easier to think about looking through a tube. The one end of the tube can be put in any place in the universe, so that's three dimensions for its location. The direction of the tube can be specified by just two numbers. This direction is sort of like latitude and longitude [point at diagram], if you're looking out from the center of the earth. Well, it's probably pretty dark in the center of the earth, but I think you get the idea.

One more way to think about it: if you take a pinhole camera, you can position the pinhole itself anywhere - that's three dimensions. The camera now takes a 2D image, which is two more dimensions. This camera keeps the x,y, and z position fixed and varies the remaining two variables.

[add ***computational photography]***

The point is, every radiance value in the universe can be specified by this 5 value function; three for position, two for direction. This has all sorts of interesting implications. For example, in the area of ***computational photography***, there are different camera configurations that allow us to vary up to all five variables to some extent, as well as over time, a sixth variable. The data gathered can be used for all sorts of applications, such as image deblurring, capture of how a material responds to light, and reconstructing a 3D computer model from a real-world scene.

[Additional Course Materials: The Light Field is described here
http://en.wikipedia.org/wiki/Light_field. Drawing from
http://en.wikipedia.org/wiki/File:Plenoptic_function_b.svg
]

Problem 1.LAST - Debugging. This should be last in unit, Gundega

[recorded 2/25]

[In the browser we should have the code for
http://www.realtimerendering.com/udacity/?load=unit1/fix-javascript-errors.js]

For your first encounter with actual WebGL and Three.js code, I will give you a simple program that creates a scene with one light and one object - a cube.

To see it, however, you will have to get your hands dirty and fix a few syntax errors. If you are familiar with JavaScript, you will probably be able to catch them by eye. If not, use the browser-specific developer tools to find and fix the lines with errors. When you fix all four of them, you should see the cube [show the actual demo here] and be able to view the scene from different angles.

Try hitting the “run” button to run the code. After that, try changing numbers in the code to see the effect. Better yet, read the three.js documentation mentioned in the Instructor Comments below and learn more about what each class does. If you get things into a broken state, just refresh the page on your browser.

If you don’t feel adventuresome, that’s also fine - I’ll be covering all of these operations in the lessons ahead. More importantly, I’ll be explaining the underlying physical principles and mathematics behind each of these areas.

When you’re done experimenting, just hit “submit” - for this exercise (and only this one) there’s no wrong answer.

[Instructor Comments: This course uses JavaScript to call three.js and WebGL itself. I won't directly be teaching JavaScript itself. For the most part you'll be able to get away with looking at the example code and adding bits to complete the exercise. Syntax errors will be detected within the exercise framework itself.

[For wiki, etc.:
For a more serious program in JavaScript, you'll want to learn the language yourself. Luckily, there are plenty of resources that will teach you this language. One place to start is https://developer.mozilla.org/en-US/learn/javascript Mozilla's resource page - Codecademy http://www.codecademy.com/tracks/javascript-combined is a good online. A similar in-browser course is at w3schools http://www.w3schools.com/js/. If you're pretty literate about computer languages, you might enjoy the book JavaScript: The Good Parts, http://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742, which teaches the parts of the language you should learn, and notes what parts are worth avoiding.

If you want to edit code files outside of the browser, all you'll need is a text editor. A reasonable free editor is Notepad++ http://notepad-plus-plus.org/. It's what I used in making this course, just because I'm used to it. Another choice is the Sublime Text editor http://www.sublimetext.com/2. It will occasionally nag you to buy it, but is mostly usable for free. Its strength is in the numerous plugins that can help you manipulate your text in various ways.

There are IDE's such as Aptana http://www.aptana.com/ and the (commercial) Webstorm http://www.jetbrains.com/webstorm/. These have the advantage of running a server internally so that you don't have to set this up. Personally, I set up my own server and use the free Notepad++ text editor and run the code in Google's Chrome browser. To debug JavaScript in Chrome, Ctrl-Shift-J (on the Mac, Command-Option-J) or F12 toggles the interface on and off, and brings you to the Console tab shows syntax errors. You can also use Ctrl-Shift-I to toggle; this method doesn't switch you to the Console tab but keeps the previous tab selected. The Sources tab lets you set breakpoints and examine variables. On Firefox you need to first install a plugin called Firebug http://getfirebug.com/. This debugger has some more powerful features, such as being able to hover over variables to see their contents. Safari also has debugging built in http://hints.macworld.com/article.php?story=20030906093300383.

JSHint http://jshint.com can help you create well-formed code without problems. It will catch some dumb errors and bad practices in JavaScript code. You simply copy your code into this window and push the "Lint" button. I usually turn off the "When code is not in strict mode" option, as this option is a bit of a pain and has never found any bugs for me. All course code has been run through JSHint.

There are a number of quickstart guides to three.js. Here http://mrdoob.github.com/three.js/docs/53/#Manual/Introduction/Creating-a-scene is the official one, and there is another here http://www.aerotwist.com/tutorials/getting-started-with-three-js/

with a bit more information.
]