CSC317 Notes

Computer Graphics

Last updated February 1, 2024

Contents

Computer graphics accept, process, transform, and present information in a visual form. These can be static or animated pictures, but this course revolves around image generation using computers.

There are three core areas of computer graphics:

- Modeling
  - Taking a shape, finding some mathematical expression for its geometry and topology. How can we represent shapes?
- Rendering
  - The simulation of how light interacts with your shape. We'll deal with simple rendering (raytracing).
- Animation

1 Raster Images

Every computer display device is a raster image display device. Images that are divided up into square sub-units called pixels (picture element).

If you zoom in on a pixel in a display, you will see that it is made of subpixels. There are different kinds of subpixel arrangements, but the typical subpixel arraignment is RGB. The combined intensity of all these subpixels gives you the color of that pixel.

There are all kinds of fancy ways this gets implemented.

VIEW THE SLIDES ON CAMERAS

1.1 Images as a Function

$$I(x, y) : \mathbb{R}^2 \rightarrow \mathbb{R}^{+n}$$

The *nD* real numbers (greyscale: $n = 1$, RGB: $n = 3$, RGBA: $n = 4$). When we want to decide how to represent an image, we want to represent it in finite no. of variables we store on a computer.

Raster images make this very particular choice that our 2D domain of our function is divided into squares we can index, and each square holds a constant value, different in each piece. We call that a piecewise constant – each part of the image holds a potentially different image. The piecewise constant assumption gives us all the properties of raster images. It gives us this indexing pattern but results in finite resolution.

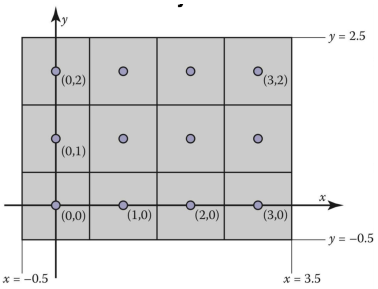
In each piece of the image, we store the radiated light value for some object.

$$\text{color} = \int_{\text{Area}} \text{Light dArea}$$

To make this useful, we need to assign a coordinate system. We need to be able to talk about which square of the image we're looking at.

There is no standard pixel coordinate system.

This is the one we'll use.



This seems a bit weird, but we're going to have to agree with this coordinate system. Here and now, integer values are going to lie in the center of each pixel, and $(0,\ 0)$ will be the center of the pixel on the bottom left.

When you're writing code, it means that if you want to go all the way out to the edge of the image, there will be a half offset from the center of the last pixel. The same happens for the first pixel.

It does mean that these index calculations will be littered with -0.5 and 0.5, so be careful when doing the first assignment.

1.2 Data Types for Storing Images

Each coordinate stores three value: RGB. We have to decide what type to score those values in. That's important because the amount of data stored determines how big your image is. For a 1024x1024 image:

- Bitmap (black or white): 128KB
- Grayscale 8bpp: 1MB
- Grayscale 16bpp: 2MB
- Color 24bpp: 3MB (bits per pixel)
- Floating-point HDR color: 12MB

Typically, when you store an image you store an alpha channel. The alpha channel lets you control the transparency of the object. If alpha is 1, the pixel is opaque, but when set to 0, the pixel is totally transparent.

Typically, we store 8 bits per channel (R from R, or G from G…)

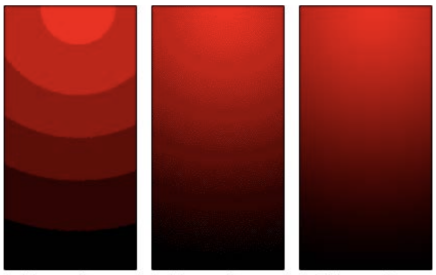
When you read in an image file, you'll read in integer values for each channel. When you want to average those values, convert them into floating point values. If you get those conversions wrong, you get very weird results. Remember to convert and quantize back to 0-255. Be very conscious on what the input format of these images are, as this is a common thing that is easy to get wrong when you do low-level image processing.

RGB is a technical thing – right away you can't tell what color it is until you actually check it.

1.3 Artifacts of Raster Images

Banding: because we only have discrete values for color information, there are big jumps when we send them to the display, especially GIFs. There is not enough color depth to represent something smoothly shaded. Fixes?

- Use more data, store everything in more bits. Great if memory is abundant.
- Dithering (that's what you happen to see in most GIFs)
  - You still see banding, but your image uses the same color information. We are flicking paint onto the screen rather than using more colors. When your eyes see the result, your eye does integration itself, averaging out the color. You can affect the color you perceive in some reason, tricking the eye to see the intermediate color by putting two discrete colors by each other.



8-bit gradient | 8-bit gradient, dithered | 24-bit gradient

This is what they use in printing to expand the colors perceived on printed documents.

You could also try clipping

- Truncate all absolutely white images

Or by tone-mapping (re-scaling, similar to your eye's pupil size changing)

Or by Gamma Correction. We typically use linear encoding for colors: if you have $R = 128$, you expect half as much as $R = 255$. The problem is that if you have a display that translates these color values to voltage if you half the voltage it will not give you half the emitted intensity. Gamma correction was invented to solve this problem. It is based on a very simple correction.

$$\text{Displayed Intensity} = (\text{max intensity of display})\, a^{\gamma}$$

Where a is the amplitude of image $[0,\ 1]$ (normalize it from 0-255) and γ is the gamma. The gamma correction may be different for each color.

How do we compute γ? Find a max image amplitude that is $\frac{1}{2}$ the display brightness, and you fit the model:

$$\gamma = \frac{\ln(0.5)}{\ln(a)}$$

1.4 Transparency

Append RGB to be RGBA. When we have transparency, we append a fourth channel alpha, which denotes how opaque the image is. A high alpha, the pixel is opaque, and a low alpha denotes greater transparency.

If you are given two images, how do you combine them? One easy way is to only use the alpha of a single image:

$$\vec{c} = \alpha\vec{c}_{\text{foreground}} + (1 - \alpha)\vec{c}_{\text{background}}$$

Compositing is the most useful image editing technique ever. Every frame in a movie

has so much compositing. In a movie shot, nothing is ever done at the same time. A compositor would combine all of them.

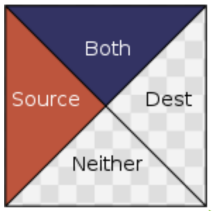
Compositing is about layering images on top of the other. How can we combine these two images?

Say we want to put the destination on top of the source image, except in places where the source image exists.

To make any compositing operation, you only need to make four choices / cases:

- Both visible
- Source only
- Destination only
- Neither visible (alpha 0)

Any compositing operation we want to make is just to decide what color we put down.



This is our operation for the image above.

You can write this out as a single image:

$$A_{src} \cdot [\text{source}] + A_{dest} \cdot [\text{destination}] + A_{both} \cdot [\text{both}]$$

Where (note that the colors are what we decide to place in the square above split by 4, nothing special):

$$A_{src} = \alpha_s \cdot (1 - \alpha_d)$$
$$A_{dst} = \alpha_d \cdot (1 - \alpha_s)$$
$$A_{both} = \alpha_s \cdot \alpha_d$$

2 Ray Casting

Define a camera point in our 3D scene, define a pixel point somewhere and we'll shoot rays into a pixel into the 3D world. We'll see if the ray hits everything, and if it does, we'll give the pixel a color or a property determined by where we hit the object. So, we think of Ray Casting as the pixel first rendering:

- For each pixel in the image
  - Generate a ray
  - For each object in the scene
    * Intersect ray with object
    * Do something with it

2.1 Basic Components of Ray Casting

- Ray
- Camera
- Intersection tests

2.2 The Ray

$$\vec{p}(t) = \vec{e} + t\,(\vec{s} - \vec{e})$$

A ray is just a line segment that goes to ∞ in one direction. We'll define all our rays parametrically. They emit from the eye $\vec{e}$, and we'll parameterize the point along the ray using a scalar parameter t.

- $\vec{e}$ is the eye point
- $\vec{s}$ is the scene

2.3 The Camera

How do we parameterize the camera?

What we would like to do, is given a camera, we want to build an orthogonal coordinate frame. If you were looking at a camera, $\vec{u}$ points to the right, $\vec{w}$ points towards you, and $\vec{v}$ points in the upwards direction relative to where the camera is looking. We're using the right-hand coordinate frame, so remember that $\vec{w}$ points backwards. Beware how the coordinate frame is defined, and it is best to use this coordinate frame.

2.3.1 View and Up to Orthogonal Frame

When you open a piece of software, the parameters they want you to set to describe the camera orientation is rather a view direction and an up direction (the up direction

is the most confusing, it just points such that anything above the half plane described by the up vector will be described as positive).

The trick is, to convert these two inputs: **view** and **up** into an orthogonal frame, and calculate the coordinates like these:

$$\vec{w} = -\frac{\text{View}}{||\text{View}||}$$
$$\vec{u} = \frac{\text{View}}{||\text{View}||} \times \frac{\text{Up}}{||\text{Up}||}$$
$$\vec{v} = \vec{w} \times \vec{u}$$

We now have the camera frame.

2.4 Generating Rays

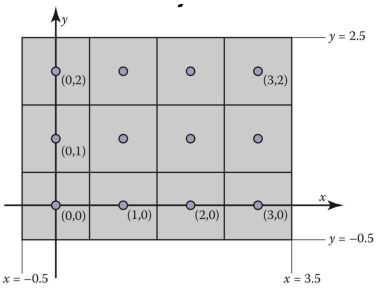
View and up only describes the camera's rotation, but we also need to understand where it is. You will be given the origin of the camera, the eye point, $\vec{e}$. Now that we have the camera, now do we generate the rays?

The ray tracing model is done on a deconstructed camera. We'll move the sensor in front of the camera at some distance d away from the eye point $\vec{e}$.

If you want to describe d in the camera coordinate frame, if we want a vector that starts at $\vec{e}$ and I want to know the coordinate on some point of the screen, it's going to be $-\vec{d}$. Annoyingly, the $\vec{w}$ vector points antagonistic to the view camera.

d represents the focal length. You can imagine what changing $\vec{d}$ does; as $\vec{d}$ goes up, the rays converge to be one parallel ray and if d goes down, you'll get a fisheye lens effect.

Keep in mind our ray has an origin $\vec{e}$, it has some direction vector, and we need to be able to compute this direction vector. The direction vector is determined by the pixels it passes by.



Remember the coordinate system we're using.

If we have the origin of the camera frame, what are the coordinates for pixel (i, j) in the camera frame?

We're going to build a linear transformation that does this. All we have to do is solve a set of linear equations. By construction, our sensor is parallel to our coordinate frame. If it's parallel, it's $\vec{w}$ coordinate will be fixed, being $-d$.

Then, we need to convert pixel coordinates (i, j) to (u, v).

Here it is: Our sensor has a width and height that we have (physical width/height of image in 3D space), and the very center of our sensor is at $v = 0,\ w = -d,\ u = 0$

- Bottom left corner of the sensor: $(i, j) = \left(-\frac{1}{2}, -\frac{1}{2}\right)$
- Top right corner of the sensor: $(i, j) = \left(n_x - \frac{1}{2}, n_y - \frac{1}{2}\right)$
- Bottom left corner $(u, v) = \left(-\frac{\text{width}}{2}, -\frac{\text{height}}{2}\right)$
- Top right corner $(u, v) = \left(\frac{\text{width}}{2}, \frac{\text{height}}{2}\right)$

So it is in our camera coordinates that we will get the physical size of our sensor.

So we can say:

$$u = a \cdot i + b$$
$$v = c \cdot j + d$$

Meaning (u, v) is some linear equation of (i, j). We can plug in our four values (the four corners) we've found above and solve for u, v. The answer is:

$$u = \left(\text{width} \cdot \frac{i + \frac{1}{2}}{n_x} \right) - \frac{\text{width}}{2}$$
$$v = \left(\text{height} \cdot \frac{j + \frac{1}{2}}{n_y} \right) - \frac{\text{height}}{2}$$

What this means is that for every (i, j) value in our grid, we can figure it out in the camera-coordinate system.

This connects the intersection in the scene/sensor to where we store the data.

Common bugs:

- Black screen: $\vec{w}$ is backwards
- The image is flipped: there is problem with your mapping; the corners that are being mapped is not correct

2.5 Ray Equation in Camera Space

We can now write our ray equation in camera space. We need to find the $\vec{e}$ parameter. We will never find $\vec{s}$ explicitly, but the direction $\vec{s} - \vec{e}$, what direction we need to cast the ray.

$\vec{e}$ is given; it is the position of the camera. $\vec{s} - \vec{e}$ is the ray. So, here it is, in camera space (where $\vec{e} = \vec{0}$), and given pixel coordinates (i, j):

$$\vec{p}(t) = t \begin{bmatrix} u(i) \\ v(j) \\ -d \end{bmatrix}$$

Where:

$$u(i) = \left(\text{width} \cdot \frac{i + \frac{1}{2}}{n_x} \right) - \frac{\text{width}}{2}$$
$$v(j) = \left(\text{height} \cdot \frac{j + \frac{1}{2}}{n_y} \right) - \frac{\text{height}}{2}$$

2.6 Ray Equation in World Space

We need to take the camera space equation to world space. You are given a position and the camera frame.

$$\vec{p}(t) = t\left(u(i)\vec{u} + v(j)\vec{v} - d\vec{w}\right) + \vec{e}$$

You can rewrite this as a matrix-vector equation:

$$\vec{p}(t) = t\begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix}\begin{bmatrix} u(i) \\ v(j) \\ -d \end{bmatrix} + \vec{e}$$

$\begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix}$ is an orthogonal matrix (it times its transpose is the identity). This is just a rigid body transform: we take our coordinate in (u, v, w), you rotate it so that it is looking at the same direction as the camera, and you move it.

Note that $\vec{e}$ is in our real-world space, so $\vec{e}$ is no longer always equal to $\vec{0}$. Instead, it is some world space position of the camera. Hence, it's always a good idea to comment what coordinate frame you're operating within your code.

2.7 Intersection Tests

Planes make good floors

Spheres are fun to ray-trace

Triangles are the building blocks of modern geometry

EVERYTHING BELOW IS WORLD SPACE AS ALL THE OBJECTS WE ARE DEALING WITH IS IN WORLD SPACE.

Compute ray equation in camera space, transform it into world space, then compute intersection.

2.7.1 Of Planes

I have the ray and a plane. I want to figure out where along the ray it intersects with the plane, t. We'll be using an implicit equation, and we'll substitute the plane equation in it and solve for t. When everything is nice and linear, root finding is simple.

You can define an infinite plane by having a point on the plane $\vec{p}_1$ and a normal vector $\vec{n}$.

If I pick any other point on the plane going from $\vec{p}_1$, they will always be orthogonal to $\vec{n}$. This means:

Let $q = \vec{n}^T \vec{p}_1$.

$$\vec{n}^T \left(\text{plane intersection} - \vec{p}_1\right) = 0$$

Just substitute "plane intersection" for $\vec{p}(t)$. When we rearrange, we get: Note that

$$s - e = \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix} \begin{bmatrix} u(i) \\ v(j) \\ -d \end{bmatrix}$$

$$t = \frac{\left(q - \vec{n}^T \vec{e}\right)}{\vec{n}^T \left(\vec{s} - \vec{e}\right)}$$

Problems? If any of them occur, do not render

1. We're not looking at the plane ($t < 1$)
2. The plane is parallel to the direction of the ray

If the ray is parallel to the plane, the dot product f $(\vec{s} - \vec{e})$, the direction of the ray, and the normal $\vec{n}^T$ will be 0. This results in a divide by zero. In this case, the way you detect that you miss the plane is to first compute $(\vec{s} - \vec{e})$, check if it is *very close to 0*, and if it is then the ray is parallel, and you don't process it.

If the ray is behind the *sensor*, don't render it.

When we're looking for an intersection test, we want that $t \geq 1$, $t < \infty$. That's the bound.

2.7.2 Of Spheres

Implicit surface:

$$f(x, y) = x^2 + y^2 - 1$$

Returns a negative number if I'm inside the shape and a positive number if I'm out of it. In the context of ray casting, this is a very useful surface representation.

The implicit equation of a sphere centered at the origin with radius r (so if you want to change the origin, $\vec{x}' = (\vec{x} - \vec{p}_1)$:

$$\vec{x}^T\vec{x} - r^2 = 0$$

Aside: if we let $\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$, then $\mathbf{x}^T\mathbf{x} = x^2 + y^2 + z^2$. If we substitute $(\mathbf{x} - \mathbf{p}_1)$ instead, where $\mathbf{p}_1 = (h, k, m)$, then $(\mathbf{x} - \mathbf{p}_1)^T (\mathbf{x} - \mathbf{p}_1)$ would be $(x-h)^2 + (y-k)^2 + (z-m)^2 - r^2 = 0$, which gives us full control over the sphere. So, really, our equation is:

$$(\mathbf{x} - \mathbf{p}_1)^T (\mathbf{x} - \mathbf{p}_1) - r^2 = 0$$

We need to find parameter t when we substitute in the ray equation:

$$\vec{p}(t)^T \vec{p}(t) - r^2 = 0$$

We end up with a quadratic equation:

Note that $\mathbf{s} - \mathbf{e} = \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix} \begin{bmatrix} u(i) \\ v(j) \\ -d \end{bmatrix}$ and more notably it does not contain t at all anywhere.

$$a = (\vec{s} - \vec{e})^T (\vec{s} - \vec{e})$$
$$b = 2\vec{e}^T (\vec{s} - \vec{e})$$
$$c = \vec{e}^T \vec{e} - r^2$$

Use the quadratic formula.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

There could be two roots (went through the sphere). One root (hit the side of the sphere), or 0 roots (meaning the ray did not hit).

What are acceptable values of t? $t \in [1, \infty)$ (So, less than ∞).

Two roots: base it off the smaller t. This surface occludes the further intersection.

If the camera is in the sphere, only account for t values ≥ 1.

The normal of a sphere is the vector from the origin to the intersection from any point.

2.7.3 Of Triangles

We aren't going to use an implicit representation anymore. This is because a triangle is defined by its boundaries. Boundaries are difficult to encode in an implicit representation.

The equation for a triangle goes like this (assume triangle's origin is $(0,\ 0,\ 0)$, on the origin):

$$\vec{p}_1 = \alpha\vec{t}_1 + \beta\vec{t}_2$$

It is an infinite plane defined by two tangent vectors $t_1,\ t_2$.

I can write the plane where the triangle lines on: $\vec{p}_1 = \alpha\vec{t}_1 + \beta\vec{t}_2$ so any point on the triangle can be written as them.

To detect if something is inside a triangle, I'll have to introduce some constraints on α and β. Firstly:

- $\alpha,\ \beta$ are all ≥ 0
- $\alpha + \beta \leq 1$

These constraints that $\vec{p}_1$ is inside our triangle.

We still want to do our intersection test, but things are a bit different. We have a parametric equation to work with.

Check via equating point on surface with point on ray

$$\begin{aligned}\vec{p}(t) &= \alpha\vec{t}_1 + \beta\vec{t}_2 \\ &\Rightarrow \vec{e} + t\,(\vec{s} - \vec{e}) = \alpha\vec{t}_1 + \beta\vec{t}_2 \\ &\Rightarrow \alpha\vec{t}_1 + \beta\vec{t}_2 - t\,(\vec{s} - \vec{e}) = \vec{e}\end{aligned}$$

We have an equation with 3 parameters. We need to solve for $\alpha,\ \beta,\ t$, and α, β must satisfy their constraints. We can rewrite this as a matrix-vector product.

$$\begin{bmatrix}\vec{t}_1 & \vec{t}_2 & -(\vec{s} - \vec{e})\end{bmatrix}\begin{bmatrix}\alpha \\ \beta \\ t\end{bmatrix} = \vec{e}$$

(It's $\mathbf{e} - \mathbf{c}_1$ where $\mathbf{c}_1$ is the first corner of the triangle)

Solve this equation, then check t, α, β – check if all of them meet our constraints. If all of these are true, then we've intersected this triangle.

You should probably get a linear algebra package to solve this.

2.8 Setting Pixel Color

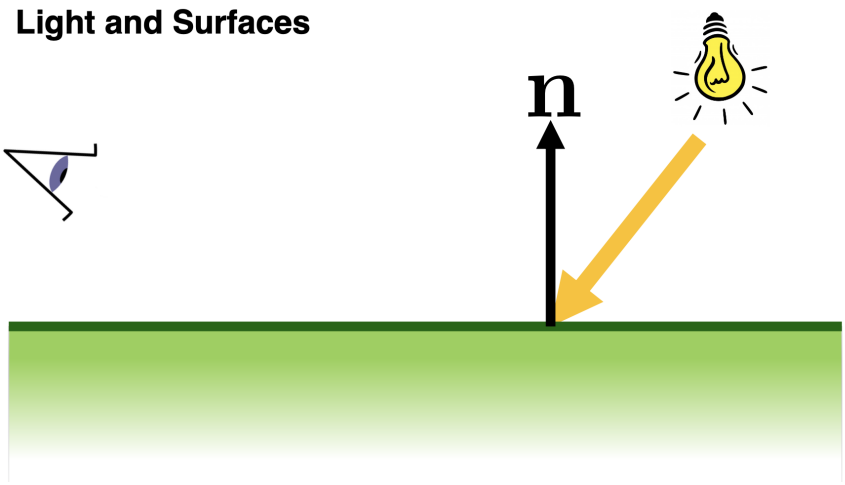
In ray casting, we're going to do a very boring type of this.

Output type:

- Object ID
  - Set pixel color that corresponds to object ID of the nearest object
- Surface normal
  - For any intersection point, you can calculate a vector that is orthogonal to the surface there
- Depth
  - t is a measure of how far an object is away, so you can make an image just by plotting t as a greyscale value

3 Ray Tracing

Assuming that everything is composed of small discs, we can talk about how light hits there. Because we have angular symmetry, we can view it through this model:



We'll assume we have a surface normal pointing towards the $+y$ direction.

How does light interact with this patch of surface? If we have a light in the scene, it will emit some energy onto the surface. Because we're doing ray tracing, we're always going to think of tracking that energy as rays.

We only need to consider what is happening around one point (look at the normal vector).

We have the direction of the light going to the origin of the coordinate system (at the tail of the normal vector).

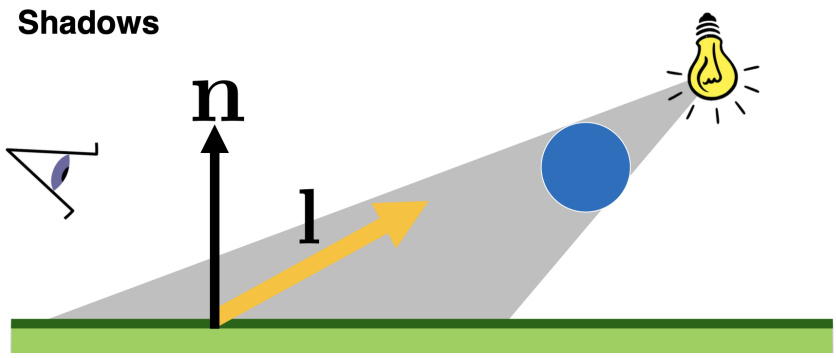
When light hits a surface, it can be reflected, refracted, and absorbed.

3.1 Types of Lights

There are two types of lights for the purposes of this class:

- Directional light: direction of light does not depend on the position of the object; light is very far away
- Point light

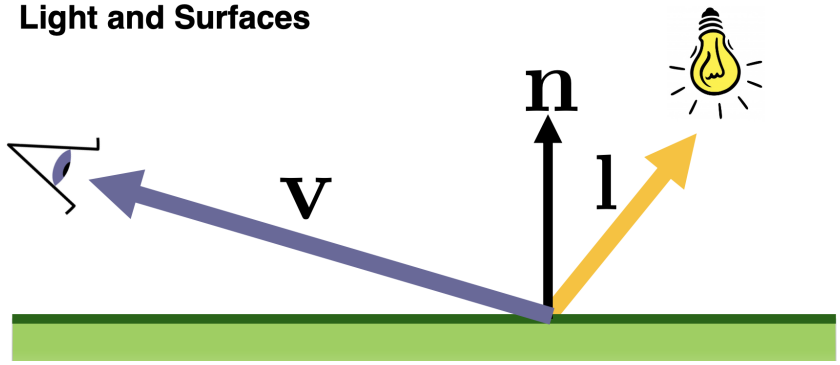
3.2 Shadows



l points to the light and is a normal vector. Ray cast from l, and if $\varepsilon < l <$ dist to light, there is a shadow. Set ε to a low value to prevent the surface from causing its own shadow, maybe 10^{-6} or something; it just has to work.

3.3 Modelling light reflected right into the camera

Light hits the surface and is reflected into the camera, and we have to determine a color for that. We'll introduce this vector v, the view vector, which points from the intersection point to the position of your camera in world space.



All the geometry we'll talk about from this point onward will use these three vectors.

We'll ask, how much light travels among v and that will define the color. We'll use simple models for this that can yield nice results. This is one of the things that made ray tracing practical when it was first proposed, is that you didn't need to follow light all over the place but just only worry about light paths that made it to the camera. This

is why ray tracing operates backwards from what you would expect. Here, we don't need to ray trace from the light.

Follow rays out of the camera and only do work if those rays hit the surface, abstracting away all the other interactions that we won't care about here.

When light hits a surface directly without bouncing from anything else, we divide this into three cases.

3.4 Diffuse / Lambertian

The amount of energy from a light source that falls on an area of surface depends on the angle of the surface to the light.

Meaning the angle between n and l. If θ were the angle between, the surface would light up proportional to θ. It does not depend on the view, v. This is a very simple version of light hits a surface and goes everywhere equally, so no matter where I am I will still see the same amount of light.

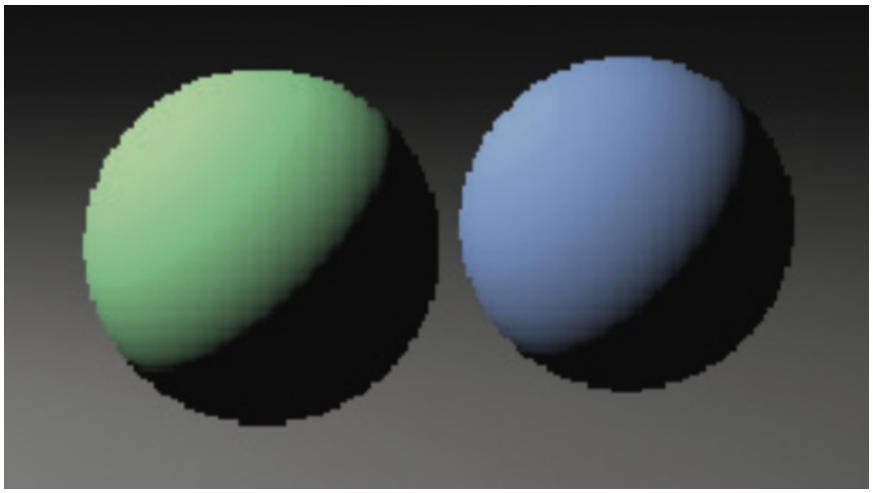
We need to translate this statement into math.

What's the way to turn two normal vectors into a scalar angle? Dot product. Lambertian shading says that the intensity L of a light that hits my camera sensor will be:

$$L = \vec{k}_d \max\left(0, \vec{n}^T \vec{l}\right)$$

The Max prevents self-shadowing. We'll get to k_d later. We aren't using linear proportionality of the angle; we're using cosine. $\vec{n} \cdot \vec{l} = \cos(\theta)$.

This has a very nice effect: as the angle gets bigger, the intensity goes down. If the light is orthogonal to the surface, the intensity is 0. As the light shines directly on the surface, we have high intensity.



Lambertian surfaces are very boring: they have no view dependences. On my laptop screen, as I move my head, I might notice that bright spots (specular) move around.

This means that *shiny* surfaces cannot be modeled by Lambertian surfaces. Chalk is a pure Lambertian surface (at least almost).

Lambert's quote says nothing about color. It only talks about the amount of energy, the intensity of light. It's true, we could make the model much more complicated to account for wavelengths of light to get nice color effects, but we're not going to do that. Rather, we're going to do a bit of a hack: we'll introduce a user parameter $\vec{k}_d$.

We'll cram all the other information we need into k_d. It is not going to be some scalar, but some vector quantity, and we're going to put the true color of the surface in k_d.

$\vec{k}_d$ is a vector quantity. For our intents and purposes, we'll store baseline diffuse color in it. This is how we convert this scalar value into a vector color value. All the lighting equations will have a k_d value and these are all user-parameterized colors.

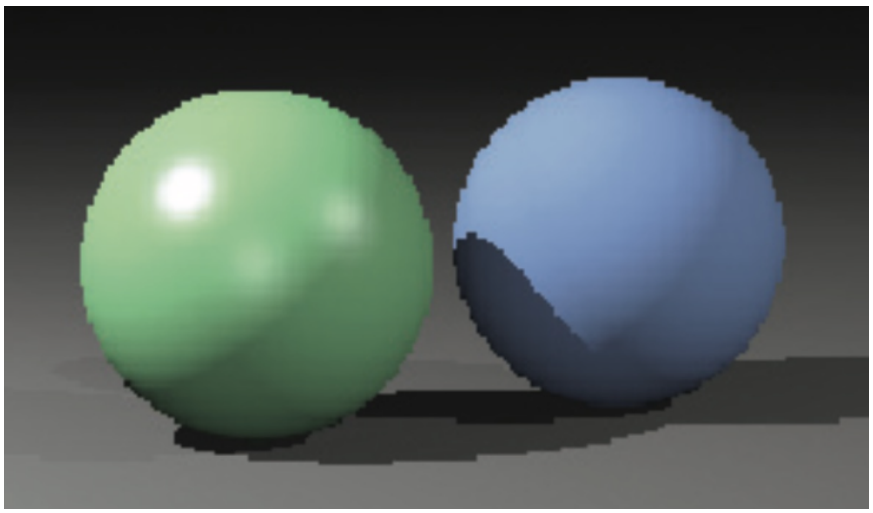
We typically have a $\vec{k}_d$ per o object. If I have a red bunny and a blue bunny in by scene, for the blue bunny I would have blue light coming out, so I would have $\vec{k}_d = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$, and for a red bunny I would have $\vec{k}_d = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$.

These $\vec{k}_d$ values do heavy lifting: it prevents a completely red light from being reflected off a completely blue surface.

Why do we do this? These things were invented not to do an exact computational simulation of light interaction; they were invented to make art. The people who were using these things were artists, animators, and designers who don't care about accuracy too much. They care more about the effects that they get out.

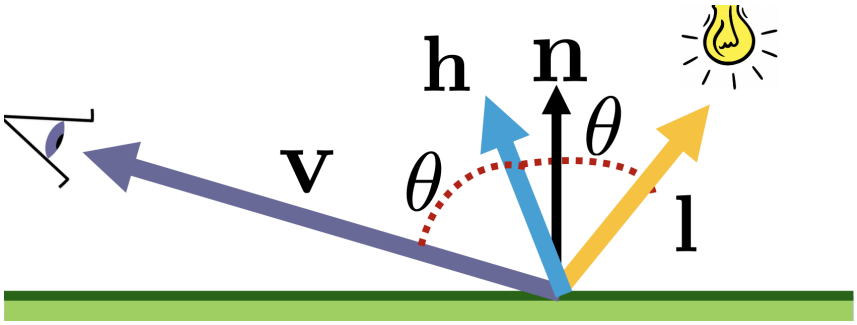
3.5 Specular Reflection

Specular reflection is dependent on the view position. As you move around, you will see these highlights move around. They pick up very small variations in the surface.



Specular reflection gives us these highlights. The model we use is the Blinn-Phong model.

The idea behind specular reflection is that we want to make the model where if we know the view direction and light direction, it produces the brightest light **when the angle between the view and the normal are equal.** Everything else, we want to fall off as that angle changes.



We'll compute a temporary vector $\vec{h}$ which bisects the angle between $\vec{v}$ and $\vec{l}$. We'll make another simplifying assumption: if $\vec{h}$ is aligned with $\vec{n}$, the angle of incidence and the angle of reflection from the surface to the normal is the same.

In other words, if $\vec{h} = \vec{n}$, I want the lightest and when $\vec{h} \cdot \vec{n} = 0$, I want the least light.

This means that the only difference between this model and the Lambertian shading model is that we're going to replace the light model with $\vec{h}$.

The half vector: This gives us a very good approximation.

$$\vec{h} = \frac{\vec{v} + \vec{l}}{||\vec{v} + \vec{l}||}$$

We're going to compute $\vec{l}$, $\vec{v}$, $\vec{h}$, and… $\theta_{\vec{h}}^{\vec{n}}$ is the angle between $\vec{h}$ and $\vec{n}$

$$L = k_d \max\left(0, \vec{n}^T\vec{l}\right) + k_s \max\left(0, \cos\left(\theta_{\vec{h}}^{\vec{n}}\right)\right)^p$$

k_s is the color specular – what color of the highlight do we want to have?

What is the p exponent? The effect we want to capture that we haven't explained by looking at the cosine, is that the smoother the surface is, the smaller and brighter the highlights are. If we have a perfectly polished surface, the highlights are small and sharp. If you sandpaper a surface, the highlights will be bigger and fuzzier.

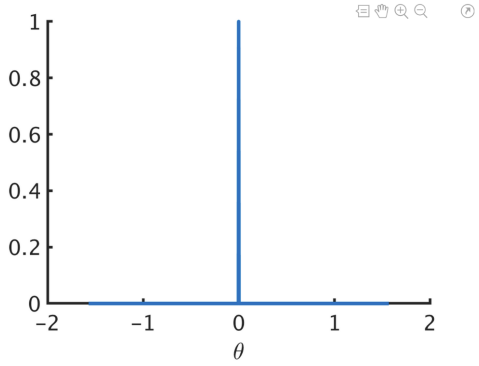
What's happening? In the Blinn-Phong model, when $\vec{h}$ is aligned with $\vec{n}$, let's say that we have this perfect reflector that has perfect angle of incidence = angle of reflection (aka no refraction – all lights go in a perfect mirror direction). Where would I have to put my eye to perceive that reflected light? I have to point it in one direction – the

perfectly reflected direction. Any other place I put the camera in the scene, the light ray misses my eye, and I don't see anything.

So, we know that $L \propto \theta_{\vec{h}}^{\vec{n}}$. I want to wrap this angle in a function that sends back a value of 1 if I'm in this perfectly reflected position and sends back a value of 0 if I'm in any other position. What does that function look like?

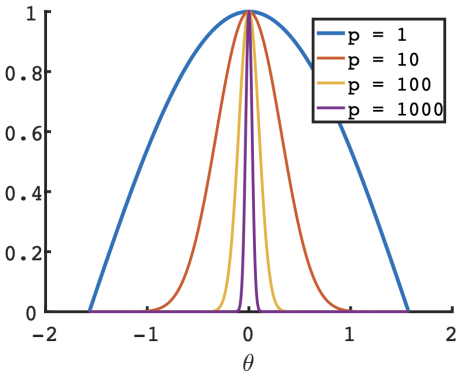
Measuring the Angle

$$L = k_s \max(0, \delta(\theta_{\mathbf{h}}^{\mathbf{n}}))$$



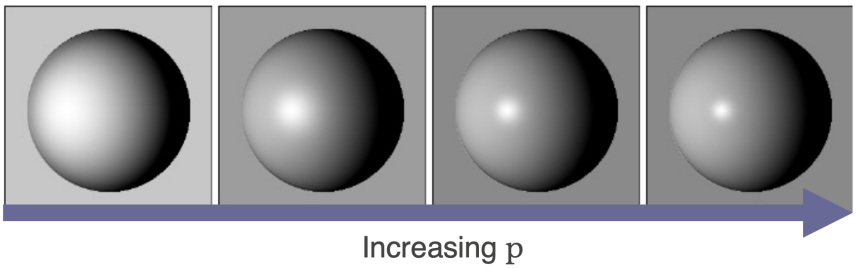
This isn't fun to render. We don't get any specular reflection, and we'll never get these surfaces in real life.

$$L = k_s \max(0, \cos(\theta_{\mathbf{h}}^{\mathbf{n}}))^p$$



So, I'd rather deal with this. L is always between 0 and 1 for all components.

This is a model for specular reflection that can cover all cases. p is the roughness exponent. Squaring this function pushes down values that are lower, being collapsed towards 0.



Increasing p brings us closer to approximating this perfectly shiny surface. We call this exponent, p, the roughness parameter.

When p is low, we're modeling a very rough surface. The higher I set p, the more I'll be modeling a complete smooth, shiny surface.

You may notice that the area under the curve isn't constant. You'll have to ignore this conservation of energy requirement because it's not something we'll cover. This is a simple model that looks good, and at the time could be evaluated with some computational efficiency. We stuck with it here because this is an intro class.

No, the brightness does not change depending on the distance between the model and the light for this model. Falloff for intensity based on distance is r^2, but this causes images to look very dark, so in graphics they use all kinds of different falloffs (like linear). Don't want something too dark.

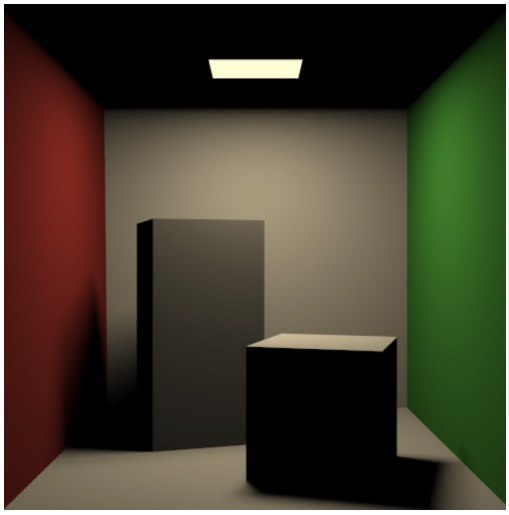
3.6 The Full Blinn-Phong Model

Light obeys the superposition principle: total amount of received light is the sum of light from all incoming sources.

$$L = \text{Lambertian} + \text{Specular}$$

When you do this in practice, you compute your intersection point on your mesh, the Lambertian intensity and specular intensity and you add them together.

3.7 Global Effects



This image has no global effects.

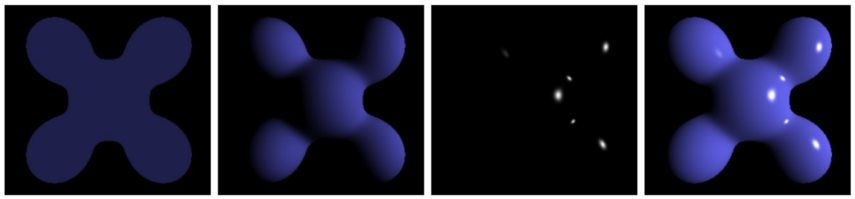
Global effects require tracking the light globally through the scene. How do we fix these errors?

The global effects we'll worry about here is reflection and refraction.

The fix for the fact that the image is very dark is going to be very upsetting: **add an extra constant term** k_a: the ambient term. This makes up for all the energy we're missing from *not* tracing all forms of light tracing around the room and getting into our eye. At least for simple ray tracing, to avoid all forms of more complex computations.

$$L = k_a + k_d \max\left(0,\ \vec{n}^T\vec{l}\right) + k_s \max\left(0, \cos\left(\theta_{\vec{h}}^{\vec{n}}\right)\right)^p$$

Here's our result:

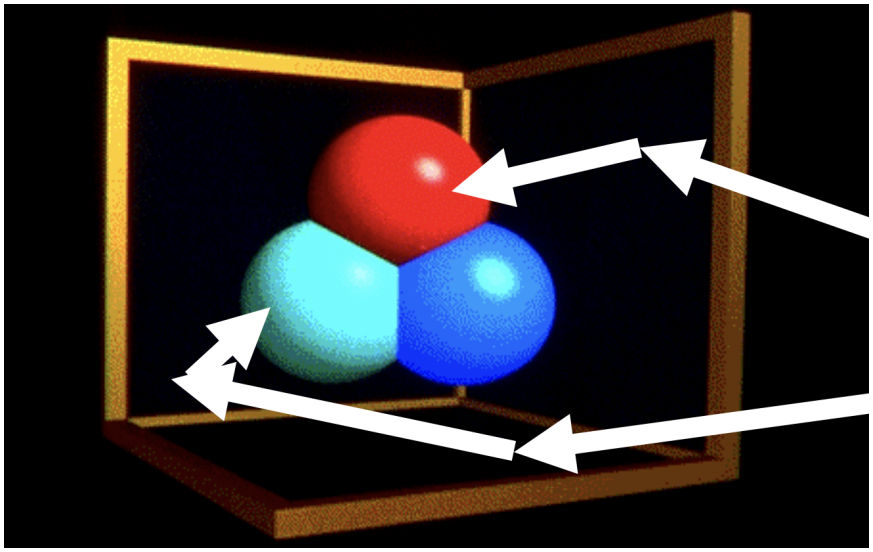


If we have multiple light sources, everything is super positioned. We would loop over

all lights and compute $k_d \max\left(0,\ \vec{n}^T\vec{l}\right) + k_s \max\left(0, \cos\left(\theta_{\vec{h}}^{\vec{n}}\right)\right)^p$ for all lights and add them up.

You'll get values that are greater than 1 and you'll have to scale them back to them being 0 and 1. This is like "squinting" or having your retinas shrink so it takes up less light – scaling the values back.

3.8 Mirror Reflections / Recursive Ray Tracing



What if we want to incorporate mirrors into our scene?

The case we have to take care off to render things like this: we're going to have some camera off screen.

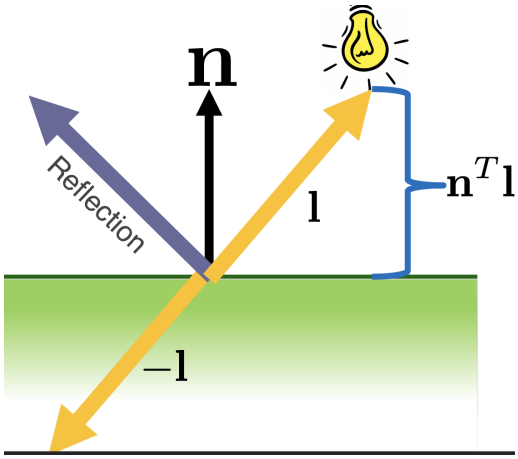
If we shoot a ray onto a scene, if it hits a mirror**, it should bounce off the mirror.** If we do this for every ray, we'll be able to get nice reflections. So, how do we compute this reflected direction?

At a surface, if we had some sort of light, how do we compute our reflection? We're going to do this using the most basic rules of optics: a reflected ray is reflected according to the relationship between the angle of incidence and the angle of reflection. In other words, that $\theta_i = \theta_r$.

All we want to do is figure out a simple formula to compute this.

If we have some light vector l, we can flip it to produce $-l$. How do we do that?

If we normalize our normal vector, we know that $\vec{n}^T\vec{l}$ is the projection of $\vec{l}$ onto the normal.



We take $\vec{l}$, negate it, do some other math and we end up reflecting it around the normal. We end up getting this equation for the reflection (I_3 is a 3×3 identity matrix, and we're doing an outer product so $\vec{n}\vec{n}^T$ outputs a 3×3 matrix):

$$-\left(I_3 - 2\vec{n}\vec{n}^T\right)\vec{l}$$

Because this reflection preserves the angle, this is the reflected direction of the ray.

How is this going to work in my ray tracer? This algorithm makes **every object in the scene a mirror** hence the recursion:

```
for each pixel in the image {
    pixel colour = rayTrace(viewRay, 0)
}
colour rayTrace(Ray, depth) {
    for each object in the scene {
        if(Intersect ray with object) {
            colour  = shading model
            if(depth < maxDepth)
                colour +=  rayTrace(reflectedRay,depth+1)
        }
    }
    return colour
}
```

Here, depth prevents stack overflow, and you should probably set it to a low value here otherwise everything's going to be too bright

Shading model is the Blinn-Phong shading model.

3.9 Transparency and Refraction

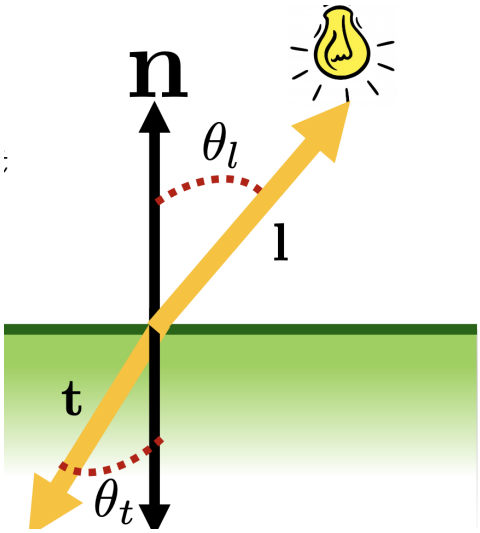
Some ray hits my object. Some of it gets reflects, but some of it gets refracted.

Snell's law: the optics law that describes refraction

$$c_l \sin(\theta_l) = c_t \sin(\theta_t)$$

Where c_l and c_t are the indices of refraction (how dense my object is to light, where I set for each material)

$$\mathbf{t} = -\frac{c_l}{c_t}\mathbf{l} + \left(\frac{c_l}{c_t}\cos\theta_1 - \cos\theta_t\right)\mathbf{n}$$



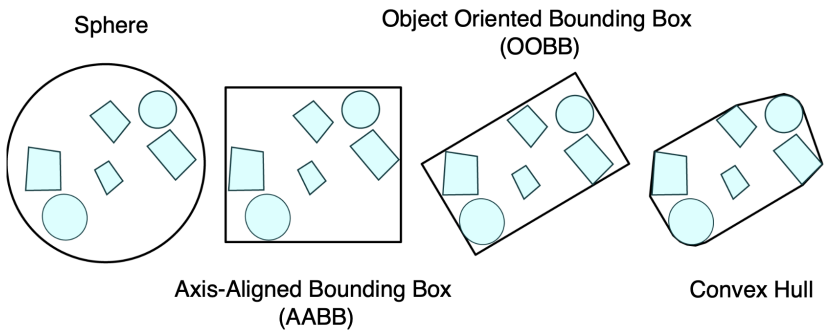
Just implement this into your code:

```
colour rayTrace(Ray, depth) {
    for each object in the scene {
        if(Intersect ray with object) {
            colour  = shading model
            if(depth < maxDepth)  {
                colour +=  rayTrace(reflectedRay,depth+1)
                colour +=  rayTrace(refractedRay,depth+1)
            }
        }
    }
    return colour
}
```

4 Bounding Volume Hierarchies

The data structure people use to accelerate intersection calculations.

Bounding volume hierarchies are made of bounding volumes. Bounding volumes are simple geometries that fully enclose a collection of other geometries.



Let's say I want to build a bounding volume for all of these shapes. If I want to choose a sphere, I only need to compute the center and the radius. Once I've computed these two numbers, I have a sphere that wraps around the geometry. If the ray caster hits any object, it hits the sphere. If it misses the sphere, it misses all the objects, saving calculation.

If spheres are so good, why do we build other types of bounding volumes? If we need something to fit to our geometry, something to do queries against, there's one other consideration.

We don't want to make our bounding volume too large. The bounding sphere is not tight to the geometry; it is not a tight bound. People developed other sort of bounding volume shapes.

One slightly better bounding volume we can make is the axis-aligned bounding box (AABB). Axis aligned, because the faces of the box are normal to directions in Euclidean space. This gives us a slightly tighter bound to this geometry.

We can take our box and rotate it to align to the distribution of our object.

If we want to get very fancy, use the convex hull. It is the polygon made by taking all the extremal vertices of the shape and connecting them. They have a very tight bounding box, but the collision checks are very expensive.

For the purposes of the assignment, spheres and AABBs will only be considered. Keep in mind **(it is an exam question every year)**, there is a tradeoff between simplicity of constructing bounding volume and intersection checks and the tightness of the bound.

4.1 Constructing a Bounding Sphere

Center, radius

Suppose I have a bunch of points, and I want a sphere that captures all the points. What's a good center point?

- Take the average of the positions of all the point
- Min, max, center of the two

Radius:

- Compute distance for all the points, take max. distance, and pad it a bit

Here it is:

$$\vec{c} = \frac{1}{n}\sum_{i=1}^{n}\vec{v}^{i}$$
$$r = \max\left(\vec{v}^{i} - \vec{c}\right)$$

Where $\vec{v}^{i}$ is vertex i. In case we are bounding spheres, you might want to increase the radius by the sphere on the edge or box the sphere.

Spheres don't work well with objects that are thin in one direction and wide in another direction.

4.2 Constructing an Axis-Aligned Bounding Box

$$x_m = \min\left(v_x^i\right)$$
$$x_M = \max\left(v_x^i\right)$$
$$y_m = \min\left(v_y^i\right)$$
$$y_M = \max\left(v_y^i\right)$$

Solve these 4 numbers and that's our axis-aligned bounding box.

4.2.1 Ray-AABB Intersection

In 2D, the box is made of 4 separating lines. In 3D, 6 separating planes. These planes define intervals that close the box. If I shoot a ray at a box, and it hits the box in two places, and I look at the x values of the two intersections, if it passes the box it has to line up in the interval. We will exploit this to do very fast set intersection calculations.

Take our box apart and look at the individual lines that make up the box. We'll ignore the corners, **so this is just a special case of plane intersection.**

Each of these planes are orthogonal to the axis. This means we'll always know the normal.

In 2D, we will have four intersection calculations. x_d and y_d are the x and y components of the direction vector. The similar thing can be said for x_e and y_e. So, $x_m - x_e$ is the x distance between x_m and $\vec{e}$.

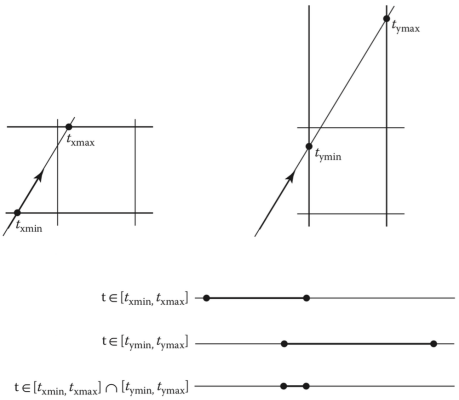
$$t_{x_m} = \frac{x_m - x_e}{x_d}$$
$$t_{xM} = \frac{x_M - x_e}{x_d}$$
$$t_{y_m} = \frac{y_m - y_e}{y_d}$$
$$t_{y_M} = \frac{y_M - y_e}{y_d}$$

We take these values of t and we try to determine whether the ray intersected with the boxes.

Suppose the ray was going from the left to the right, bottom to top.

If a ray hits the box, it will either go directly through or out from one of the sides. We can look at t_{x_m} and t_{y_M}.

So all we have to figure out, is that there are some set of intervals to see if we intersected the box.



If I have two intervals, $\left[t_{x_m},\, t_{x_M}\right]$ and $\left[t_{y_m},\, t_{y_M}\right]$, the intersections of the two is $\left[\max\left(t_{x_m},\, t_{y_m}\right), \min\left(t_{x_M},\, t_{y_M}\right)\right]$.

This will always give us two values. We need some chunk of ray to be within the box.

The ray must enter the box before it leaves the box. So, the ray intersects the box if:

$$\max\left(t_{x_m},\, t_{y_m}\right) < \min\left(t_{x_M},\, t_{y_M}\right)$$

If we don't have the assumption above, we're going to have to recalculate some variables:

$$\begin{aligned}
&\textbf{if } (x_d \geq 0) \textbf{ then} \\
&\quad t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d \\
&\quad t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d \\
&\textbf{else} \\
&\quad t_{\text{xmin}} = (x_{\text{max}} - x_e)/x_d \\
&\quad t_{\text{xmax}} = (x_{\text{min}} - x_e)/x_d
\end{aligned}$$

A similar argument can be made for the *y* and *z* directions.

If the ray origin is in the bounding box, check if the origin is in the bounding box. Do this by ensuring that you're in the correct side of all the plane (this is useful property for convex shapes).

This equation breaks if the ray is parallel with one of the planes. Figure out how to get around this case.

4.2.2 What happens in 3D?

In 3D, we're going to have intervals in z. Do this in the exact same way:

$$\max\left(t_{x_m},\, t_{y_m},\, t_{z_m}\right) < \min\left(t_{x_M},\, t_{y_M},\, t_{z_M}\right)$$

4.3 Building an Object-Oriented Bounding Box

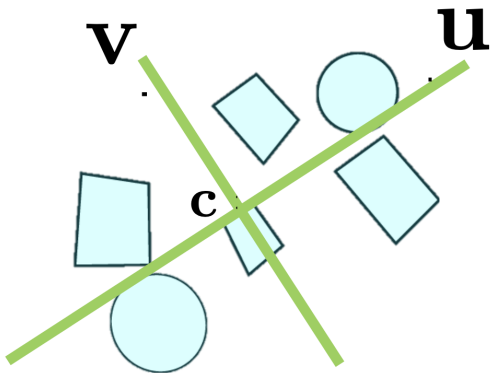
Sometimes, spheres and axis aligned bounding boxes don't really do a good job of keeping a tight bound of your geometry. For example, if your geometry has a large diagonal extent, it makes the axis-aligned bounding boxes huge.

Objected oriented bounding boxes work better. All you have to do is one extra trick: deal with the object alignment. Change the coordinate system. Pick a coordinate system that is well aligned with the distribution of the objects in the space. Find out the direction where the objects spread out the most in space.

There is a very common way to do this: principal-component analysis (PCA): PCA is a magical method in NumPy or MATLAB that: you give it a matrix of points and you get an orthogonal matrix back.

$$\mathbf{c} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{v}^i$$

$$\begin{bmatrix} \mathbf{u} & \mathbf{v} \end{bmatrix}$$



Find directions of maximum and minimum variance

Collision queries are done in that coordinate system. This is just exploiting the fact that you can move between different coordinate systems very effectively. Once you

can build an axis-aligned bounding box, with one function call (PCA), you can do the object-oriented bounding box.

4.4 Spatial Data Structures

How do we use bounding boxes? There are two types of subdivisions: object-based and spatial.

Our object-based data structures will be bounding volume hierarchies or BVHs.

BVHs are hierarchies of BVs represented by trees.

4.5 BVH Distance Queries

Calculated a bit differently due to some edge cases.

Use **BFS**. If you're at a node, if the distance to some child node is already greater than your current estimate, then you can discard that node, as no object in that child node can be any closer.

This means you would be culling away regions of implausible way.

When you hit a leaf node, you compute the minimum distance to the actual geometry. If the geometry is made of triangles, compute the distance from the point to every other vertex.

The code below here does not use BFS or any optimization, so you must modify it. You will know the furthest distance just by traversing one node, so that's some information you can get back just by one traversal.

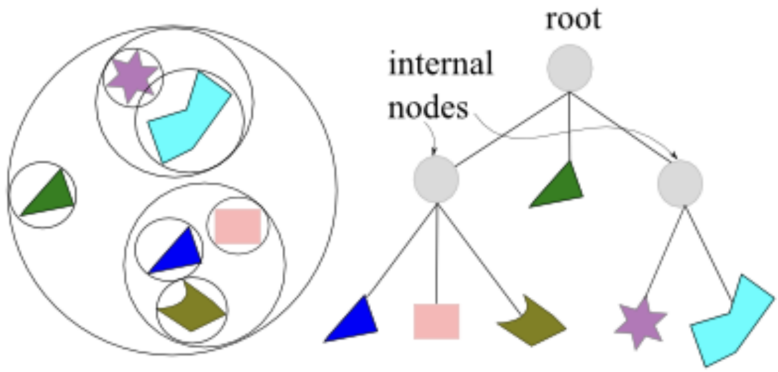
BVH Distance Queries

```
minDistance(bvNode, point, currentMin)
{
  d1=minDistance(bvNode.left, point, currentMin);
  d2=minDistance(bvNode.right, point, currentMin);

   if(min(d1,d2) > currentMin) {
     return currentMin
   }

   return min(d1,d2)
}
```

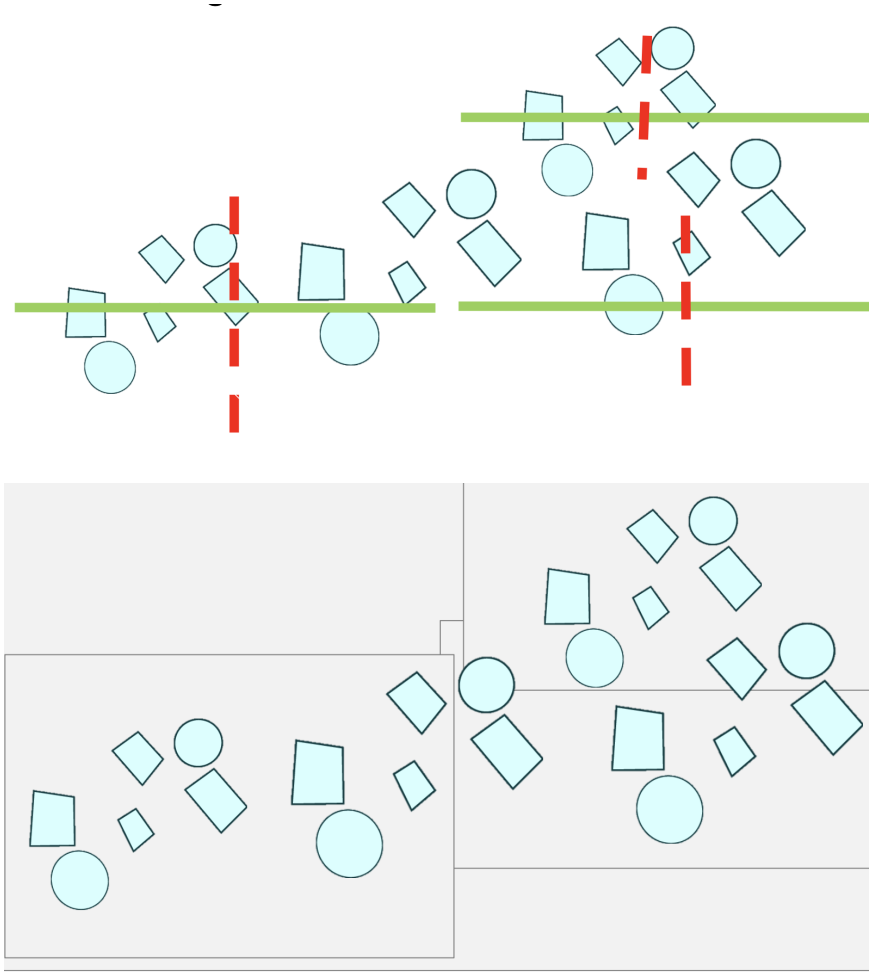
These algorithms apply for any sort of object-centric bounding volume hierarchy.



4.6 Constructing a BVH

Are there better ways to build BVHs? Yes. But anyways:

We have a bunch of objects, and we're going to put a BVH around them. We take the group of objects, and we pick an axis. We look at the extent of the objects in that axis. We split them up. Repeat.



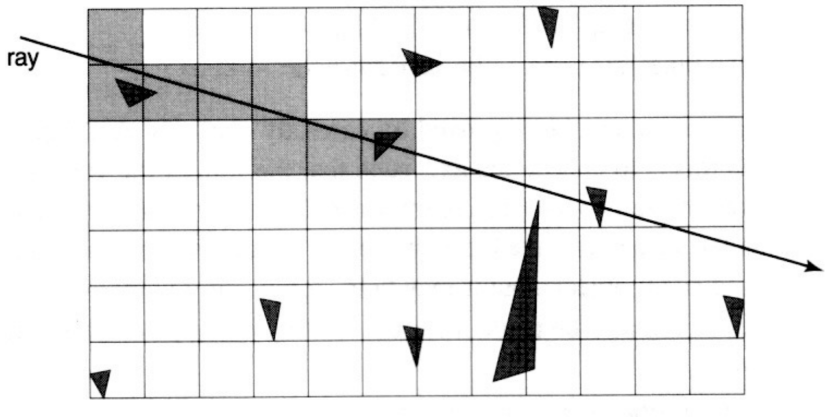
You put bounding volumes around them, and then you make a bounding volume that surrounds that of the child nodes.

One construction guarantees that the parent contains all of the child objects, and other construction guarantees parent contains all the child nodes.

4.7 Spatial Bounding Volume Hierarchies

Instead of dividing objects into groups, we'll be dividing the space the objects occupy into groups.

An axis-aligned grid is the simplest way to divide space.



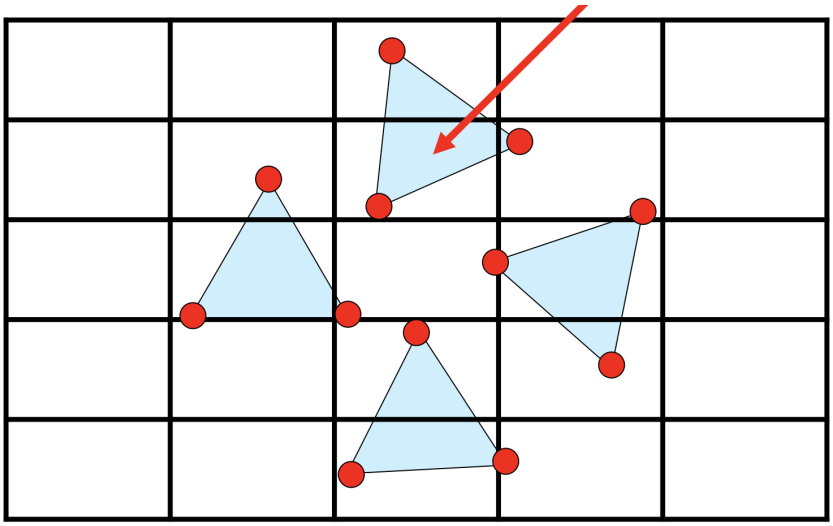
All special subdivision structures are a fancy hierarchical version of this type of axis-alignment.

So, how do we construct them?

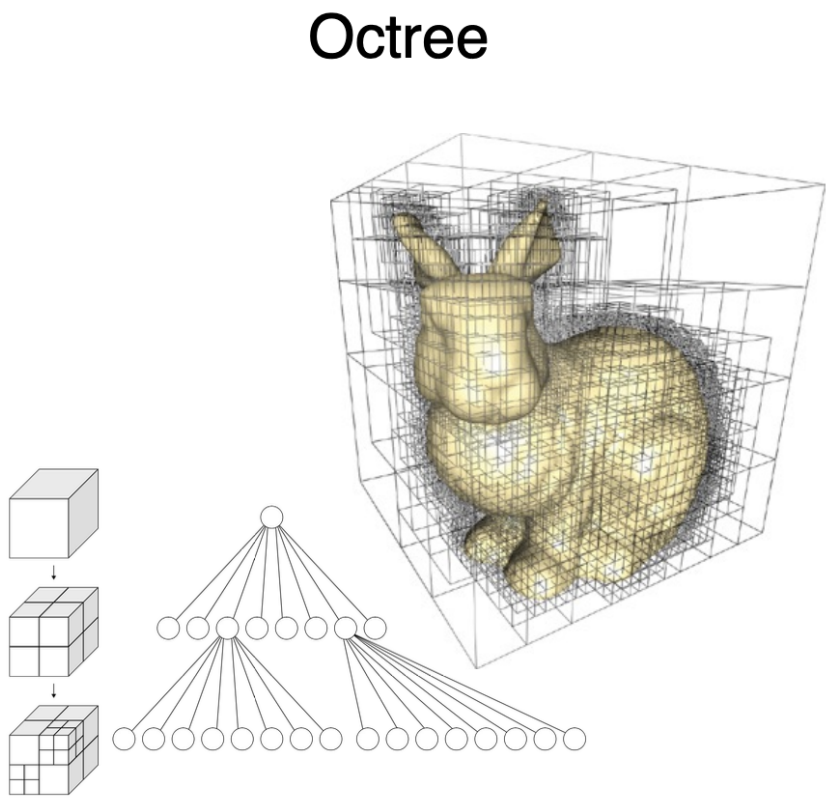
4.7.1 Construction

Take all the objects, take the vertices of an object. If all the vertices of an object are in one box, then that object belongs to that box.

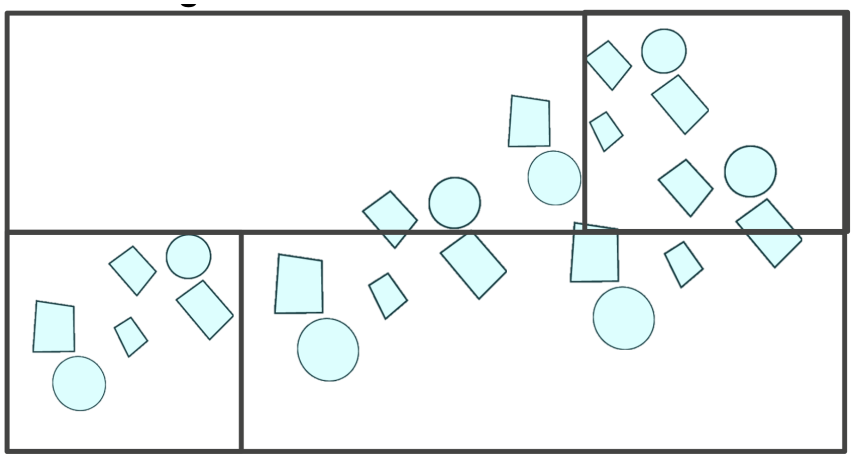
But what if an object has vertices that go over one box? Make it belong to all boxes.



In 3D, we would construct an octree: we divide space into 8, then again, and again, splitting things that aren't empty.



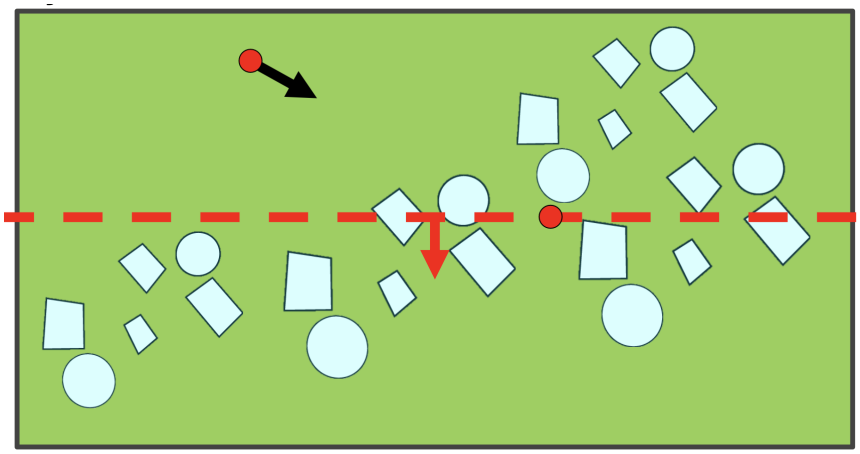
The construction process is the same, but we're working with space, not objects. We start with a root node, which is a box that encloses all of space. Continue splitting.



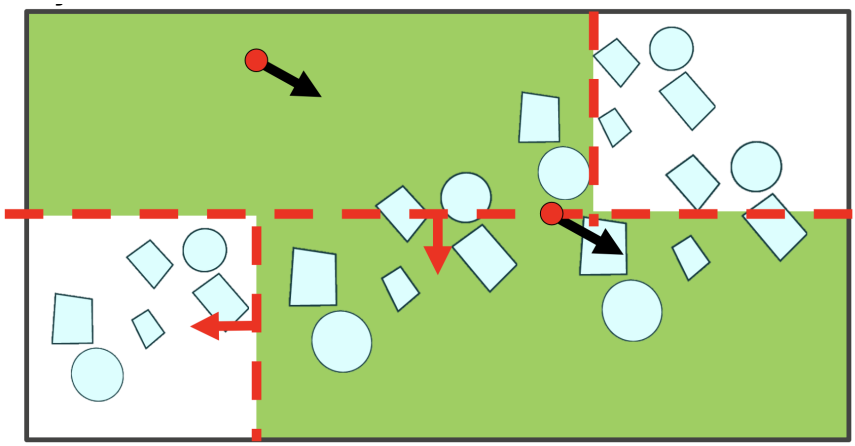
4.7.2 Ray Intersection Tests

DFS. View slides.

Ray comes from some point in space. Our goal with this is that we want to throw away regions of space that the ray can't possibly intersect with.



Check for intersection among the plane that splits.



Ignore children that the ray will never hit.

5 Meshes

5.1 Surfaces Representations in Graphics

There are two main types of surface representations in computer graphics:

- Implicit
- Parametric

How do I define an implicit surface? We have an equation, and we define it such that any point in the surface will return a constant value if we plug it into this equation, and for ease of use the level set is where the function would equal 0.

The parametric representation is the opposite: we're going to come up with some reduced set of variables and we're going to write down a set of equations that transforms 2D into 3D, and by definition every point produced by this equation is on the surface.

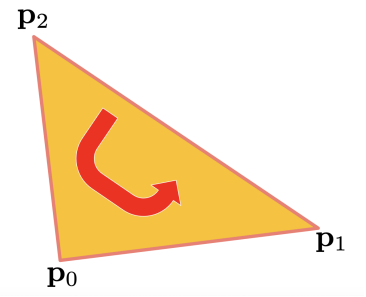
These representations are great, and the reason we use them in ray tracing, implicit surfaces especially, as they are easy to define and are easy to do collision checks. We only saw one parametric surface: the triangle; it is easy to define as a parametric surface, yet the collision check is not as nice as with an implicit surface.

What we'll do is take triangles to the max. We're going to do that by talking about how to efficiently represent huge bunches of triangles and do various operations on them. Such as find neighboring triangles, etc.

So here, we'll introduce the concept of a mesh. To do that, we're going to need to define a few terms. One thing we need to consider is we need the notion of *what is the outside of a shape, and what is the inside of a shape?* This is really important in commercial surface; if you walk up close to an object and you stick your camera through it, you'll see what you would normally see in no-clip mode. That's because lots of game engines define a front and back for the triangle. If the triangle is facing away from you, the triangle doesn't render. That's why X-raying works when you're inside a wall.

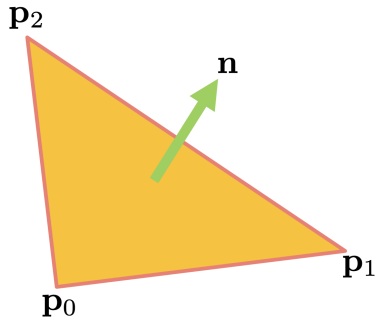
5.2 Winding a Triangle

Take a 3D triangle, enumerate the vertices. If we follow the path defined by the vertices, it rotates around the face. We define one direction of rotation to be front facing. If there were a clock in this triangle, we would go counterclockwise if we followed the order of the vertices.



If we flipped this triangle, we would observe a clockwise winding.

We need this convention as oftentimes, we'll need to compute surface normal, and the order in which we define points on the triangle will determine which direction the normal goes. It will always be normal to the triangle, but how do we define whether it goes out or goes in the screen?



These orderings come down to the way we're computing these normals, by doing cross products. We compute two tangent vectors by subtracting: $t_1 = p_0 - p_1$, $t_2 = p_0 - p_2$.

$$\vec{n} = \frac{\vec{t}_1 \times \vec{t}_2}{||\vec{t}_1 \times \vec{t}_2||}$$

You really want to ensure that your normals are pointing in a consistent direction. You

may get very weird effects that if you don't ensure that. The notion of winding gives us a guarantee that if we compute our normals using this cross-product formula, we get the right orientation.

One of the things we'll talk about is computing more interesting normals than this, but it is worth pointing out that as triangles have a planar shape, we only have one surface normal. For any point on the triangle, the surface normal is the same. If I do any calculation on my triangle mesh, any lighting, and I only use the "per-triangle" normal, your mesh is going to look very blocky. You're going to have this calculation that only depends on the per-triangle normal. If you have a mesh that is made out of a bunch of triangles, the shading will only change if you move to another triangle. So, how would I relax this restriction without making the geometry more complicated?

5.3 Interpolating color on the triangle

Let's say that I give color to each vertex of a triangle. What is the color at each point? This is an interpolation problem. One tool we use a lot when dealing with meshes is by using Barycentric coordinates:

$$\begin{aligned}\vec{p}_1 &= \alpha\vec{t}_1 + \beta\vec{t}_2 + \vec{p}_0 \\ \vec{p}(\alpha,\ \beta) &= \alpha\,(\vec{p}_1 - \vec{p}_0) + \beta\,(\vec{p}_2 - \vec{p}_0) + \vec{p}_0 \\ &= \alpha\vec{p}_1 + \beta\vec{p}_2 + (1 - \alpha - \beta)\vec{p}_0\end{aligned}$$

These α and β will be our first two barycentric coordinates. What can we do? We can unpack all of these equations, rearranging so we collect all like terms around all of the relevant vertex positions.

So now, I have $\vec{p}_1$, which is a position. If I know $\alpha,\ \beta$ and $\vec{p}_1,\ \vec{p}_2,\ \vec{p}_0$, then I can compute $\vec{p}$ for any point on the triangle. It works for any α, β for my triangle.

Surprisingly, I can replace $\vec{p}_0,\ \vec{p}_1,\ \vec{p}_2$ with color information. With this, I can interpolate color data. **That's the idea.** Use colors instead of points.

Typically, what people do, instead of keeping around $1 - \alpha - \beta$, they introduce a third barycentric coordinate: $\gamma = 1 - \alpha - \beta$ (constrained). This does not change the equation

or the math; we introduced this third extra barycentric coordinate and constrained it. When doing ray casting, we had to constrain α and β because we can't let them run out to $\pm\infty$.

$$\alpha \geq 0,\ \beta \geq 0,\ \alpha + \beta \leq 1$$

Once I have α and β, γ is defined. If I want to use this with my ray tracer, I get the α, β the ray tracer computers and I can use them to get the color back.

RGB space is just another 3D space, where every point is a color.

5.4 Storing Triangles / Meshes

A triangle soup is a list of unconnected triangles. You have no idea how they are connected in space. This means you can't find neighboring triangles, as no information is stored that determines that.

Meshes have connectivity information. Triangles that share a vertex know that they share that vertex.

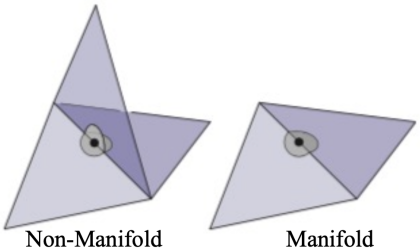
5.4.1 Topology

Topology is concerned with the connectivity of a mesh (surface).

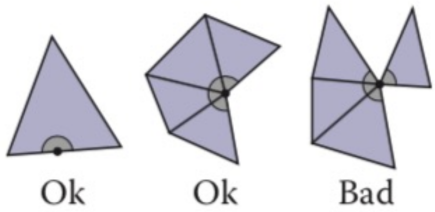
Many algorithms are easier to implement when connectivity data is available.

We are going to assume that are meshes are 2-manifolds. The concepts are really easy.

A 2-manifold is a surface for which the neighborhood around all points (*and only touching the point*, *locally*) can be flattened onto the plane in some way. ($\forall$ points, $\exists$ way to flatten them) If we can smash it in *one* way without it overlapping, then it is a

2-manifold. 

(Really, a 2-manifold is a 2D surface embedded in a higher dimension)



Every edge (not on a vertex) attaches to at most two triangles.

Manifolds must be continuous, so that's why the surface on the right above is not a manifold.

5.5 Watertight

A watertight mesh is a mesh with no holes in it.

5.6 Geometry

Geometrically, a mesh is a piecewise planar surface. Almost everywhere, it is planar. Exceptions are at the edges where triangles join.

Often, it's a piecewise planar approximation of a smooth surface.

5.7 Storing Triangle Meshes

What do we care about?

1. Compactness. Store as little data as we can to represent a shape.

2. Efficiency of queries
    a. All vertices of a triangle
    b. All triangles around a vertex
    c. Neighboring triangles of a triangle

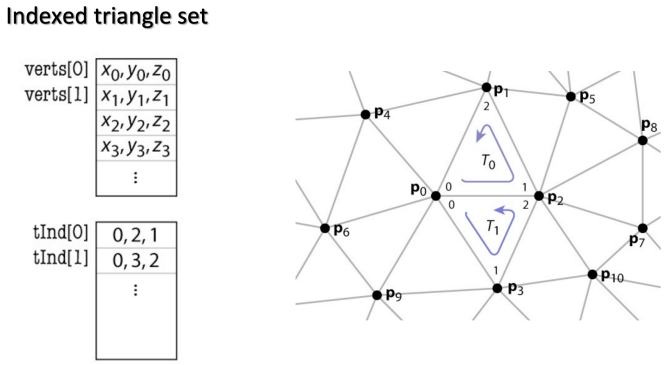
Methods?

5.7.1 Separate Triangles

We have always been doing this. We store the positions of the three vertices that make up that triangle, storing the in the correct winding order (counterclockwise when viewed from outside).

5.7.2 Indexed Triangle Set

More compact storage of the above.



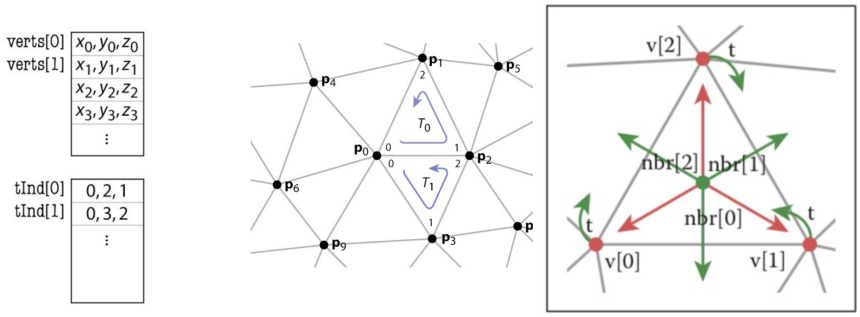
Avoids duplicating vertex data and we finally get connectivity data.

This gives us all the information we need but the queries are unbelievably slow.

5.7.3 Triangle-Neighbor Data Structure

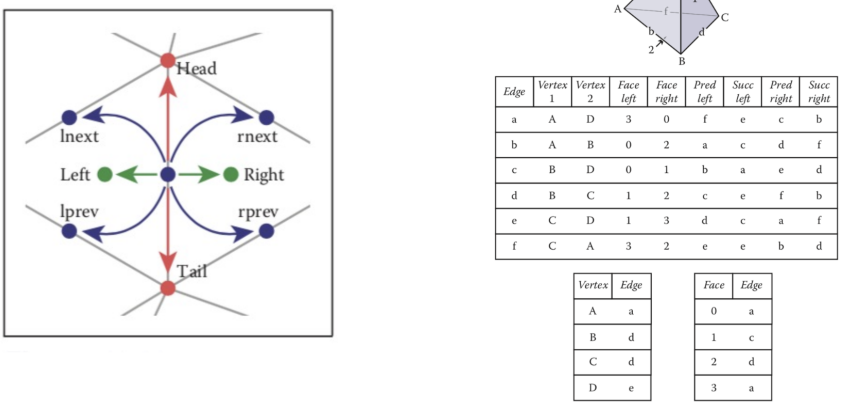
Trading memory for performance. With the addition of the above, each triangle now stores the set of neighboring triangles, but now neighboring triangle queries are con-

stant time.



5.7.4 Winged-Edge Data Structure

If we're going to store something on an edge, there is a fixed amount of data, which is two triangles.



Edge	Vertex 1	Vertex 2	Face left	Face right	Pred left	Succ left	Pred right	Succ right
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	e	b	d

Vertex	Edge
A	a
B	d
C	d
D	e

Face	Edge
0	a
1	c
2	d
3	a

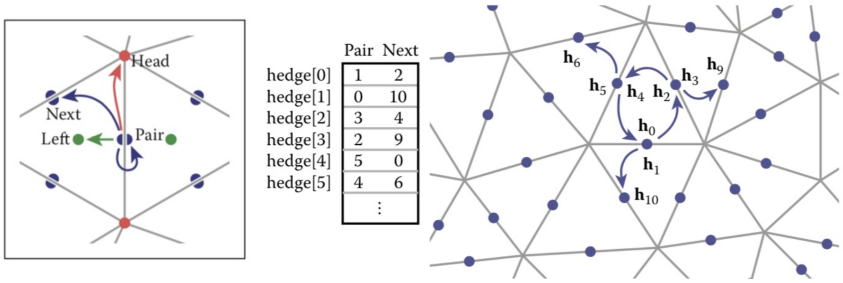
Any edge will be connected to two triangles. What we are going to store is a pointer to each of the two triangles.

We walk around the triangle using pointers and we gather all the vertices. If we want to get 3 vertices, it is a constant time query. If we want to know what triangles share the edge, it's really easy: constant time query. If we want to find the triangles that are all connected to any vertex, you know what to do.

5.7.5 Half-Edge

A half-edge

Half-Edge Data Structure

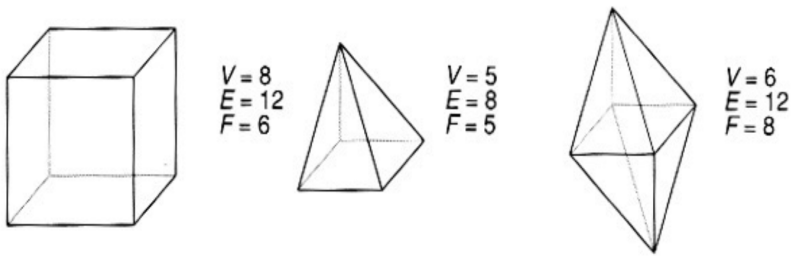


5.8 Relationships between Primitive Types

What is the relationship between the number of vertices, the number of edges, and the number of triangles in a mesh?

It turns out that we have Euler's formula:

$$V + F - E = 2$$



Interesting relationships:

$$3F = 2E$$

$$\text{Half edges} = 2E = 3F$$

Every triangle has to be in a closed mesh. Meaning it has to be attached to three half-edges, so Half edges = $3F$

5.9 Data on Meshes

Often need to store additional information besides just the geometry. Can store additional data at faces, vertices, or edges. For example:

- Colors stored on faces, for faceted objects
- Information about sharp creases stored at edges

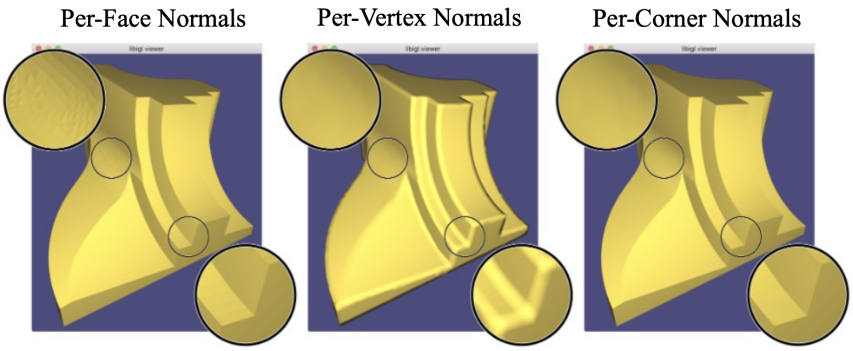
5.10 Surface Normals

When you have smooth geometry like cylinders and spheres, the surface normals are smoothly variant. This is very different if we were to deal with triangle meshes, as the normal is going to change abruptly when we change to new triangles.

If you just used normals for rendering that are at the faces, one surface normal per triangles, you get this very faceted view (low poly).

If you compute new normals for your mesh but you store them on the vertices of your mesh, then you can interpolate them across the surface. Now, you get a very different normal and everything gets smoothed out. The problem is that when you have sharp features like corners, they will be smoothed out.

Solution? If you detect a big change in orientation, store the normal differently. This is per-corner normals.



5.10.1 Per vertex

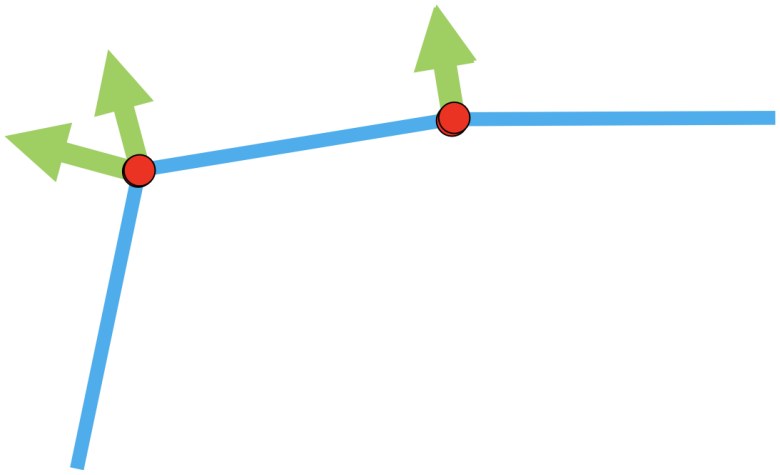
Average the normals of the surrounding triangles.

5.10.2 Per corner

Disambiguate between sharp and smooth. Look at the angle between the two normals using the dot product. If the angle is big, it's a sharp corner. We'll do two different operations depending on whether the corner is sharp.

The tricky part? When we store per-corner normals, we are storing as if the data was on a triangle soup. For every triangle, I'm going to store three normals. If I have two triangles that are attached, I am going to store two normals at that vertex.

For smooth: store one normal. For sharp: store two normals.



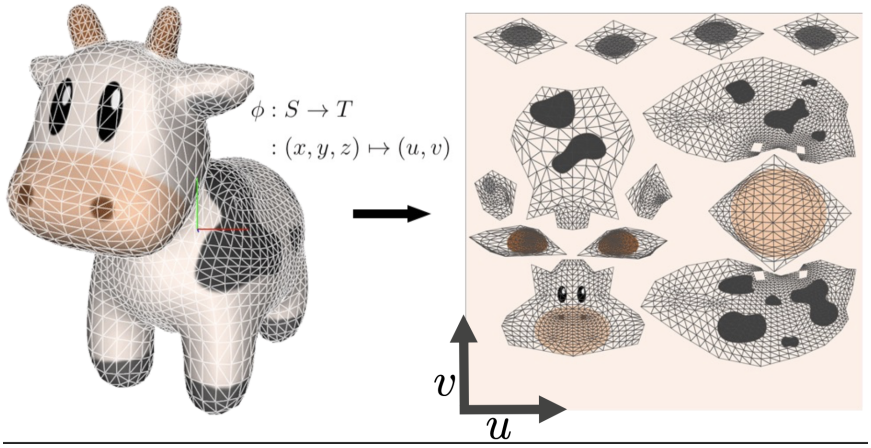
5.11 Quad Meshes

Meshes using quadrilaterals instead

Application? Subdivision. The particular brand of subdivision will only apply to quad meshes.

5.12 Texture Maps

It allows you to represent some of the detail on your mesh in a non-geometric way. If you have some very detailed mesh you want to render, if you wanted to get these nice round spots on the cow, you will have to make sure you would have edges on the mesh on the right place… it would be very tedious. Instead of doing this geometrically, we will store high-resolution images in a 2D space, and we will use these patches of images to wallpaper the 3D object. This is texture mapping and relies on a UV map. For every point on the triangle, you need to go from a 3D point to a (u, v) point in 2D space. When we render, we compute the (u, v) coordinate from (x, y, z), look up the color in (u, v) space, and we render that color. The difficult bit is taking the image, spreading it out in the 2D space, and assigning what are the UV coordinates to the 3D object.



For certain types of maps, this is not so difficult. If we want to make a planar texture map.

5.13 Subdivision Surfaces

When you render something a bunch of times and want it to be good, you want a smooth surface. If you do it infinitely many times, you will get a smooth surface.

5.13.1 Catmull-Clark Subdivision

The nice thing about subdivision surfaces is that we can get our smooth surfaces by iteratively repeating subdivision.

6 Shaders

The graphics pipeline is the order in which the graphics card processes the information that you send it.

6.1 Rasterization

Loop over all objects in the scene (or every triangle) and back-project that triangle onto the pixel grid. If we only look at planar triangles, we project the vertices onto the pixel grid. You'll get a boundary on the pixel grid, then you'll check if the pixels are on that triangle.

The reason that this algorithm is popular is that all of the operations are quick and parallelizable. Filling in pixels, checking if they're in the triangle is a well-known and fast-on-hardware operation.

The problem is that vs. ray tracing, the reason why we don't talk about rasterizing first is because we use a lot of rasterization techniques we get for free when we do ray tracing. In raytracing, it is the conic shape of the ray we fire through the pixel grid that gives perspective in the image. If an object is closer, more rays will hit it so more pixels will be activated, and the further away the less density of rays that hit it and the object will appear smaller.

We don't get that for free in rasterization.

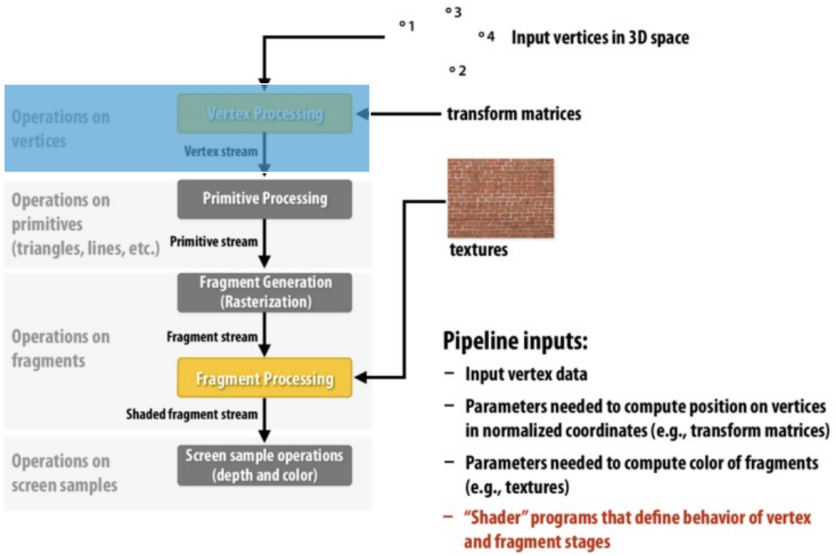
We don't also get easy depth culling. The t values are very nice; it tells us if an object is in front or behind the camera, and whether an object is in front or behind the other.

Again, that's something we don't get for free in rasterization.

Note that we can parallelize ray tracing. You could argue that RT is more efficient for big scenes, but it is hard to implement in a fast way. Large data structures are costly.

Graphics cards are linear algebra machines. They were built for that because all of these projections, transforms we'll be doing will be implemented as matrix-vector products. You'll have to specify the appropriate projection matrix.

6.2 Modern Graphics Pipeline



You send in your mesh as input vertices to the pipeline. You also send in a set of indices.

The first step is the vertex processing steps. You'll move all the vertices around in 3D space. All of these transformations will be specified as matrices.

Modern graphics pipelines are programmable, and the programmable aspect will manifest itself in three stages of varying granularity.

6.2.1 Vertex processing

A vertex shader takes in one primitive set of vertices at a time, takes all your transformation matrices, and outputs a transformed vertex. Using this programmability, you can do all sorts of things to vertices on the card. If you need to make something rotate, send in a rotation matrix.

When you see water in games, the way that works is, the geometry that is being sent to the graphics card is a plane being tessellated into a ton of triangle, and your card will displace the vertices by a sinusoidal wave.

Note how we store triangles: V, F as seen in the assignments

6.2.2 Primitive processing

Geometry shaders.

They take in all the vertices and the connectivity (things that define triangles), and you can finally do operations on triangles.

Most common operation is tessellating the geometry.

Example: The geometry shader. it can take in a single triangle and can dice it up into a ton of triangles. It can generate a ton of vertices in a card, as if it up samples the geometry.

6.2.3 Fragment processing

A fragment is a small part of geometry that lies in a pixel.

We then take all of this geometry and project it to the screen. We'll check each pixel, check if it is inside the new geometry, and if it is, it will color in that pixel. When it checks if a triangle is inside a pixel, it dices up the triangle. It looks at a fragment of a triangle and checks if it is inside the pixel. That coloring in of the fragment is also programmable, called *fragment processing.* There is a fragment shader that runs in parallel for every fragment. The input to the fragment shader, with some user input like textures and colors, you get the **output color** of the fragment.

Fragment processing occurs in parallel for each fragment.

> People write arbitrarily complicated fragment shaders to do all kinds of things. Shadertoy.com is a website where you can write shaders in a web interface, and people can also submit things and they're all stored here. People can do all sorts of crazy stuff there.

At the fragment shader, you can set what information is passed between different parts of the pipeline. There are two types of variables that the shader performs:

- Uniform: set for all programs that execute. If I really needed to know the center of a sphere, it is the same for all vertices, I can see it as a uniform and pass this variable to all vertices. It's like a global variable.
- Varying variable: A variable that is changing over the face of a triangle. One example is, maybe a surface normal. If you have per-vertex normals, you specify a normal at each vertex on your mesh. You'll tell your fragment shader that this is a varying variable, so it will linearly interpolate it across the triangle.
  - Want to do fancy shading? BP shading, if you want to compute the diffuse light you need the normal and light vector at the pixel. The way to do that in a fragment shader, is send in a normal and light vector in the vertices, and with the fragment shader you get nice interpolated normal and light vectors.
  - In A5's webpage, when you linear interpolate your normals, it makes your mesh nice and smooth. You have a fragment shader that does the BP lighting, and you get a very smooth mesh.
  - When the first cards came out with this capability, the demo was the following: you send in 2 triangles and a square, and you set on the square the normals as if they were taken from a sphere. The fragment shader does all this interpolation, you take the code you write for BP shading, paste it into the fragment shader, run it, and when you look at the square polygon, it renders like a perfect sphere.

Fragment shaders are cool.

We get the final output color of the fragment.

You can do screen space operations then.

6.2.4 Screen sample operations

All 2D-concerned things on your screen that can be done on Photoshop.

Masking?

Pixel image-level effects.

Recap: Vertices → primitives → fragments → image

6.3 Linear Transformations

Vertex shaders can move your vertices around.

GPU are linear algebra machines. They love doing linear algebra, and in particular they love doing linear transforms. A linear transform, for vectors, is any operation that can be expressed as a matrix.

$$A\vec{x} = \vec{b}$$

What are some transformations we can do?

The origin never moves when you do linear transforms.

Here is the list of affine transformations:

6.3.1 Scale

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Scaling factor.

6.3.2 Rotation

$$\text{rotate}(\phi) = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix}$$

Rotation matrices are **orthonormal**, meaning their inverses are their transpose. Angles go counterclockwise.

6.3.3 Shear

Adds some component of a y value to an x value or vice versa.

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + sy \\ y \end{bmatrix}$$

Square to a parallelogram.

6.4 Homogenous Coordinates / 2D Linear Transforms with Translation

It would be great if we could write a translation using these matrices:

$$T \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \vec{t}$$

It is impossible to add a constant term by 2D matrix transforms.

So what can we do? Homogenous coordinates. We're going to add one extra dimension to all our variables, but we're always going to set that coefficient to one.

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + t_x \\ a_{21}x + a_{22}y + t_y \\ 1 \end{bmatrix}$$

If we want to convert a point from standard notation to **homogenous coordinates,** we add a 1 to it. If it's a point / vertex in space, we should be able to translate it around.

A point in homogenous coordinates is a homogenous point.

Geometrically, we added this extra 3D space and now we're saying is that happens on when the extra component $w = 1$.

$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$, vectors in these forms are unaffected by the translation component of transformation matrices. Useful for velocity vectors.

Every final exam has questions on homogenous vectors.

Scaling of homogenous projects are equal. Two homogenous points are **equal if they are proportional.**

Vectors, unlike positions, measure change.

6.5 Normals Under Transformations

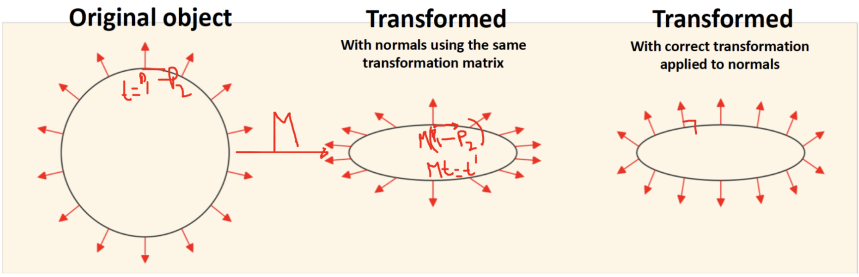
How do you transform normals when your shape has been transformed with a linear transformation?

We could:

- Do nothing (use the untransformed normals) – but then your lighting is going to look very ugly
- Transform every vertex of our sphere using the transformation matrix A, and apply A to all surface normals. The problem is that this produces very strange results and very incorrect surface normals.
- What we want are the correct surface normals: what makes them surface normals? It's normal to the surface.

Using this idea, we can figure out the right way to do this.

Let's say that n' is one of the new surfaces normals, and t' is a vector that is tangent to the surface. We know that they exist, but we never compute t, ever. If M was the transformation matrix to the shape:



$$n'^T t' = 0$$
$$(Xn)^T (Mt) = 0$$

We see that normals don't transform with M, they transform with some other matrix.

We want:

$$n^T X^T M t = 0$$

So, if $X = \left(M^{-1}\right)^T$, then

$$n^T X^T M t = n^T M^{-1} M t = n^T t = 0$$

So, in summary, we need to update our surface normals using $X = \left(M^{-1}\right)^T$

The hardest thing when figuring these out is figuring out what we need to build. t is used to make an argument, but you never build it. You just have to convince yourself that it exists.

6.6 Composing Transformations

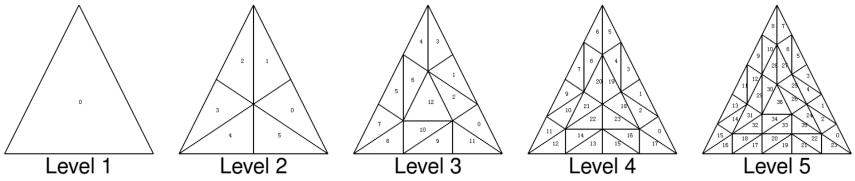
Want to stretch and rotate an object? Use matrix multiplications.

Matrix multiplication does not commute. Transformations go from the right to the left.

If you need to translate something on a different anchor point than the origin, you'll have to translate first then translate back when you're done. You get these translation-transform-translate back sandwiches. The mistake is that you'll get the translation to and from the origin backwards, so look out.

6.7 Tessellation Shader

The tessellation shader will take in one triangle. The GPU driver will accept how many chunks along each edge you want to divide your triangle into, and it will build this mesh for you. Downstream, all calculations will be based on the result.



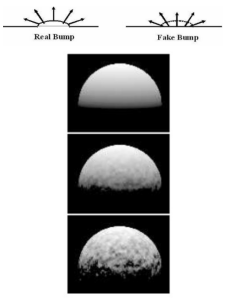
6.8 Fragment Shader Tricks

We've already seen one fragment shader trick: texture mapping. You can take in the texture coordinates and hand them to an image, and it will look up color / data for the image and you'll set the output data / color for the image.

You can put all kinds of data into an image. One of the first non-standard uses was a technique called normal mapping.

It is a way to turn smooth geometry into very bumpy geometry without changing the geometry. If you have a smooth surface and a slightly bumpy surface, the bumpy surface normals have a lot of variation.

What you can do is store surface normals in a texture. If you have some bumpy texture you want to apply, store bumpy surface normals in your texture and in your shader look up the normals from the texture and the lighting will be based on those normals.

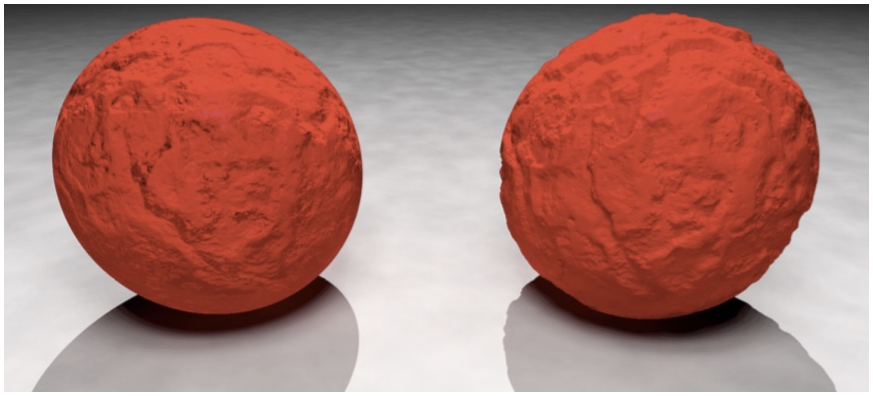


Here's a trick:

The image on the left is just a flat piece of paper with some shadows drawn on it.

6.9 Displacement and Bump mapping

No shadows impacted.



Solution? Displacement and bump mapping.

Instead of just changing the surface normals, you can change the geometry. The displacement map tells us how far to push the vertices in the direction of the normal.

In the shader, for each vertex, we can look up the displacement for the map and push out the vertex in the normal direction. This allows us to encode geometry using this type of mapping.

And this is why you need tessellation shaders – you need this so that triangles can be displaced at high variations in a small area.

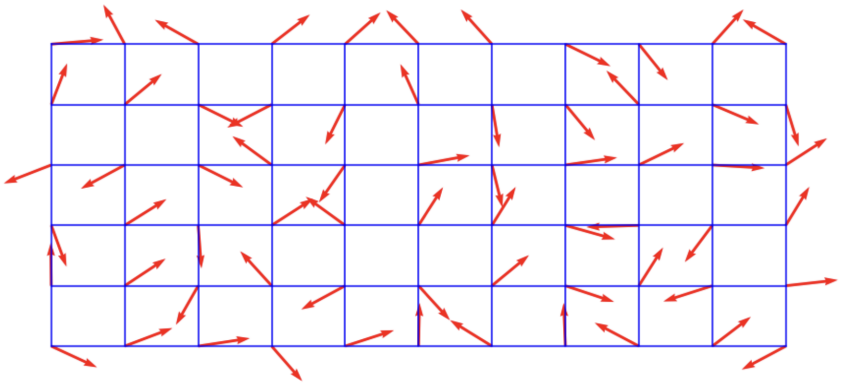
Everything here is a bunch of triangles. Bunches of triangles is what GPUs like.

As long as you can render objects very quickly, you can rasterize shadows.

6.10 Perlin Noise

Procedural means "not directly specified by an artist". We have an algorithm that takes in some parameters and randomly produces based on these parameters.

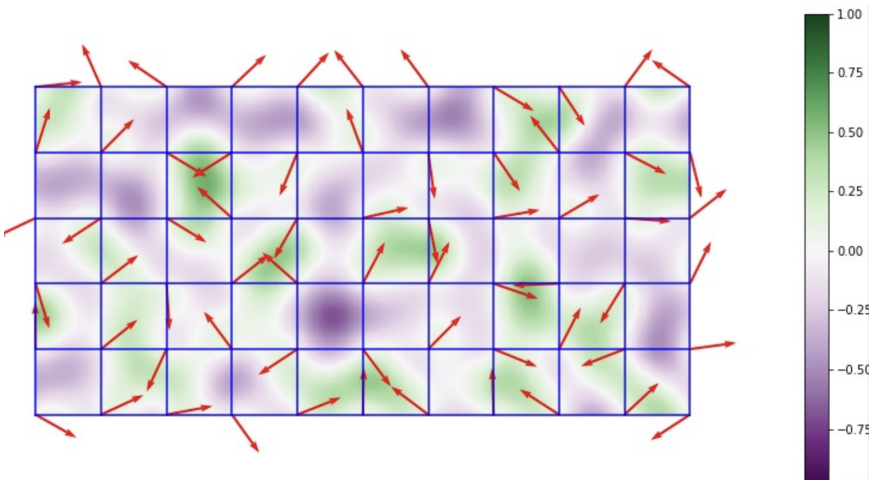
The reason why people like Perlin noise is that it's very expressive and can generate all kinds of interesting shapes and colors, and it can be easily implemented on the GPU.



The high-level idea:

- Make a grid of some size.
- We want to fill in this grid with information, colors, and displacements and it will function as an implicit texture.
- For each vertex, you randomly assign a normalized vector.

- Pick two colors, set –1 to one color and +1 to another color, and we define a color map so every value in between maps to some combination of these colors and we randomly assign values –1 to 1 over the grid. Because we are interpolating, we get a very patchy color field.
- We'll blur them together using gradients. The way that texture lookup works for Perlin noise, is that your fragment shader is going to run, you'll get color at some point of the grid, and for this point on the grid, you'll build a vector for each corner and what you'll do is take the dot product between it and the existing vector. We'll use this to interpolate the colors of the corners. This will give us a smooth color filed that tries to vary each color along these random vectors. That's it.



The finer resolution of your grid, the higher frequency of color change you'll get.

You don't just have to think of these as colors, you can think of these values as heights.

7 Animation and Kinematics

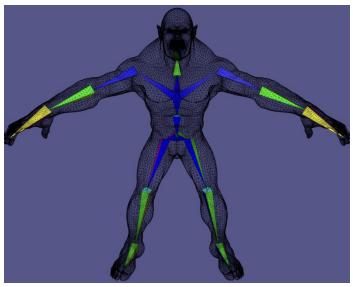
Key-frame animation comes out of traditional animation where an artist would draw a few key important poses of the character, and the extra frames will have to be filled in.

With computer animation, we have a different problem: how are we going to specify the motion of an object? You've seen that a lot of our objects we deal with is stored as meshes. Meshes are complicated in terms of the amount of data they have. Even a very simple mesh can have thousands of vertices in it.

One way we could specify an animation is an artist could move every vertex. That is obviously going to take forever, and it is unnecessary if you think about it. The degrees of freedom, the number of variables you have here is too big to be useful for most animation.

All animation systems try to reduce this number of variables somehow. Skinning is a particular type of reduction. In particular, in skinning, we're going to exploit the fact that most of the things we animate are characters, human or humanoid. They have limbs, these limbs are made up of bones and joints. Rather than move all the vertices, we're going to use the concept of bones, illustrated by pyramids. What the animator or the computer will care about moving is the bones, and we're going to establish a very simple mapping between the position of the bones and the position of the vertices.

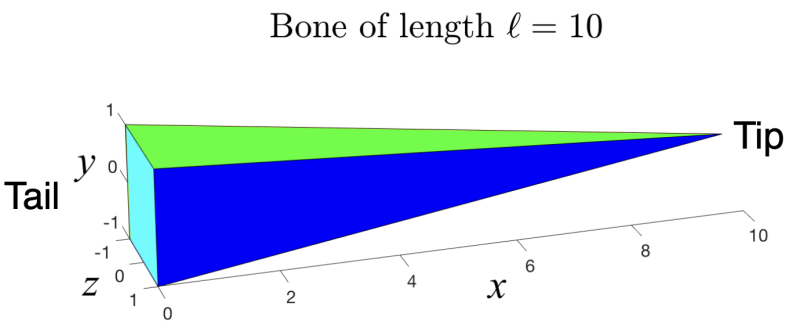
In this way, we can reduce a mesh of arbitrary complexity to some small set of bones.



It is the job of the one who rigs to place all of these bones in the right place.

7.1 Bones

This is a computer graphics bone. It is a pyramidal shape.



THE TIP IS ONLY FOR DISPLAY AND IT DOESN'T MATTER.

We'll define an isolated bone inside the bone space. Whenever we talk about transformation, we're going to talk about transforming between spaces.

We define our bone as having a tip, ALWAYS at $(x, 0, 0)$, in the x direction in some length (image: length = 10), and the bone base is in the zy plane. The length of the edges in the base are length **2** in the bone space and the center of the base is at the origin. This is very important. When we do animation, we'll move this bone around, so when we start applying transforms to this bone without translating anywhere, you want to cleverly place the origin, as in the image above.

The length of the bone isn't very important for now. The length doesn't matter as you'll have to transform the bone anyway by a transformation, and we are allowed to change the bone size when we do that.

7.2 Moving Bones

How do we move bones around? I want to bend my arm, rotate my forearm, and this is all encoded by moving the bone.

We need to be able to parameterize rotations of the bone. Why? We're going to operate under the assumption that when you're doing these animations, just like real bones, our virtual bones do not change length. If your bones don't change length, the position of you are based on the rotation of your bones. This assumption holds most of the time unless you're animating someone like Elastic girl from The Incredibles.

Rotation transforms are orthonormal. We'll encode rotation transforms as Euler angles:

$$(\theta_1, \theta_2, \theta_3)$$

We're using this one to encode translations as it is easy to code up and it's convenient.

- Twist around x-axis: θ_1
- Bending around z-axis: θ_2
- Bending around x-axis: θ_3

Gimbo lock is when you lose a degree of freedom. For example, this is why you spin when you move your mouse up and horizontally in an FPS.

Oh wait, you need four numbers to represent rotations perfectly. The fourth number can represent the whole field of rotations. That's quaternions. That avoids your parameterization from being stuck, and there is no singularity. In industrial implementations, quaternions are used but that's a pain to use.

7.3 Bone Translation

We have one bone. At a high level, we'll move bones around by rotating them. We're not translating anything so rotations will always rotate around the center of the base.

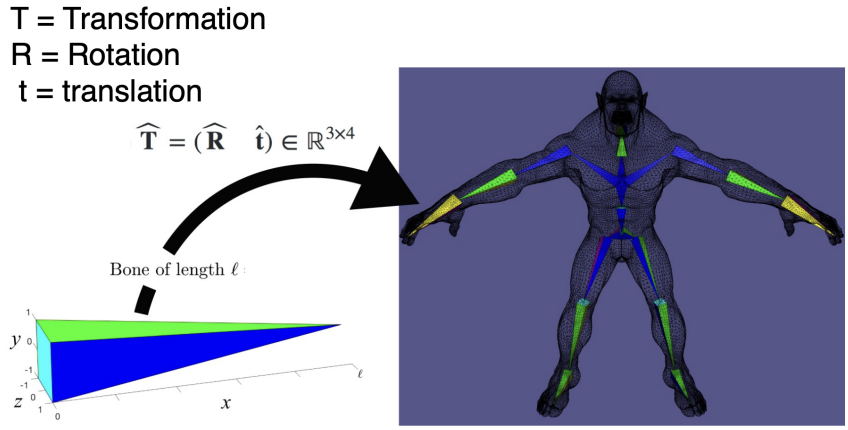
But first, we'll have to place the bone into our character. The way we do this is by defining a transformation $\widehat{T}$ that does exactly this. It moves the bone from the bone space into the rest pose of the character (input, T-pose).

The one who rigs specifies a rotation and translation matrix that transforms the bone in bone space to the character: $\widehat{R}$ and $\widehat{t}$ (this is matrix concatenation, not multiplication).

$$\widehat{T} = \begin{bmatrix} \widehat{R} & \widehat{t} \end{bmatrix} \in \mathbb{R}^{3\times 4}$$

Why are we using a 3×4 matrix? To save space. For all homogenous matrices, the

fourth row is always $[0, \ldots 0, 0, 1]$, and you'll know that the 4^{th} position you have to store is a 1 so we're not going to do that computation.



7.4 Posing a Bone

Once we start moving the bones around, we're talking about transforming the bones from the rest space to some position in the world. This is the final change here. There's going to be two important things to talk about:

- How do we move these bones around nicely, so they stay connected?
- Given some bone poses, how do you move the mesh?

7.5 Forward Kinematics

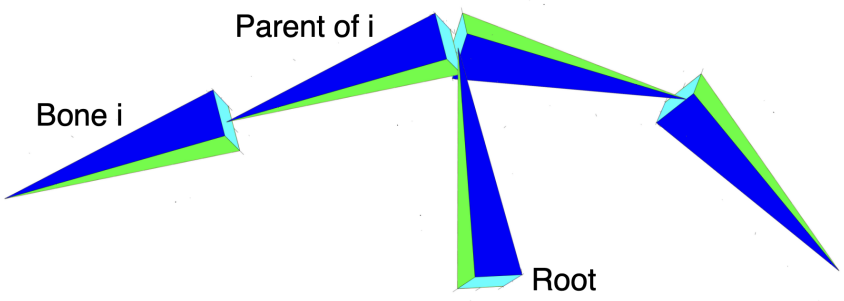
Kinematics – study of motion without consideration of what causes that motion.

Forward Kinematics – Generate motion by setting all the bone positions by hand.

To do this we need a more rigorous understanding of how our bone motions are represented.

Once we understand FK, we'll talk about IK, the idea where you only need to see the position of one bone and given the position of that bone, we can automatically compute the position of the rest of them (the motion of the arm and shoulder will automatically be computed, smoothly).

7.6 Bone Structure



- Think of a skeleton as a tree.
- The root bone is typically chosen at the center of mass of the character.
- All of the other bones are some trees emitted from the root bone.
- We'll end up at some leaf bones (such as fingertips)

Conventions:

- Parent of *i* is the bone where the tip attaches to the base of bone *i*

Why impose this additional structure? One of the things that is really important when we move these bones around is that we do not want to go apart. We don't want some animator to move the arm of the character and it dislocates entirely from the character. It will be painful and weird.

If you just wanted to assign transformations to each bone, you can completely discombobulate your character without meaning to. So instead, let's describe transformations of bones incrementally from their parent. If you know where your upper arm is in space, you will describe the motion of your forearm relative to this position.

We will express everything hierarchically. We will express bone transformations incrementally from their parent by walking the tree.

Here's how transformations work:

$$T_i = T_p \Delta T_i$$
$$\Delta T_i = \begin{bmatrix} \Delta R_i & \vec{0} \\ \vec{0} & 1 \end{bmatrix}$$

Where ΔR_i is in the **rest** space not the bone space.

We have to build this incremental rotation such that everything is happening in the right place. When you want to do transformations around a particular point in space, we get this sandwich: translate it to the origin, rotate it around the origin, and translate it back. We have a nice place to rotate it.

7.7 Understanding The Details of Euler Angles

Back to the bone space.

Euler angles are when you use 3 separate rotations around different axes. Knowing these do not define an algorithm. We need to choose an order to apply them.

This is the order where we are going to apply our rotation.

R_{x_1}: The first rotation is a rotation around the x-axis. The only parameter is an angle around the x-axis. It is very easy to compute the correct rotation matrix. This can ONLY twist the bone.

R_z: The next is around the z-axis; we rotate.

R_{x_2}: We'll rotate around the transformed y-axis. Let's assume there is an axis attached to the bone. Rotating around the attached y-axis will actually rotate it around the bone space x-axis. This gives us access to the final piece of the rotation.

We call this the XZX rotation, the order we applied these rotations.

Rotating around the _-axis: poke a pencil through a paper where the pencil is your axis.

7.7.1 Bone to Rest Space

T_p is based on the transformations that occurred to the parent bones THAT OCCURRED BY THE ANIMATOR.

$$T_i = T_p \Delta T_i$$
$$\Delta T_i = \begin{bmatrix} \Delta R_i & \vec{0} \\ \vec{0} & 1 \end{bmatrix}$$

In bone space, how do I get into or out of the rest space? We need

$$\Delta \mathbf{T}_i = \begin{pmatrix} \hat{\mathbf{T}}_i \\ 0\ 0\ 0\ 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_x(\theta_{i3}) & \mathbf{0} \\ 0\ 0\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_z(\theta_{i2}) & \mathbf{0} \\ 0\ 0\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_x(\theta_{i1}) & \mathbf{0} \\ 0\ 0\ 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{\mathbf{T}}_i \\ 0\ 0\ 0\ 1 \end{pmatrix}^{-1}$$

$\widehat{T}_i$ is transformations applied to get the bone in the T-pose (rest pose).

So, you do this incrementally for all your bones.

Because we compute positions incrementally, we don't need to store the position of the bone in storage.

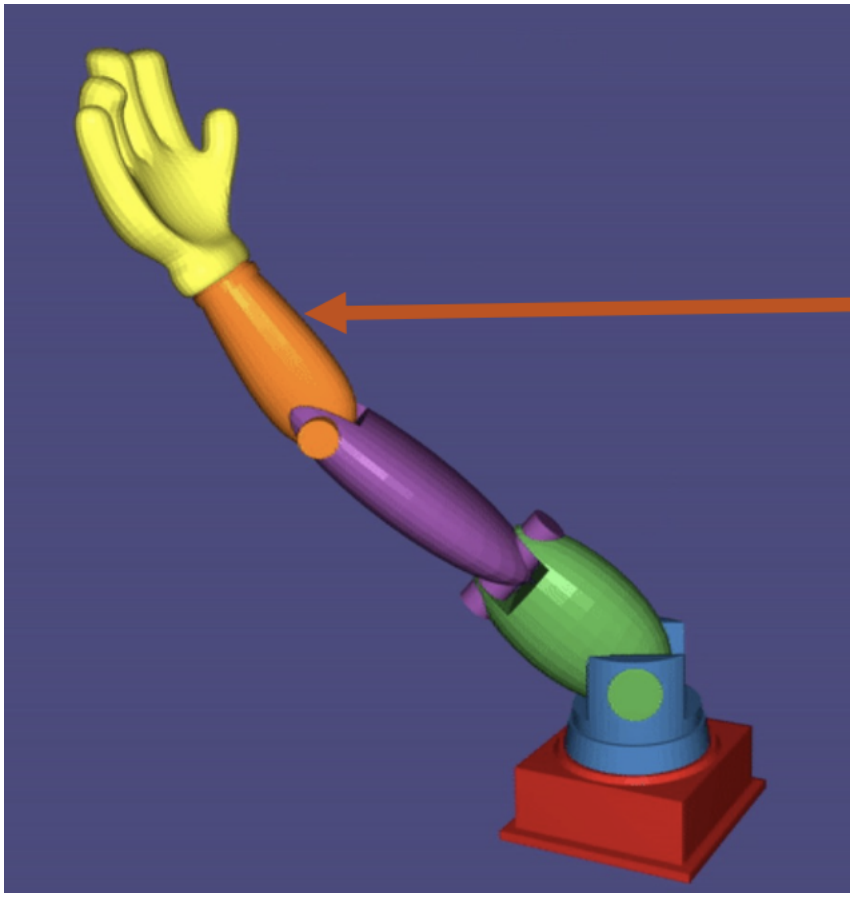
How is $\widehat{T}_i$ calculated? You have a translation for your root bone, and your animator will give you a transformation that tells you how to move the bone from the rest pose to the world space. If you want a transformation that moves from the bone space all the way to the world space, what do we need to do? For Root, we take the given animated transformation (takes from rest space to world space).

7.8 Rigid Skinning

Attach a vertex to a single bone. You have a rigid transformation. One thing you could do, is take the *j*th vertex of my mesh, find the closest bone, and move the vertices by taking the transformations of the closest bone and applying that transformation.

$$\vec{v}_j = T_i \vec{v}_j$$

This is the simplest version of skinning. One of the examples I'll animate is the robot arm:



Solution?

7.9 Linear Blend Skinning

Attach each vertex to some number of bones and average bone transformations together. We call it linear blend skinning as for each vertex in the rest character space, you'll pick some number of bones and for those bones, you'll prescribe a weight between 0 and 1. If you ant to move a vertex, you will visit all of the bones, look up the weight which describes how bone *i* impacts vertex *j*, sum all up and apply the averaged transformation to the vertex to get the new position. How do we get the weights? You set it up.

$$\vec{v}_j = \sum_{i=1}^{n\text{Bones}} w_{ij} T_i \vec{v}_j$$

Each vertex has a weight for every bone. This scalar weight is indexed over bones and vertices, so this results in lots of data to store. That is intractable if you have a huge number of bones, so most systems limit how many bones can be attached to a vertex to 3 or 4, and that's what you store. For most of our examples, the weights are blended locally, so most of the weight is 0.

When a character is in a different pose, T_i would be different.

7.10 Specifying Keyframes

We will use keyframing. We read in a set of poses and the computer is going to interpolate these poses in time. Each keyframe is a pose at a particular time.

All our poses are specified by Euler angles.

$$\theta = \begin{bmatrix} \theta_{11} \\ \theta_{12} \\ \theta_{13} \\ \vdots \\ \theta_{n1} \\ \theta_{n2} \\ \theta_{n3} \end{bmatrix}$$

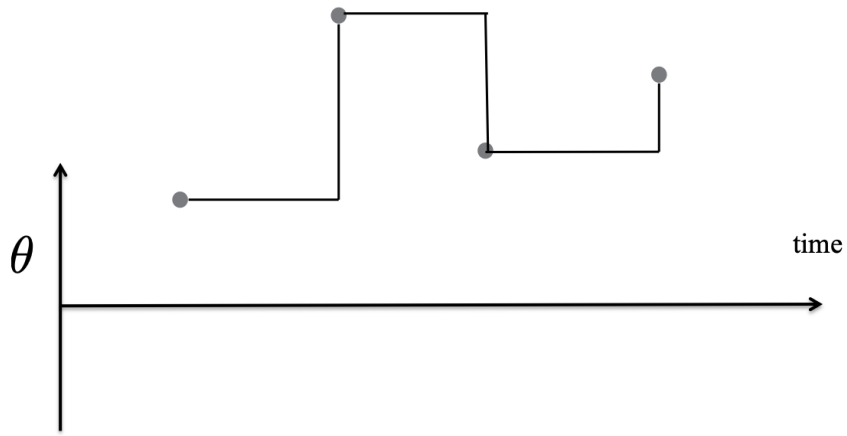
So how do we build a 1D curve in pose space? You'll never have to render it; it's a HIGH DIMENSIONAL space where every axis is a θ. Every keyframe is a point and you'll compute curves in this pose space between our keyframes.

But overall, this will result in a 1D function per angle.

7.11 Interpolating Keyframes

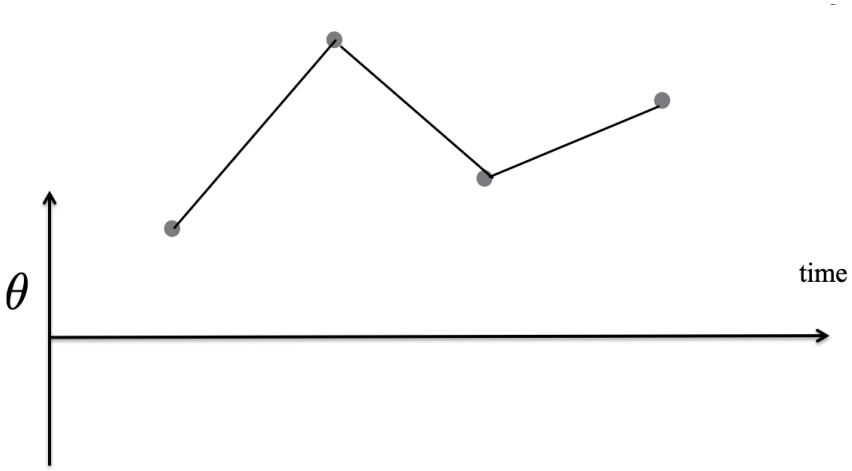
$\theta = \vec{c}(t)$ is a curve in the pose space. How do we construct such a curve from keyframes?

7.11.1 Hold



Piecewise constant: the function isn't going to change for some chunk of the function.

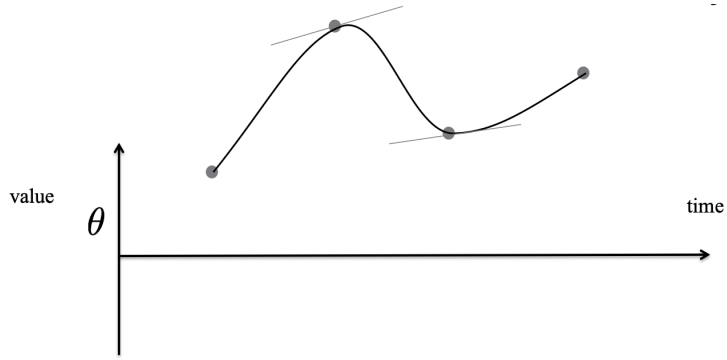
7.11.2 Linear



Every local chunk of the curve will be a line. Beware, velocity is piecewise constant, and this looks odd.

7.11.3 Spline

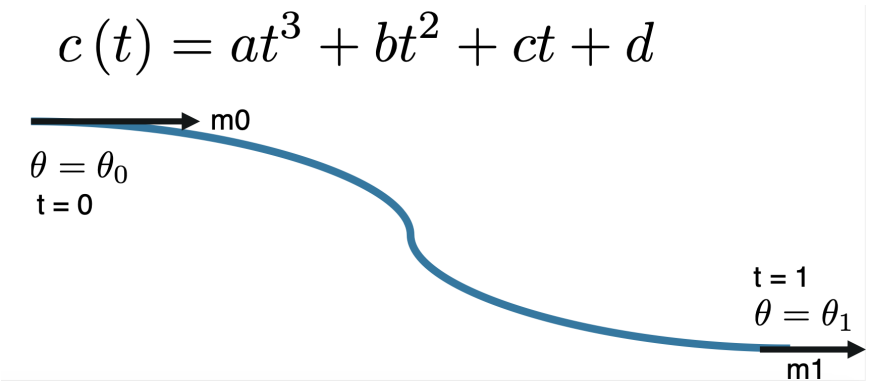
Piecewise cubic curves.



Given some number of keyframes, how do I compute a cubic curve that goes through.

It's much better than you think.

$$c(t) = at^3 + bt^2 + ct + d$$



a, b, c, d can be anything (vectors) as long as vector addition and scalar multiplication is defined.

We'll use a technique called the Catmull-Rom spline. It relies on knowing 4 pieces of information:

- Keyframe at start of curve
- Keyframe at end of curve

- Tangent vectors of the curves: some notion of how θ is changing at the start and the end of the curve (counts as two pieces of information), which you can probably adjust

You get a linear system.

For any time interval starting at $t = 0$, we can translate everything to time = 0 in case you're starting keyframe isn't exactly at 0. It is common to normalize the time interval to simplify things.

The tangent of any curve is just the derivative with respect to its parameter.

Note the following:

$$\begin{aligned} c(0) &= d \\ c(1) &= a + b + c + d \\ \frac{dc}{dt}(1) &= 3a + 2b + c \\ \frac{dc}{dt}(0) &= c \end{aligned}$$

We have four pieces of information: build a small matrix and solve this system.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} \vec{a}^T \\ \vec{b}^T \\ \vec{c}^T \\ \vec{d}^T \end{bmatrix} = \begin{bmatrix} \vec{\theta}_0^T \\ \vec{\theta}_1^T \\ \vec{m}_0^T \\ \vec{m}_1^T \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \vec{\theta}_0^T \\ \vec{\theta}_1^T \\ \vec{m}_0^T \\ \vec{m}_1^T \end{bmatrix} = \begin{bmatrix} \vec{a}^T \\ \vec{b}^T \\ \vec{c}^T \\ \vec{d}^T \end{bmatrix}$$

$$\mathbf{a} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{\theta}_0^T \\ \vec{\theta}_1^T \\ \vec{m}_0^T \\ \vec{m}_1^T \end{bmatrix} = \left(0 \cdot \vec{\theta}_0^T + 0 \cdot \vec{\theta}_1^T + 0 \cdot \mathbf{m}_0^T + \mathbf{m}_1^T \right)$$

So that's how you do high dimensional interpolation. If you do this, you will generate all of these smooth animations.

There is one slight catch, which is why this is called the Catmull-rom spline. **It does not take the tangents as inputs.** It computes the tangent in a finite-difference manner.

When you compute a Catmull-rom curve for two points for keyframe no. k:

$$\vec{m}_k = \frac{\vec{\theta}_{k+1} - \vec{\theta}_{k-1}}{t_{k+1} - t_{k-1}}$$

Estimate the derivative without shrinking that interval.

DO NOT SOLVE ANY SYSTEM, THIS IS A GIVEN

Catmull-Rom Spline

After solving and rearranging we end up with

$$\mathbf{c}(t) = (2t^3 - 3t^2 + 1)\theta_0 + (t^3 - 2t^2 + t)\mathbf{m}_0 + (-2t^3 + 3t^2)\theta_1 + (t^3 - t^2)\mathbf{m}_1$$

This is a general cubic spline (cubic in t).

Catmull-Rom choses the tangents using "Finite Differences"

8 Physics-Based Animation

Where all of the motion is generated by forces acting on that object. We're going to implement a small physics simulator and that is going to treat every object as

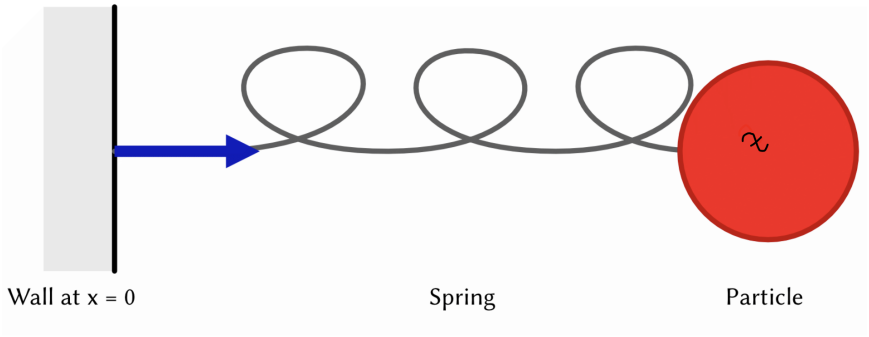
masses connected by strings, and by solving that problem you'll be able to generate animations.

8.1 Newton's Laws

Classical mechanics is based on Newton's 3 laws.

1. Objects are in rest or uniform motion unless acted on an external force
2. $F = ma$
3. For every action, there is an equal and opposite reaction

Everything we're going to talk about today will be concerned about the second law. We're going to solve $F = ma$ for a particular kind of physical system. Our physical system is made up of particles. Connected to each other by strings. This is just a simple one-dimensional example:



We can make some decisions about the particle:

- At rest if the center of the particle is directly at the wall

There are lots of ways to write forces, but in general, for graphics, we want to write simple forces that are easy to evaluate. For this, we can write a simple linear relation:

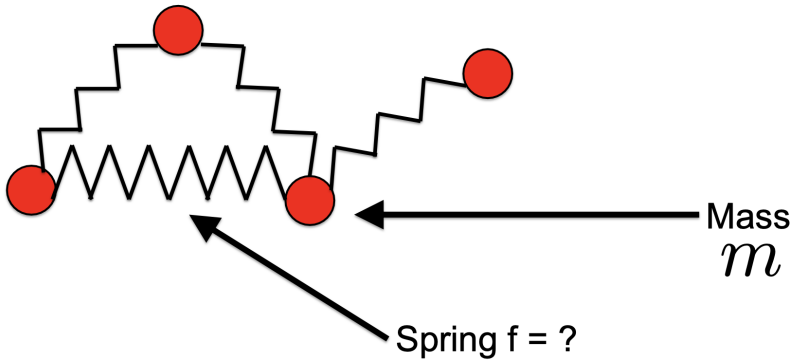
$$f = -kx$$

Where k is the spring constant which tells us how stiff the spring is, and x is the position of our particle. The further the particle is away from the wall, the more the spring

would want to pull the particle towards the wall.

We're going to need something more elaborate though. We want to lift this thing up into higher dimension. Instead of having a simple mass-spring, we'll have a mass-spring system in multiple dimensions.

8.2 The Mass-Spring System

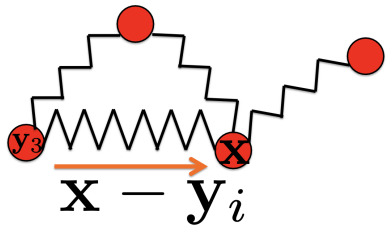


We'll connect each particle with strings, and we'll assign a scalar mass for each particle.

The important bit is we need to lift up this spring force into higher dimensions. Each force is proportional to a spring constant, and also how far the particle is from the rest state. The force also has a direction. In 1D, the direction is either the same or opposite of our distance from equilibrium state. The goal is we want to take all of these concepts: material parameters, distance from rest state, and we want to lift that whole thing up into an equation we can apply in higher dimensions.

We don't need complicated mechanical models, so we'll do this in the simplest way possible.

Focus on one spring.



Pick a particle x. What are the forces exerted between x and one of its neighbors, y_3? The formulas will generalize for any string.

$$\vec{f}_x = -k\left(||\vec{x} - \vec{y}_i|| - r_{ix}\right) \frac{\vec{x} - \vec{y}_i}{||\vec{x} - \vec{y}_i||}$$

Let's break this apart.

Our problem setup when we have a mass-spring setup is a bit different from the 1D problem.

In the 1D problem, the rest state of the particle is when the spring has 0 length. Where we have zero force is what we call the rest state.

If you're trying to simulate some deformable character and you have zero rest length on all your strings, your character will crumple up into a singularity. So that's not quite right. We'll have to modify this notion – introduce a nonzero rest length. Rather than the length of the string $\vec{x} - \vec{y}_i$, you want to have the spring go to the rest state of the system. This stops a painful collapse of a geometry.

8.2.1 Looking at the equation

$$\vec{f}_x = -k\left(||\vec{x} - \vec{y}_i|| - r_{ix}\right) \frac{\vec{x} - \vec{y}_i}{||\vec{x} - \vec{y}_i||}$$

Each parameter:

- r_{ix} is the rest length
- $-k\left(||\vec{x} - \vec{y}_i|| - r_{ix}\right)$ magnitude
- Other parts: direction

How stretched out a spring is? We're going to take the distance between the two particles. We'll introduce another constant, r_{ix}. This is the rest length of the spring; the value of the norm when you first load the object.

By definition, $\vec{x} - \vec{y}_i$ = rest length. So we guarantee that the force will be 0 in the rest configuration.

As we stretch or compress the spring, the norm will get larger or smaller than r_{ix} and we will get a nonzero force.

The other piece of the equation is that we need to know the direction of the force. Given two particles in space, we have no idea what direction to apply the force. Well, we know that springs only really apply force along the spring direction. You only want to pull along the spring.

So, our force will act along the normalized of $\vec{x} - \vec{y}_i$ (this vector points towards $\vec{x}$).

Why do we have the negative sign?

The magnitude of the force only depends on the length of the spring, and the direction depends on which particle you look at. If you want to get two particles back together, you want the force of $\vec{x}$ to take it towards its neighbor, $\vec{y}_3$, so you'll have to negate it to go in the right direction.

If you wanted to look at $\vec{y}_3$, I wouldn't need that negative sign. **The forces exerted on the two nodes are equal and opposite.**

8.2.2 For Each Particle

Sum up all the forces to get the equation **for each** particle. So, the force is:

$$m_x \vec{a}_x = \vec{F} = \sum_i \vec{f}_x(\vec{y}_i)$$

8.3 Solving The 2nd Order Differential Equation

Instead of talking about the system of equations like

$$m_1\vec{a}_1 = \sum_i \vec{f}_1(\vec{y}_i)$$
$$m_2\vec{a}_2 = \sum_i \vec{f}_2(\vec{y}_i)$$
$$\vdots$$

We could use this, where we stack up all the accelerations (good reason NOT to dot product – oh wow, a matrix of matrices):

$$\underbrace{\begin{bmatrix} \underbrace{m_1 \cdot I}_{3\times 3} & 0 & 0 & 0 \\ 0 & m_2 \cdot I & 0 & 0 \\ 0 & 0 & m_3 \cdot I & 0 \\ 0 & 0 & 0 & m_4 \cdot I \end{bmatrix}}_{\text{mass matrix}} \cdot \underbrace{\begin{bmatrix} \underbrace{\vec{a}_1}_{3\times 1} \\ \vec{a}_2 \\ \vec{a}_3 \\ \vec{a}_4 \end{bmatrix}}_{\vec{a}(t)} = \underbrace{\begin{bmatrix} \vec{f}_1 \\ \vec{f}_2 \\ \vec{f}_3 \\ \vec{f}_4 \end{bmatrix}}_{\vec{f}(\vec{y}(t))}$$

$$M\vec{a}(t) = \vec{f}(\vec{y}(t))$$

Acceleration is a differential quantity. They are an intermediate stop to what we really want – the positions of all the particles. If you want to generate an animation, you want to figure out where these particles are moving to.

This is called **time integration.** We have to integrate this thing twice to convert it back to positions. There is a whole bunch of ways to do this.

Acceleration and forces are time-variant functions. As you change the position of the mesh in time, the forces will change, and the instantaneous acceleration will change.

8.3.1 Forward-Euler time integration

A totally valid algorithm. The only problem with this method is that for objects that are elastic, that oscillate, it is 100% unstable. For any method, there will always be a range where your values will be unstable.

8.3.2 Time Integration

The first thing we'll do, is to immediately get rid of the acceleration and replace it of position.

$$M\frac{d^2\vec{y}(t)}{dt^2} = \vec{f}\left(\vec{y}^{t+1}\right)$$

Guess what, we have this (beware: y^t is NOT an exponent, but are the positions in time):

$$\frac{d^2\vec{y}(t)}{dt^2} \approx \frac{1}{\Delta t^2}\left(\vec{y}^{t+1} - 2\vec{y}^t + y^{t-1}\right)$$

You have $\vec{y}^t$ and $\vec{y}^{t-1}$, so you can solve $\vec{y}^{t+1}$. You also have to discretize $\vec{f}\left(\vec{y}(t)\right)$.

If you choose $\vec{y}(t)$ to be $\vec{y}^t$ (state of current time), you'll experience pain. Don't do that. You'll do something that seems a bit counterintuitive. We'll evaluate our forces at the next time-step. By looking ahead in the future, we can prevent snapping past the equilibrium state and exerting huge forces.

These are first-order accurate, meaning your solution will never blow up.

8.4 Implicit Time Integration

Break up the equation.

$$M\left(\frac{1}{\Delta t^2}\left(\vec{y}^{t+1} - 2\vec{y}^t + y^{t-1}\right)\right) = \vec{f}\left(\vec{y}^{t+1}\right)$$

$$\Rightarrow$$

$$M\vec{y}^{t+1} - M\left(2\vec{y}^t - \vec{y}^{t-1}\right) - \Delta t^2\vec{f}\left(\vec{y}^{t+1}\right) = \vec{0}$$

How do you find when some equation = 0?

Not with root finding, it's really hard to implement.

8.4.1 Implicit Integration as Optimization

If we can write a function $E(\vec{q})$ such that $\nabla_{\vec{q}} E\left(\vec{y}^{t+1}\right) = 0$

Then, rather solve

$$M\vec{y}^{t+1} - M\left(2\vec{y}^{t} - \vec{y}^{t-1}\right) - \Delta t^2 \vec{f}\left(\vec{y}^{t+1}\right) = \vec{0}$$

We can solve:

$$\vec{y}^{t+1} = \arg\min_{\vec{q}} E(\vec{q})$$

How are we going to build this scalar function? Let's take all terms that relate to the mass and find an energy that represents them and take the force term and handle that.

- Mass: $M\vec{y}^{t+1} - M\left(2\vec{y}^{t} - \vec{y}^{t-1}\right)$ to find $E_1\left(\vec{y}^{t+1}\right)$
- Force: $\Delta t^2 \vec{f}\left(\vec{y}^{t+1}\right)$ to find $E_2\left(\vec{y}^{t+1}\right)$

So, for absolutely no reason:

$$E_1\left(\vec{y}^{t+1}\right) = \frac{1}{2}\left(\vec{y}^{t+1}\right)^T M\vec{y}^{t+1} - \left(\vec{y}^{t+1}\right)^T M\vec{b}$$
$$\vec{b} = 2\vec{y}^{t} - \vec{y}^{t-1}$$

8.4.2 Potential Energy

How do we find E_2?

Where $\vec{q}$ is all the degrees of freedom you can move in your system

If $E_2(\vec{q})$ is a potential energy, then:

$$\nabla_{\vec{q}} E_2 = -\vec{f}(\vec{q})$$

And now, what is our potential energy for our particular case, this spring?

$$E_{ij} = \frac{k}{2}\left(\left|\left|\vec{y}_j - \vec{y}_i\right|\right| - r_{ij}\right)^2 \geq 0$$

If we want the energy of the entire system, we just sum up the energy of all the springs.

$$E_2 = \sum_{ij} E_{ij}$$

So now, we can solve:

$$\vec{y}^{t+1} = \arg\min_{\vec{q}} E_1(\vec{q}) + \Delta t E_2(q)$$

8.5 Local-Global Solvers for Mass-Spring Systems

WHILE Not done
    For Each Spring
        Local Optimization

    Global Optimization
END

The “for each” is executed in parallel for each spring.

These algorithms are stable and produce visually plausible results very quickly.

8.6 Rethinking Potential Energy

Do you see what is going on? You’re just converting a scalar into a vector so you can stuff it into the norm.

$$E_{ij} = \frac{k}{2}\left(\|\mathbf{y}_j - \mathbf{y_i}\| - r_{ij}\right)^2 \text{ is equivalent to}$$

$$E_{ij} = \arg\min_{\mathbf{d}_{ij}, |\mathbf{d}_{ij}| = r_{ij}} \frac{k}{2}\|\mathbf{y}_i - \mathbf{y_j} - \mathbf{d}_{ij}\|^2$$

Given $\mathbf{y}_i - \mathbf{y_j}$ we can quickly find $\mathbf{d}_{ij}$

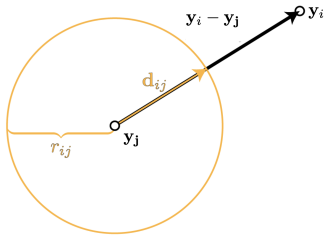
In order to make this work:

The hardest part of these optimizations is handling the potential energy term. It has so many terms which makes it not quite quadratic, and this is the source of all the difficulties.

What we will do is force this thing to be quadratic but force it to be quadratic in a gross and ugly way.

You could take this potential energy and introduce an extra vector $\vec{d}$. If we introduce this, we can rewrite E_{ij} as just the two norms squared. This just gets rid of the scalar.

If you are given the end point positions of the spring $\vec{y}_i$, $\vec{y}_j$, it's easy to find $\vec{d}_{ij}$ with the constraint $\left|\left|\vec{d}_{ij}\right|\right| = r_{ij}$. How do you find the direction?



Now we finally have a quadratic, so we can optimize.

8.6.1 What can we do with this?

TYPO: get rid of the arg in the "arg min d_{ij}"

We can expand a bit more …

$$E_{ij} = \arg\min_{\mathbf{d}_{ij}, |\mathbf{d}_{ij}|=r_{ij}} \frac{k}{2}\left(\|\mathbf{y}_i - \mathbf{y_j}\|^2 - (\mathbf{y}_i - \mathbf{y_j})^T \mathbf{d}_{ij} + \mathbf{d}_{ij}^T \mathbf{d}_{ij}\right)$$

Aside from the constraints, this is a nice quadratic energy

$$\mathbf{E}_1\left(\mathbf{y}^{t+1}\right) = \frac{1}{2}\left(\mathbf{y}^{t+1}\right)^T M\mathbf{y}^{t+1} - \left(\mathbf{y}^{t+1}\right)^T M\mathbf{b}$$

$$E_2 = \sum_{ij} \frac{k}{2}\left(\|\mathbf{y}_i - \mathbf{y_j}\|^2 - 2(\mathbf{y}_i - \mathbf{y}_j)^T \mathbf{d}_{ij} + \mathbf{d}_{ij}^T \mathbf{d}_{ij}\right)$$

Okay you should look at the slides because I can't copy this all down

The problem with *d* is that it is a constrained value

8.6.2 Block Coordinate Descent

Optimize one, then the other, then back and forth.

When E_1 and E_2 are convext and all your constraints are convex sets, your algorithm converges.

So now, an optimization algorithm exists that is well-suited to solve this difficult problem.

In practice, there are two steps. Each step is a minimization with respect to each individual variable.

8.7 The Global Step

Slides

But the summary is just:

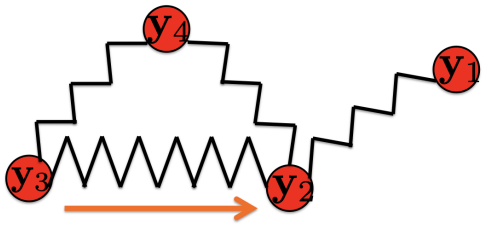
$$\nabla E_1 = M\vec{y} - M\vec{b}$$

And

$$E_2 =$$

This gets tricky.

Global Step



$$\mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \\ \mathbf{y}_4 \end{pmatrix}$$

$$\Delta\mathbf{y} = \underbrace{\begin{pmatrix} I & -I & 0 & 0 \\ 0 & I & -I & 0 \\ 0 & I & 0 & -I \\ 0 & -I & I & -I \end{pmatrix}}_{G} \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \\ \mathbf{y}_4 \end{pmatrix}$$

Each row is a spring

G has a typo here on the last row.

Adjacency matrix… sort of, but not at the same time. This gives us the edge vectors. And now, we can rewrite E_2 as:

$$E_2 = \frac{k}{2}\left(\vec{y}G^TG\vec{y} - 2\vec{y}^TG^T\vec{d} + \vec{d}^T\vec{d}\right)$$
$$\nabla E_2 = kG^TG\vec{y} - kG^T\vec{d}$$

So, the global step finds y so that:

Using this we can rewrite the second energy as

$$E_2 = \frac{k}{2}\left(\mathbf{y}G^TG\mathbf{y} - 2\mathbf{y}^TG^T\mathbf{d} + \mathbf{d}^T\mathbf{d}\right) \qquad \mathbf{d} = \begin{pmatrix} \mathbf{d}_{12} \\ \mathbf{d}_{23} \\ \mathbf{d}_{24} \\ \mathbf{d}_{34} \end{pmatrix}$$

So the gradient becomes

$$\nabla E_2 = kG^TG\mathbf{y} - k\mathbf{G}^T\mathbf{d}$$

And the total global step finds y so that

$$\nabla(E_1 + \Delta t^2 E_2) = (M + \Delta t^2 kG^TG)\mathbf{y} - (M\mathbf{b} - \Delta t^2 k\mathbf{G}^T\mathbf{d}) = 0$$

$\vec{d}$ is a stacked vector.

The purpose of G^T is to take in every edge.

8.8 How to read the lecture notes

Read it in reverse. For all terms you don't know, go back. Constructing G is the hardest part of the assignment. You should expect that the first time you run the simulation, something crazy is going to happen.

In G, which one gets the positive and which one gets the negative? It doesn't matter. For each row, one I needs to be positive and the other I needs to be negative. Each row can only have 2 Is.

9 Text to Image Generation

A task where the goal is to generate an image that corresponds to a given textual description.

9.1 Image Distribution

For each feature, for each image, there is a probability of whether that image fits that feature.

That is a model distribution.

How do we do this?

We collect many images (each $\vec{x}$), label them (each y).

The machine learning model takes in $\vec{x}$ and computes y.

Note that models must have their area sum up to 1, so:

$$p(\vec{x}) = \frac{y}{Z}$$

Where Z is the normalization term. However, this is incredibly hard to compute as we have to integrate over the entire image domain.

How do we fix this? We can take the log on both sides:

$$\log\left(p\left(\vec{x}\right)\right) = \log(y) - \log(Z)$$
$$\nabla_{\vec{x}} \log\left(p\left(\vec{x}\right)\right) = \nabla_{\vec{x}} \log(y) - \nabla_{\vec{x}} \log(z)$$
$$\nabla_{\vec{x}} \log\left(p\left(\vec{x}\right)\right) = \nabla_{\vec{x}} \log(y)$$

We want to maximize $\nabla_{\vec{x}} \log\left(p\left(\vec{x}\right)\right)$

9.2 How do we start generating?

We start with an image filled with noise.

On the image distribution, we iteratively take steps (gradient ascent) until we get an image with high probability for our label. This is how sampling or inference works, and this is a very slow process.

9.3 Where does the Diffusion Model name come from?

This type of modeling is called a score matching problem.

9.4 Forward Diffusion Process

Given a cat image $\vec{x}^0$:

Sample a random noise image.

Add noise by blending.

We have full control of this process. After T steps, we end up with a noisy image $\vec{x}^T$.

Given any $\vec{x}^0$, we can fully control this process; ε is the random noise image.

This seems useless. Why do we do that? Because what we are concerned about is reversing this.

9.5 Reverse Diffusion Process

Given a noisy image, how do we generate a clean image?

The idea is, we can create a machine learning model:

Given a noisy image $\vec{x}^T$, our model gives us $\varepsilon_\theta\left(\vec{x}^T,\ T\right)$

And then we can:

$$\vec{x}^T - \varepsilon_\theta\left(\vec{x}^T,\ T\right) = \vec{x}^{T-1}$$

By iteratively doing this process, we can deblur the image.

9.6 Model Fitting

How do we get that model in the first place?

Recall in the forward process, we know everything about giving noise to an existing image in our training set.

For any intermediate step, from 0 to t, we can draw some t steps:

I don't know what this is, but record it down: This is a loss function

$$\mathbb{E}_{t,\vec{x}^0,\varepsilon_t}\left|\left|\varepsilon_\theta\left(\vec{x}^t,\ t\right) - \varepsilon_t\right|\right|^2$$

ε is the noise that is added, and we want to figure out the noise that was just added to that iteration of the image without knowing the noise.

9.7 Connecting with the two definitions

Score matching and diffusion are initially two different ways to solve the problem. We want to fit:

$$\nabla \log\left(p\left(\vec{x}\right)\right)$$

With

$$s_\theta(\vec{x})$$

However, $p(\vec{x})$ is unknown. One way to fix that problem: instead of considering $\vec{x}$, we instead add noise to the model:

$$x \rightarrow q_\sigma(\tilde{x}|x) \rightarrow \tilde{x}$$

We can instead train this loss function for the score matching model:

$$\mathbb{E}_{\tilde{x},\vec{x}} \; ||s_\theta(\tilde{x}) - \nabla \log(q_\sigma(\tilde{x}|x))||^2$$

The high-level idea, we add noise, and we try to learn the denoising step.

9.8 Conditional Vs. Unconditional

Instead of doing a simple machine learning model, we could also give a prompt.

Conditional models have more control for users and are empirically better.

To add conditions:

$$\varepsilon_\theta(\vec{x}^t, t, y)$$

9.9 Cross Attention

Q: query $\vec{x}^t$

K: key y

V: value y

The formula to compute that:

$$softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$$

The similarity score is QK^T, soft max is the normalizer. Condition is kept in places where condition and image are similar.

9.10 Advanced Diffusion Model-Based Editing Tools

It's not quite easy to generate something very specific. Say I want to modify an existing image. I can add a text prompt.

9.10.1 Control Net

Retains edge information and nothing else. Add a new adjective, and it generates a new image that is very similar to our existing image.

9.10.2 Identity

Have a very specific dog? Use "A [V] dog" and you'll tell the image generator that you want this very specific dog.

9.11 What can these models do?

Quick generation of complex, high-quality and artist images; good for creative exploration

Integrating specific styles

By adding artist names, you can generate something in the same style (also, too bad you can't do this anymore).

9.12 What can't these models do?

Composition. Models tend to lose accuracy.

These models also struggle in generating multiple objects or coloring multiple objects.

Being reasonably unbiased.

9.13 Where does Bias come from?

When training data is collected and annotated, human bias can be generated here. This is where most common human biases come from.

When the model is trained, we risk overfitting, underfitting, default effect, and anchoring bias. Beware of the model you choose.

Once the model is out, we can rank the results

And once you post it, you do not want models to train on their output.

9.14 How to do better?

Bias can never be fully removed.

Something that is considered okay now can end up biased later on.

How do we address bias? See the assignment.

9.15 Generative Models and Artists

Do not take anything here as fact.