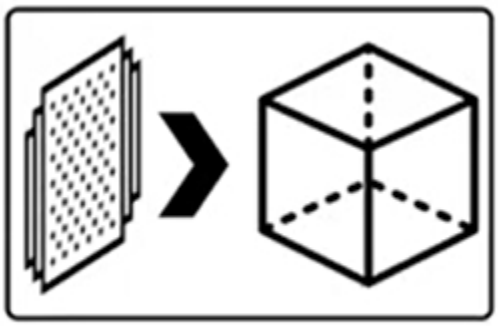
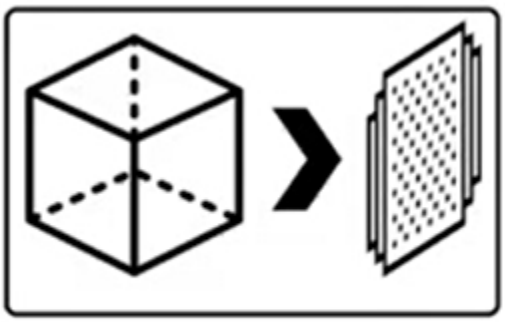
G1 - What is graphics?

So many applications like video games and animation, digital maps, fonts and even the entirety of the OS.

“Computer graphics study how to use computers to create and interact with visuals.”
visuals = “big data”, efficient computational representations
interact = speed, efficiency in algorithms
create = control, ease of use/creation



vision



graphics

In vision, take images from real world and analyze/do work on them.
In graphics, take existing information and create images.

Main subfields of graphics

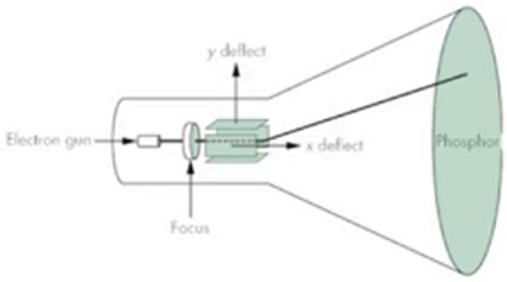
Modeling
Computational representations of geometry to create 3D models
Rendering
Algorithms to generate images from 3D models
Animation
Representing and editing motion with 3D models

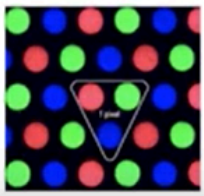
G2 - Image Revisited
Camera and projection (lecture 13-14) is important.

“Raster” or “bitmap” elements are a 2D grid of pixels, each with color value from a color space.
“Framebuffer” is an area of memory used to store an image/raster. It usually has a resolution. It also usually has a “depth” (pixel value type like 8-bit grayscale, 24-bit RGB).

In the past we had CRT displays

- Electron beam “steered” using voltage
- Hits phosphor elements line-by-line (i.e. in a “raster scan”)
- “Refresh rate” is limited by hardware





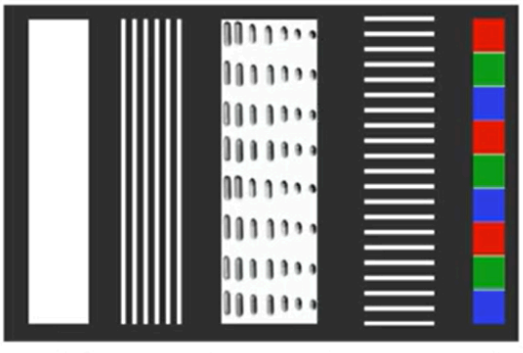
Flicker fusion rate: frequency at which flickering light appears steady to a human observer.

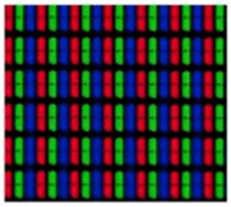
- High ambient light and large FOV: 80 Hz
- Low ambient light: 20-30 Hz

This is why film is at 24 fps and tv is at 30, while computers are usually at 60. They used interlacing.

Now we have LCD displays

Liquid crystal displays (“LCD” or “LED LCD”)





No voltage → no change in polarization → light blocked

Voltage → polarization rotated → light passes

We can control this LCD display by changing voltage of liquid crystals.

What about OLEDs?
Compared to LCDs, we don't have a backlight, each element is emitting its own light individually. Improves image contrast.

G8 - Geometric modeling

Modeling

Computational representations of geometry to create 3D models

Geometric modeling - Computational description of geometry.

We need data structures to represent geometry and algorithms to process representation. We need to model before we render.

Some models can be quite complex like boeing airplanes and forests.

Two types of geometry representation

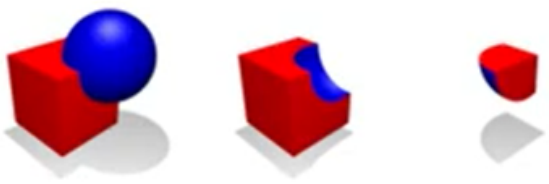
Implicit: Define equations that points on surface must satisfy

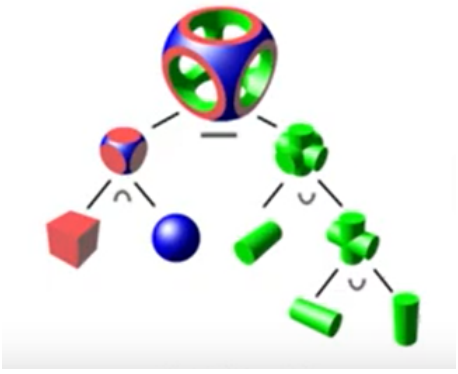
- Algebraic surfaces and distance functions
- Constructive solid geometry
- Level sets

We can have an implicit equation like

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

This defines "signed distance function" where we can get the distances from points inside and outside the sphere from examining the values of f.
Another example is constructive solid geometry where we combine simple geometries using boolean operations





Explicit: Define coordinates of points on surface

- Bezier curves and patches
- Point clouds
- Polygon meshes

One example is point clouds. We use a parametric equation to define a sphere (use spherical coordinates)

- How about $(\cos\theta \sin\phi, \sin\theta \sin\phi, \cos\phi)$ for $0 \leq \theta < 2\pi, 0 \leq \phi \leq \pi$?

This is what a point cloud is, just a list of 3D points. Getting more points on the surface is easy, but finding "signed distance" for points inside/outside of sphere is hard.

What about polygon meshes?

Define surface as a list of polygon vertex coordinates and the connections between vertices from polygons.

PROS AND CONS

- **Implicit representations**
  - \+ Inside/outside test is easy
  - \+ Efficient representation (e.g., just a few terms in an equation)
  - \+ "Infinite resolution" for simple shapes (i.e. we have algebraic formulas)
  - \- Hard to model complex geometries
  - \- Hard to efficiently sample points on surface
- **Explicit representations**
  - \+ Easier to model arbitrarily complex geometries
  - \+ Easier to sample points on surface
  - \- Inside/outside test is harder
  - \- Usually "finite resolution" (i.e. only explicitly define a discrete set of points)

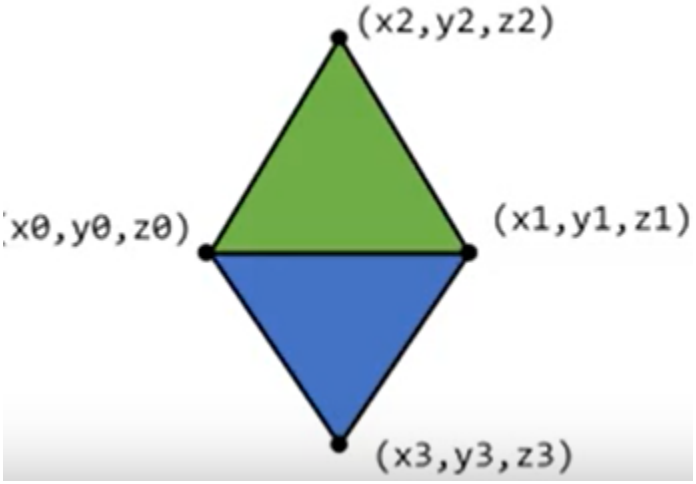
G9 - Polygon meshes

Polygon meshes describe surfaces by the position of the vertices and the specific edges.

RAster images represent an arbitrary image with 2D grid of elements

Polygon mesh represents arbitrary surfaces with a set of polygon elements. They are both simple data structures and make algorithms easier.

How do we encode these meshes? Triangle soup.



Store 3 vertices per triangle. This is simple but inefficient. We would store shared vertices between triangles, like P_0 and P_1.

To find shared edges/vertices, we need to iterate over entire data structure, so hard and inefficient.

We can do better!
Vertex list and face list

Vertex list

```
[(x0,y0,z0),
 (x1,y1,z1),
 (x2,y2,z2),
 (x3,y3,z3)]
```

Define all vertex coordinates in list (array)

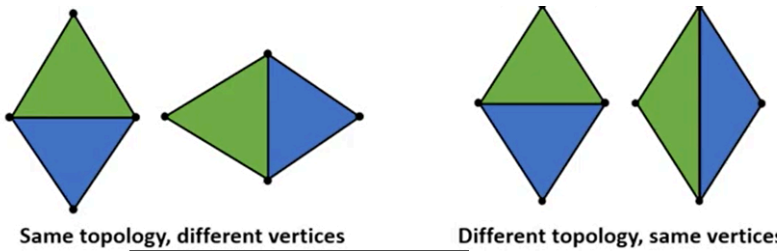
Face list

```
[(0,1,2),
 (0,3,1)]
```

Then add level of indirection using indices to vertex list to define each triangle face.

Now there are no duplicate vertex coordinates, but still hard to find triangles sharing vertices and edges.

Aside topology vs geometry.
Topology: global structure, doesn't care about actual coordinates (global positioning)
Geometry: care about local structure



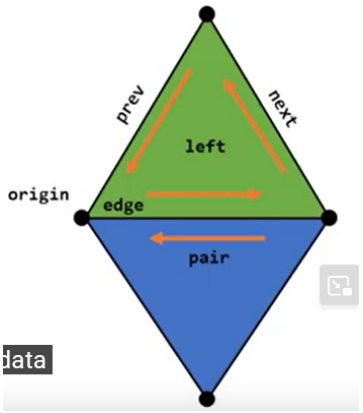
We can do even better by half-edge data structures. Use pair of “half-edges” each side of a shared edge to store info about adjacent faces/vertices/edges.
AKA “doubly connected edge list” (uses pointers like linked list).

Pseudo DS:

```
Halfedge {
  Halfedge* pair;
  Halfedge* next;
  Halfedge* prev;
  Face* left;
  Vertex* origin;
  Edge* edge;
}
```

```
Vertex { Halfedge* half; }
Edge   { Halfedge* half; }
Face   { Halfedge* half; }
```

What about our example?



Let’s go through some examples. What if we want to traverse all edges around a vertex?

Start on right most vertex

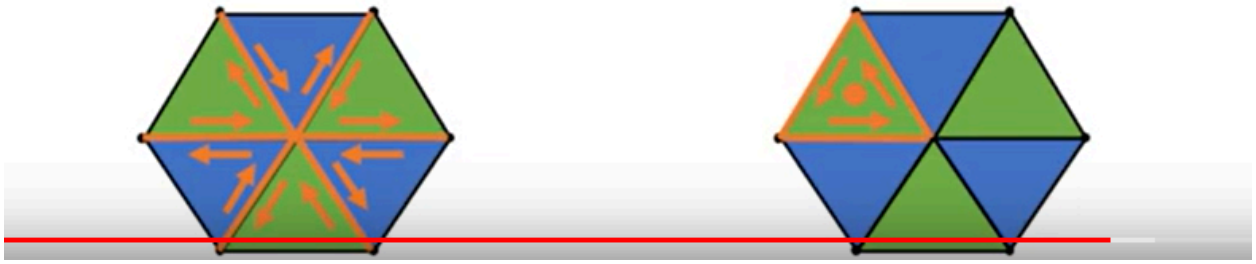
Traversing edges around a vertex

```
Halfedge* h = vertex.half;
do {
  // do something with h.edge
  h = h.pair.next;
} while (h != vertex.half)
```

Start on left face

Traversing edges around a triangle

```
Halfedge* h = face.half;
do {
  // do something with h.edge
  h = h.next;
} while (h != face.half)
```

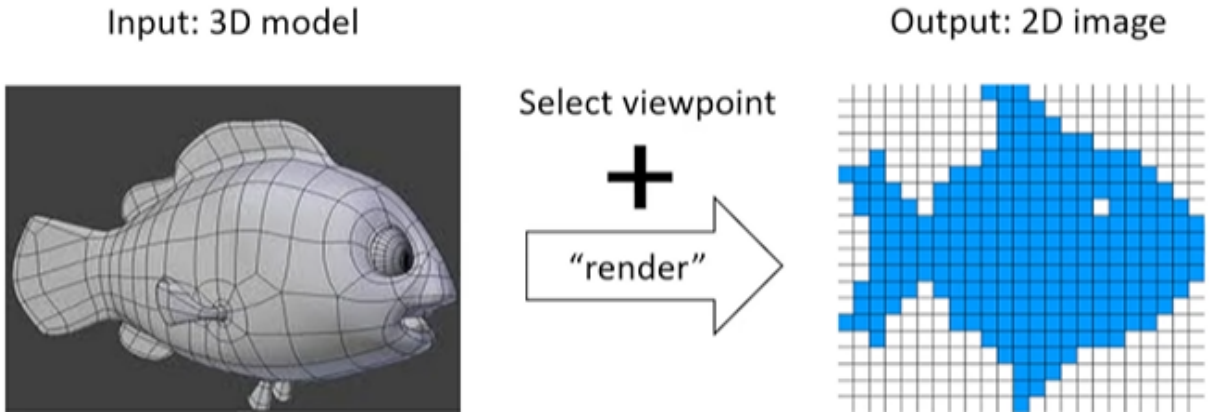


What about colors, surface normals, etc. Use same idea as “vertex list”: store attributes for each vertex in list, corresponding to indices.

```
positions = [(x0,y0,z0),(x1,y1,z1),(x2,y2,z2),(x3,y3,z3)]
colors =    [(r0,g0,b0),(r1,g1,b1),(r2,g2,b2),(r3,g3,b3)]
```

G3 - Rendering

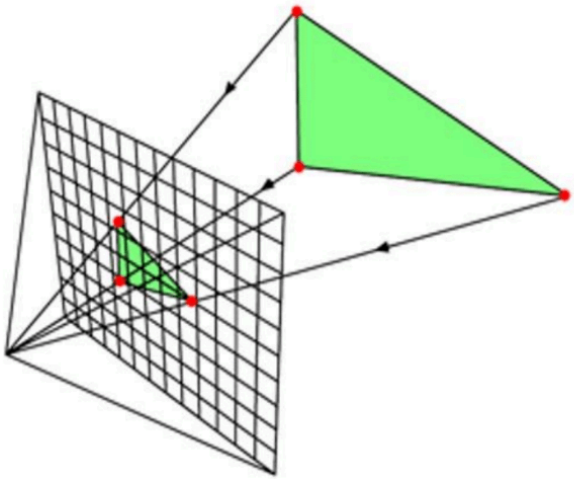
In G1, we saw rendering was defined as algorithms to generate images from 3D models.



"Rendering" = simulating the "physics of light". Because we want to approximate how objects look in the real world

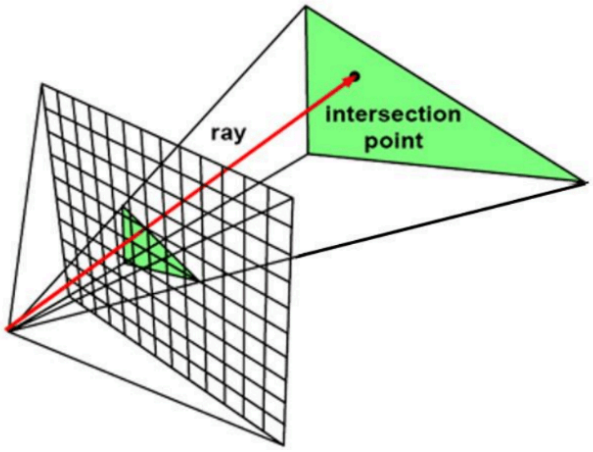
Rasterization

For each triangle: *"Which pixels are covered by this triangle?"*

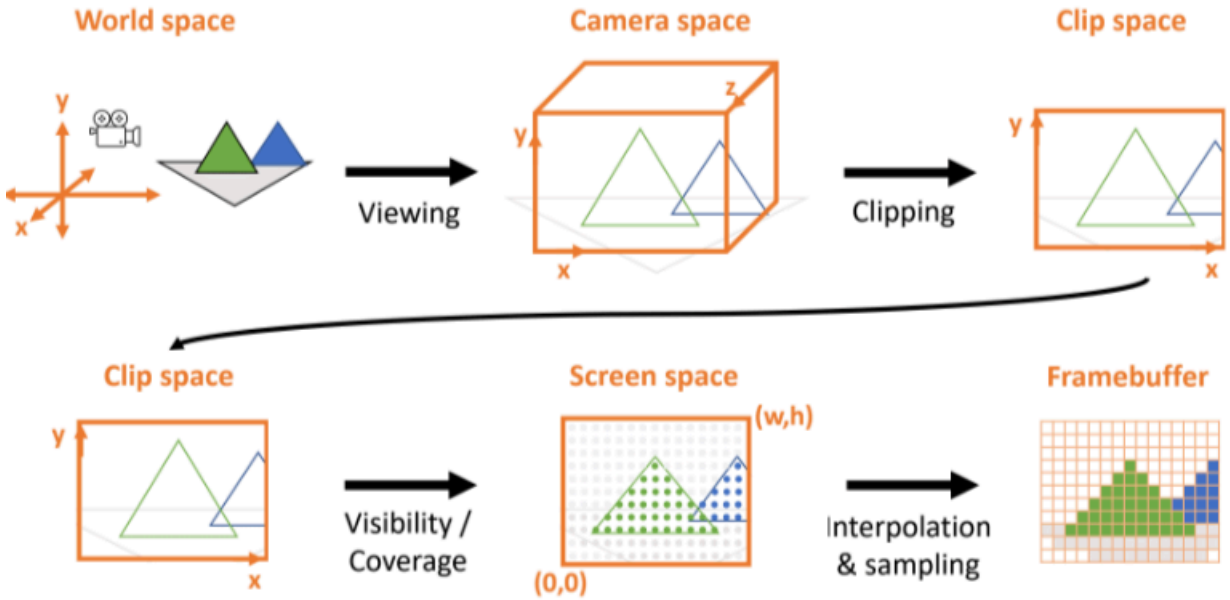


Ray tracing

For each ray (pixel): *“Which triangle is visible from this pixel?”*



G4 - Rasterization

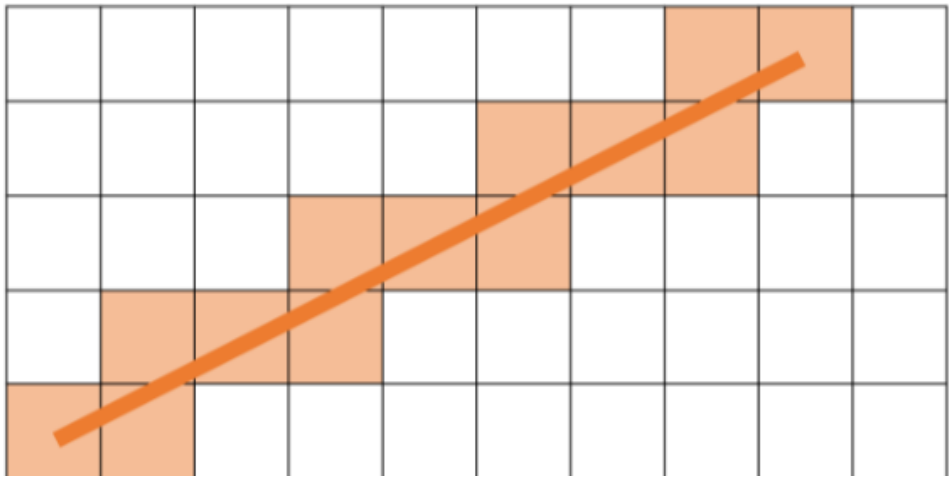


Note: GPU rasterization using OpenGL/Direct3D works this way, some “cosmetic” API differences

Viewing/projection - project model to camera
Clipping - Determine what is visible to camera
Visibility/coverage - What is visible to perspective

Interpolation and sampling - Presenting final image

How would we rasterize a line?

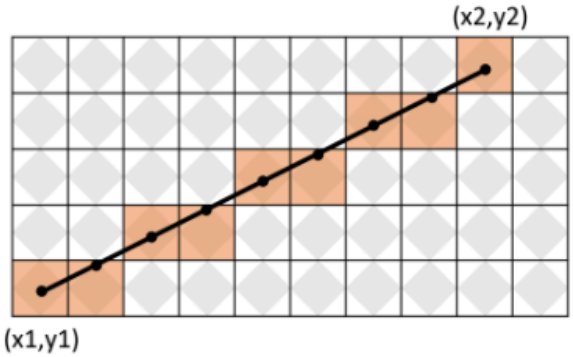


This is naive as check if the pixel hits the pixel (or diamond inside) is a O(n) or O(n^2)

This is why we have the DDA

Better: DDA (Digital Differential Analyzer)

- Think of line as: $m = \Delta y / \Delta x$
- Thus: $m = (y2 - y1) / (x2 - x1)$
- For now, assume: $0 \le m \le 1$ and $y1 < y2$ and $x1 < x2$
- *"Move from x1 to x2 by steps of 1 while increasing y by m = Δy"*
- Simple, extend by symmetry
- **Not** how lines rasterized in modern hardware & software (see Bresenham's algorithm in Chapter 8.9 of textbook)



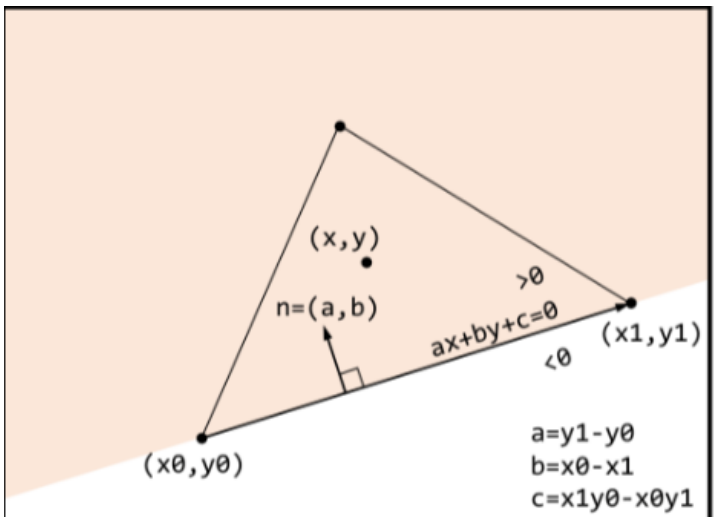

```
y = y1;
for (x = x1; x <= x2; ++x) {
  y += m;
  setPixel(x, round(y), color);
}
```

Rastering a triangle:

We want triangles as we can decompose any polygon into triangles, they are convex, flat in 3D, simpler rasterization and interpolation algo's → faster

Q: *"is fragment in triangle?" or "is fragment covered by triangle?"*
Also known as: "inside-outside testing"

Let's think about a triangle as 3 lines, 3 bounding half planes. P inside triangle if inside all 3 half planes.



- Line equation of line v0 (x0,y0) to v1 (x1,y1)
- Normal n points to "left of line"
- Convention: inside is on the left for counter-clockwise polygon
- **Algorithm**: *for each point* (x,y) *check whether inside all three lines forming triangle*

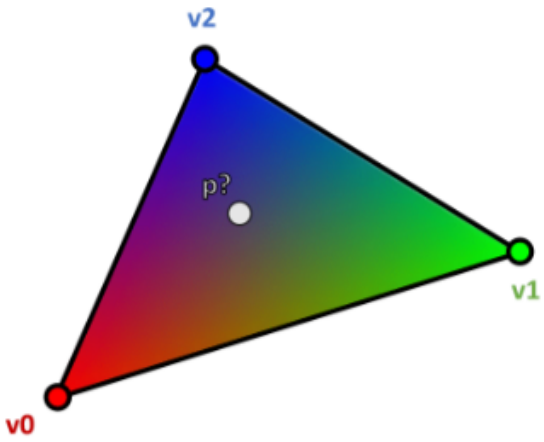
Handling edge cases: edges touching sample point: line equation = 0. So we need to adopt convention to create only one fragment.

- Common convention: "**top-left rule**". Point is inside triangle if point is on:
  - **Top edge**: edge exactly horizontal and above other edges
  - **Left edge**: not exactly horizontal and is on the left side of the triangle. Triangle can have one or two "left edges"

Now we don't do this for all triangles, computing inside test for all sample points can be quite wasteful like a small triangle.
Instead, compute triangle bound box first and do inside test only for points inside box.

```
xmin=ceil(min(v0x,v1x,v2x));
xmax=ceil(max(v0x,v1x,v2x));
ymin=ceil(min(v0y,v1y,v2y));
ymax=ceil(max(v0y,v1y,v2y));
```

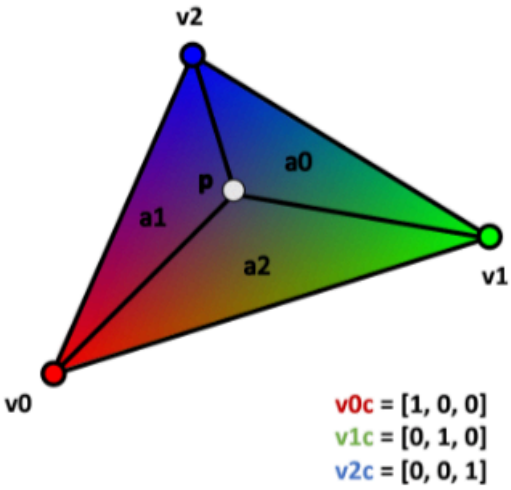
Barycentric interpolation



GIven colour values at triangle vertices, what is the color value at each point inside?

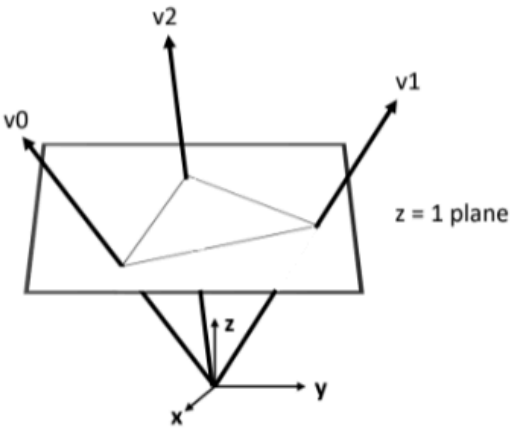
Lets split up the triangle into 3 smaller ones, and make a unit coordinate system that measures how big the splitting triangle is.

- Compute **area of three triangles** formed by p and vertices v0, v1, v2
- a0+a1+a2 = A (area of big triangle)
- Barycentric coordinates:
  - u = a0 / A
  - v = a1 / A
  - w = a2 / A
  - u + v + w = 1
- Then, to get color at p:
  pc = u*v0c + v*v1c + w*v2c



Okay but how do we actually do this?

- One way using a neat geometric trick:
  - Turn vertices to **homogeneous coordinates** by adding a 1; e.g., v0 = (x0,y0,1)
  - area(v0,v1,v2) ∝ volume(v0,v1,v2)
  - Triple product gives volume(v0,v1,v2) = (v0×v1)·v2
- Then, note that:
  - v0×v1 = (y0 - y1, x1 - x0, x0y1 - x1y0)
  - v0×v1 = (a, b, c) from line equation v0→v1
  - So for a point p (x,y) inside triangle, area is (v0×v1)·p = ax + by + c
  - Can reuse line equation from inside test!



How to determine what is in front?

How do we order triangle rasterization to get correct visibility (showing surfaces closest to us and blocking further surfaces)

We have a trivial method of “painters algorithm”: where we sort triangles from farthest to nearest and rasterize in that order. But we can have overlapping triangles that fail. Need something more advanced

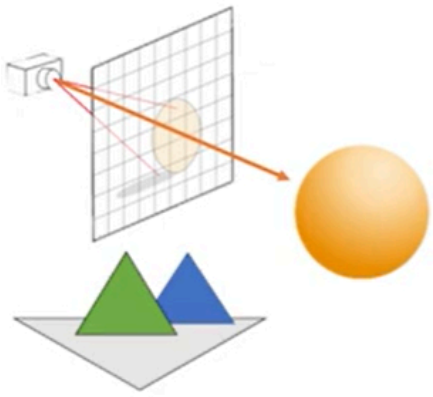
Z-Buffer algorithm

When we rasterize, we also store z value for that raster (store camera distance).
Then we initalize a framebuffer with z = -infinity for all pixels.
When creating a rasterized fragment, check z current value in framebuffer and only write color and z if z < current z (depth test)

G5 - Ray tracing

Instead of tracking where light reflects from an object to the camera, we track which possible light rays will reflect to the camera from the object.

We shoot a ray through each pixel that the camera faces to see if it will hit our target object.



Algorithm: *“For each ray, find closest object it hits”*

```
for pixel sample point p {
  r = ray(o,p);  // ray from camera origin o through p
  r.min_t = infinity;  r.obj = null;
  for object o {
    if (r.intersects(obj)) { // ray-object intersection
      if (r.distance(obj) < r.min_t)
        r.min_t = r.distance(obj);  r.obj = obj
    }
  }
}
```

min_t will be explained later.

How do we actually construct a ray?
Use parametric equations, r(t) = o + t*d.
o is camera origin, d is normalized vector of light direction.

For specific surfaces, the ray intersection test will be different. I.e. ray-triangle or ray-box. Let's focus on ray-sphere:
Implicit equation for sphere: all points x where f = 0
All points on ray: r(t)
Replace x with r and solve for t.

- Example for unit sphere: $f(\mathbf{x}) = |\mathbf{x}|^2 - 1$

$$\Rightarrow f\big(\mathbf{r}(t)\big) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$

$$\Rightarrow \overbrace{|\mathbf{d}|^2}^{a}\, t^2 + \overbrace{2(\mathbf{o}\cdot\mathbf{d})}^{b}\, t + \overbrace{|\mathbf{o}|^2 - 1}^{c} = 0$$

$$\Rightarrow t = -\mathbf{o}\cdot\mathbf{d} \pm \sqrt{(\mathbf{o}\cdot\mathbf{d})^2 - |\mathbf{o}|^2 + 1}$$

using quadratic equation formula $t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

This is a closed form solution for ray casting. How do we get to ray tracing?
Recursive ray casting.
Getting the camera ray to the surface intersection, we can do another ray cast from the surface to the light source.
Getting the camera ray to the ground, ray casting to the light source and seeing the surface intersects the ray cast from ground to light source → shadow.

This is computationally challenging as this is exponential with respect to the number of bounces, as each bounce will send out multiple ray casts. We can make the image brighter by allowing multiple bounces.
Sampling light paths is also challenging, as we don't always have a direct direction ray, like refraction.

Ray tracing is elegantly simple, but expensive. It's easy to implement but costly expensive.
Rasterization: loop over triangles first

- Never have to store entire scene, supports unbounded size scenes
- Store depth/frame buffer (need random access to regular structure of fixed size)

Ray casting: loop over pixels/rays first

- Never have to store closest depth for entire screen (just current ray)
- Natural order for rendering transparent surfaces (process surfaces in the order they are encountered along the ray either front-to-back or back-to-front)
- Must store entire scene (random access to irregular structure of variable size)

G6 - Spatial acceleration

Recall that ray tracing is expensive as it iterates over all objects. We can make it more efficient by

1) Call fewer intersection tests
2) Make intersection tests more efficient

Two key ideas

1) Hierarchal partitioning
2) Bound objects with simpler primitives

Spatial acceleration structures: data structures and algo's that above these.
We can distinguish partitioning with object vs. space partitioning, which is uniform vs. irregular data size.

Bounding volumes - bound object with simpler volume

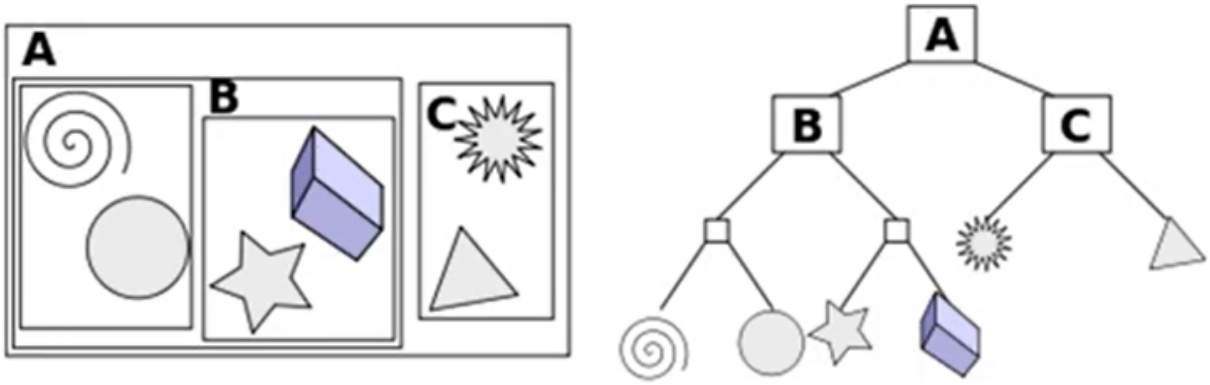
We can test ray intersection with bounding box, if no intersection, guaranteed no ray object intersection. Otherwise, do more expensive ray-object intersection test. This is worth when we have a simple shape as bounding volume, contained object is complex and good "tightness of fit".

a) **Spheres**: easy to intersect, but usually not very tight
b) **Axis-aligned bounding boxes (AABBs)**: easy to intersect, often tighter
c) **Oriented bounding boxes (OBBs)**: easy to intersect & tighter for arbitrary objects, but need to transform (orient the box)

Implementing a or b is quite easy. For (b), think of a cube.

We can compute a ray-slab intersection test by parameterizing ray, intersection with slab if interval intersection is not empty.

Bounding volume hierarchies (BVHs)



For multiple objects in a set, we can bound all of the objects by having on box fit all the objects, then make a box to split into subsets… and repeat.

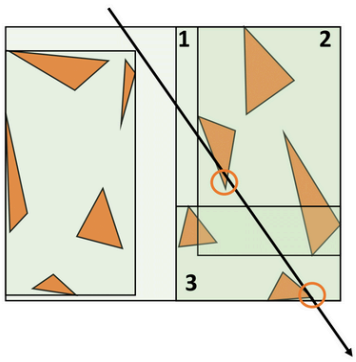
- Recursion: bound whole scene, split into two, recurse on two sides
- Stop when only few primitives in box

We can partition with SAH

- Theorem: given box B completely contained in box A, probability that ray traversing A intersects B is given by $\mathbf{SA}(B)/\mathbf{SA}(A)$
- Compute expected ray traversal cost for a subtree of hierarchy
- Then, greedily choose split that minimizes cost

$$\mathbf{Cost}(V \to \{L, R\}) = K_T + K_I \left(\frac{\mathbf{SA}(V_L)}{\mathbf{SA}(V)} N_L + \frac{\mathbf{SA}(V_R)}{\mathbf{SA}(V)} N_R \right)$$

BVH ray tracing example

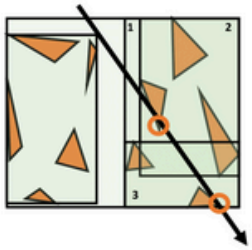


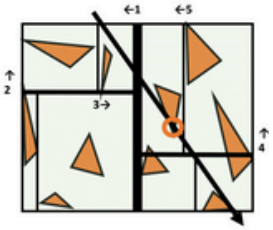
k-d trees

Instead of doing object partitioning, we do space partitioning.
Adaptive space subdivision, unlike uniform grids.

We traverse the ray line down the split hierarchy, terminate search on first hit.

- Object vs spatial partitioning
  - **Object partitioning** (e.g., BVHs): easy to bound number of nodes required, simpler to update if objects added or change position
  - **Spatial partitioning** (e.g., k-d trees): first intersection is closest intersection, but may intersect same object multiple times
- Adaptive vs non-adaptive (i.e. uniform)
  - **Adaptive**: expensive to construct, but better performance for non-uniformly distributed objects
  - **Non-adaptive**: cheap to construct, but worse performance unless objects uniformly distributed





G7 - Shading

What is a shading model? Recall that rendering = light transport simulation. This will tell us the path but what should the colour be?

Shading model: a computational model to colour the scene

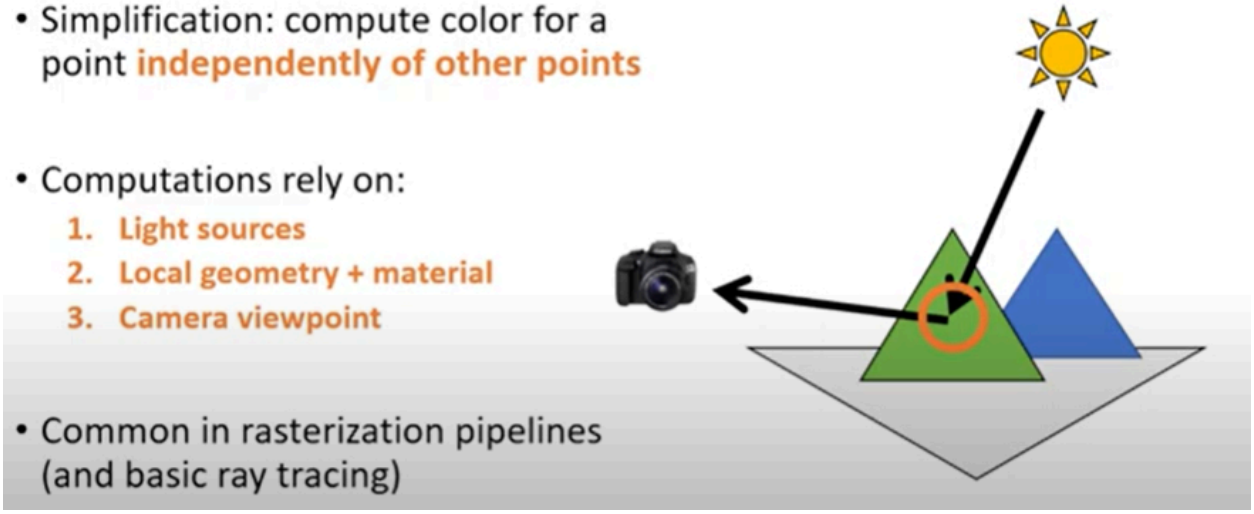
- Make use of an illumination model
  - Input: point p in the scene + scene (lights, objects, surface materials)
  - Output: color at p

There are two families of illumination models

- Local: illuminate surface point through direct light source
- Global: illuminate point with light transport from entire scene

Global illumination accounts for light transport paths between all surfaces like multi bounce ray tracing. Not commonly used with rasterization pipelines

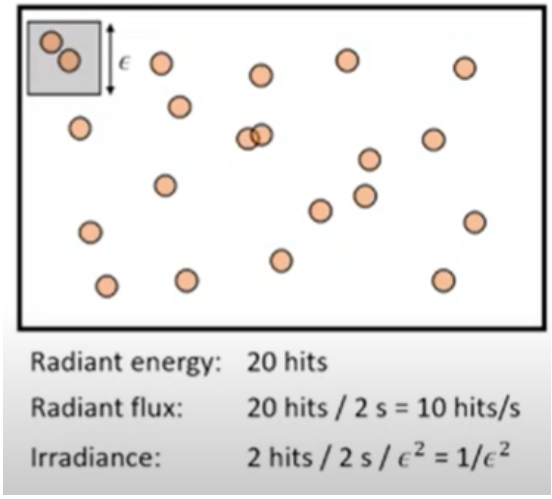
Local illumination computes color for a point independently of other points.



Quantifying light

Key idea: surface "brightness" is
~ energy of photons hitting surface
~ total # of photons hitting surface

Radiant energy: total # of photon hits
Radiant flux: hits per second
Irradiance: hits per second per unit area



Now, let's quantify these radiometric values!

- We counted photon hits so far; how much energy in a photon?
- Planck-Einstein relation:

Planck's constant, speed of light

Energy in photon $Q = \frac{hc}{\lambda}$ wavelength (color)

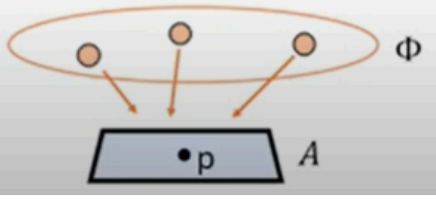
$h \approx 6.626 \times 10^{-34}$ [J · s]
$c \approx 3.00 \times 10^{8}$ [m/s]
$\lambda \approx 380 \text{ to } 750 \times 10^{-9}$ [m]

- Radiant flux: energy / unit time

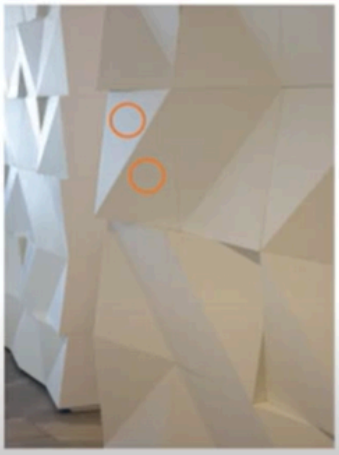
$$\Phi = \lim_{\Delta \to 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt} \quad \left[\frac{J}{s} = W\right]$$

- Irradiance: energy / time / area

$$E(p) = \lim_{\Delta \to 0} \frac{\Delta \Phi(p)}{\Delta A} = \frac{d\Phi(p)}{dA} \quad \left[\frac{W}{m^2}\right]$$



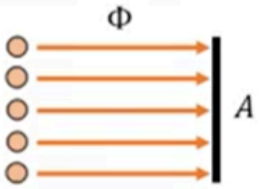
Now that we know about radiometry…



Faceted Walls [Corian]

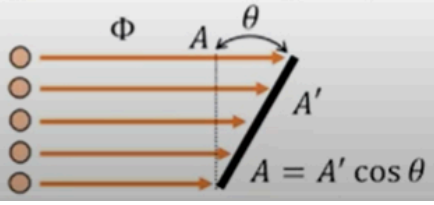
- Why are some surfaces darker and some brighter?
- Light beam of flux Φ incident on surface of area A:

$$E = \frac{\Phi}{A}, \Phi = EA$$



- Now, tilt surface by angle θ relative to light beam

$$E = \frac{\Phi}{A'} = \frac{\Phi \cos\theta}{A}$$



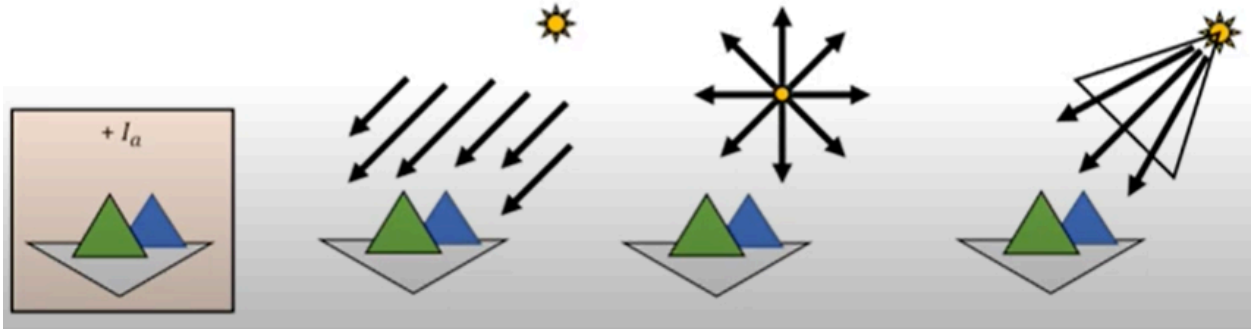
Light will be lighter when arriving perpendicular to the surface, and darker when arriving at an angle to the surface. This is because of the cosine term, leading to lamberts cosine law.

- Simple shading model: "**Lambertian reflectance**" shading, good approximation for "diffuse" (not shiny) surfaces
- Take **dot product of surface normal N and light direction L**
- Then: `color = surface_reflectance * max(0, dot(N,L)) * light_intensity`

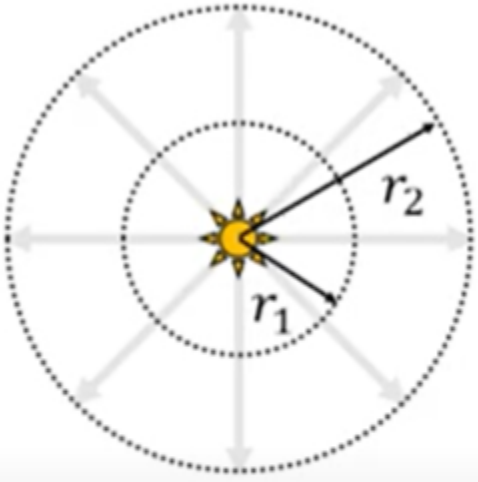
Light_intensity is a variable that affects how much light we are receiving
Surface_reflectance is a variable that affects how reflective the surface is

Let's now talk about light sources

- Ambient light: uniform everywhere & in all directions
- Directional light: distant source, constant light direction vector
- Point light
- Spotlight



Point lights: isotropic point source light. Assume light emits radiant energy in uniformly in all directions.
Consider total flux on spheres of differents radius' r_1 and r_2 and recall that $\Phi = EA$



$$\Phi = E_1 4\pi r_1^2 = E_2 4\pi r_2^2$$

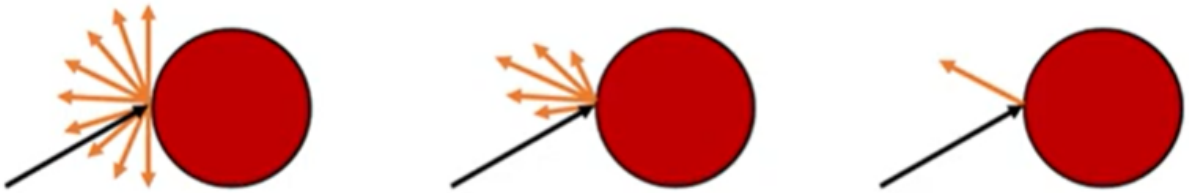
$$\frac{E_2}{E_1} = \frac{r_1^2}{r_2^2} = \left(\frac{r_1}{r_2}\right)^2$$

- Irradiance E falls quadratically with distance from light: proportional to $\frac{1}{r^2}$

Lets come back to surfaces.

Let's come back to surfaces

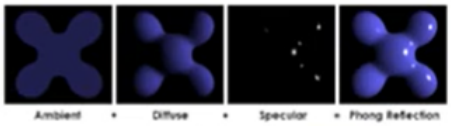
Diffuse / "Lambertian" "Glossy" reflection Specular reflection



Phong reflection model

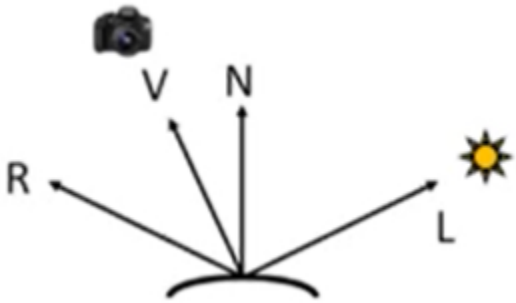
- Three parts: **ambient + diffuse + specular**

```
ca = ka * La;
cd = (kd / d^2) * max(0, dot(N,L)) * Ld;
cs = (ks / d^2) * max(0, dot(R,V) ^ shininess)) * Ls;
```



Phong equation illustration [Brad Smith]

ca = ka * La - Colour Ambient = reflectance * ambient light
cd = (diffuse reflectance coefficient/distance^2)*max(0, N*L) * diffuse light intensity
For cs



Tracks "light spots"
Will appear bright when our camera V is close to the light reflection R.

Looks mirror like

\

shininess is an approximation for how broad or narrow the angle of brightness can be.
This can be expensive as we need to compute dot(R,V) for every point! Idea: use "half-way" vector H between L & V, then replace dot(R,V) with dot(H,N).
This is actually the openGL shading term.

So far we've just talked about illumination models, but what about all points on a surface?
IM - illumination model

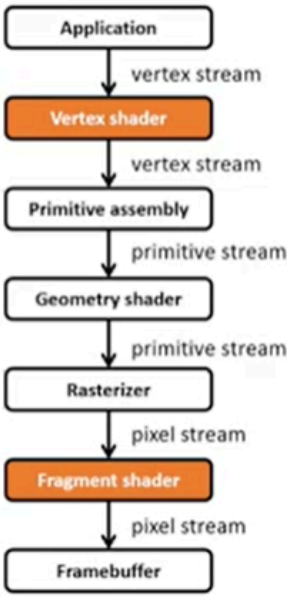
- How to shade over all points on a surface?
- **Flat shading**: evaluate IM once per polygon, use same color for all points on polygon
- **Gouraud shading**: evaluate IM at vertices, interpolate vertex colors across surface
- **Phong shading**: interpolate normals, evaluate IM at each point on surface using interpolated normal at that point

G15 - GPUS and Graphic Pipelines

GPUs are rasterization machines

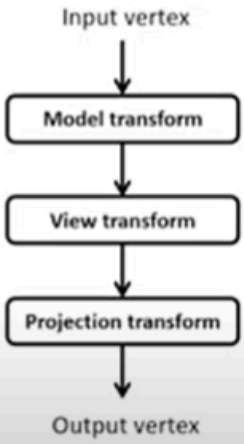
- **Embarrassingly parallel problem**!
  For each pixel sample:
  - Inside-outside test
  - Barycentric interpolation
  - Shading
  - Texture mapping
  - ...
- Massive throughput gains possible by **leveraging parallelism**
- Most **common operations require linear alg**

- Traditional view of graphics pipeline
- Programmable "shader stages" carry out specific tasks
- Application: sets up scene parameters and sends vertex stream with attributes
- Vertex shader: does per-vertex computations and passes on computed attributes
- Fragment shader: does per-fragment computations and passes for output to framebuffer
- Note: shader code is doing parallel stream computations! Written in GLSL (C-like language) for OpenGL/WebGL



Specifically Application

- Model transformation: transforms object to be rendered from its own coordinate frame (model space) to the world space
- View transformation: transforms from world space to camera space (positioning the world within the camera's frame of reference)
- Projection transformation: projects from camera space to clip space; typically in "normalized device coordinates" (NDC) which is a $[-1,1]^3$ cube
- Also: send actual vertex attributes to GPU!



Last 4 minutes of video has vertex and fragment shader pseudo code.

G11 - Texture mapping

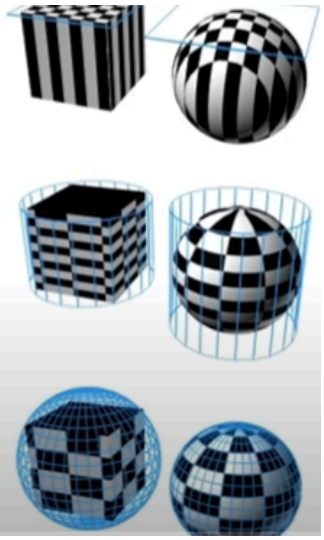
Textures are images representing detailed surface patterns.

They attempt to efficiently store and render surface appearance.

- **Images → pixels:** picture elements
- **Textures → texels:** texture elements

We can project to other objects

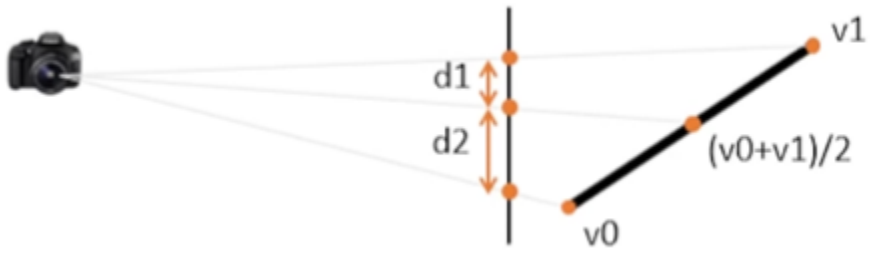
- **Planar projection**
  - Map 2D coordinates (u, v) to plane
  - Equivalent to "dropping" one 3D coordinate, taking vertex (x, y, z) to image coordinates (u, v)
- **Cylindrical projection**
  - From 2D (u, v) to cylinder with cylindrical coordinates
  - Follow surface normal to go from point on cylinder to shape
- **Spherical projection**
  - Map 2D (u, v) to sphere with spherical coordinates
  - Follow surface normal to go from point on sphere to shape
- **General issue: distortions** at "poles" and when intermediate object and surface differ



We can also use UV mapping that maps a (x,y,z) coords to a (u,v) plane. We can make a composite "atlas" system where a section of the uv plane corresponds to a specific 2D surface on the 3D object. (Think of a house)

But let's take a step back for a second, how do we get UV coordinates for each point?

If we try to use barycentric color interpolation, when we project under a specific perspective it will be distorted incorrectly.



Distances in 3D not proportional to distances in screen space.

Perspective-correct interpolation

Transform values before interpolation, interpolate as usual then untransform to get correctly interpolated values.

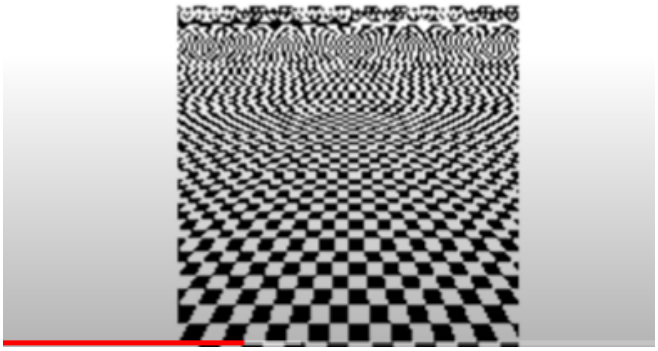
Compute depth from camera z and interpolate u/z, v/z and Z = 1/z.
Then invert and multiply interpolated Z to get correctly interpolated u and v.

Minification and magnification

Sampling can lead to fake artifacts due to aliasing.

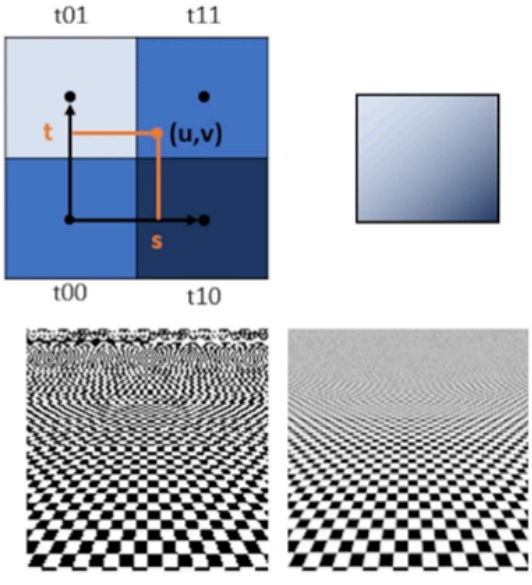
[Akeley & Hanrahan]

- **Magnification:**
  screen pixel ≪ texture texel
  - AKA "over-sampling"
  - Camera close to surface
  - Interpolate value at pixel center
- **Minification:**
  screen pixel ≫ texture texel
  - AKA "under-sampling"
  - Camera far from surface
  - Average over texels in "pixel footprint" to avoid aliasing



How do we fix this? Use bilinear interpolation.

- Take four texel values around the sample point (u,v) coordinate: t00, t10, t01, t11
- Compute coordinates (s,t) of sample point in texel domain from bottom left to top right texel
- Then, **bilinear interpolation**:
  - Bottom row: `(1-s) * t00 + s * t10`
  - Top row: `(1-s) * t01 + s * t11`
  - `(1-t) * ((1-s) * t00 + s * t10)`
    `+  t  * ((1-s) * t01 + s * t11)`



In top part of grid picture above, pixel covers many texels, can get big jumps in color with small shift in screen pixel location.

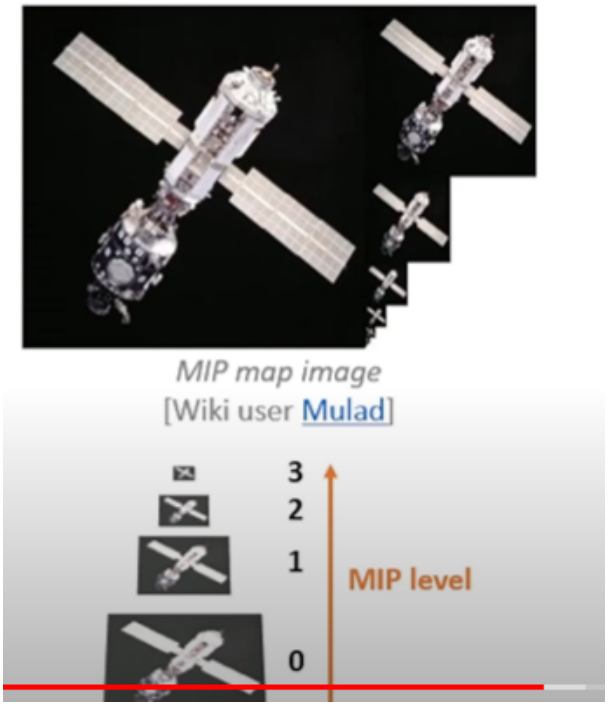
Idea: take average of "pixel footprint" rather than texel color at center.
But: expensive to average so precompute averages in "prefiltered" textures to look up later.

This is essentially the idea of MIP maps: set of down-sampled textures. An image pyramid by repeated down-sampling. When we have 4 pixels we down sample to one picture

- When sampling texture, read single texel at higher MIP levels for larger pixel footprint
- Choose level l using distances d_x, d_y between neighbor pixel samples:

$$l = \log_2\left(\sqrt{\max\left(d_x^2, d_y^2\right)}\right)$$

- Texel averages larger region at higher levels



G12 - Reflection and shadow mapping

Texture mapping - efficiently render surface appearance from textures. Textures can contain more than colors! This leads us to use texture mapping as a general way to associate attributes with surfaces.

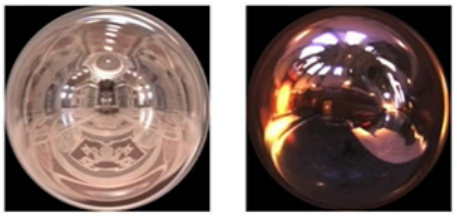
Reflection mapping (environment mapping)

Idea: texture represents surrounding environment

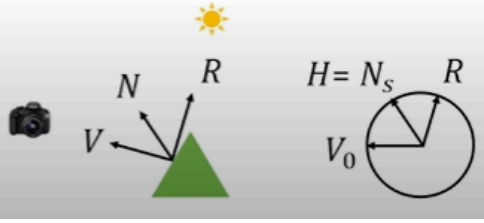
It’s texture maps will enable fast, approximate ray tracing reflections in the rasterization pipeline.

Sphere mapping

- Acquire “light probe” using mirrored ball, and store as environment map
- Assume environment is **infinitely far away from object**, find reflection direction on object, then find point on sphere with same reflection direction
- Note that point (x, y, z) on unit sphere has normal (x, y, z) → half-way vector $H = R + V_0/\|R + V_0\|_2$ has same reflection direction R when viewing V_0
- Thus: $u = \frac{H_x+1}{2}, v = \frac{H_y+1}{2}$



St. Peter’s Basilica & Grace Cathedral Light Probes [Paul Debevec]



Pros:

- Conceptually simple
- Captures 360 vuew
- Efficient: single texel lookup to get reflection at each surface point

Cons:

- Need mirrored balls to get light probes and hard to render using rasterization
- Uneven distribution of environment in texture texels → distortion

We can fix these problems called

Cube mapping

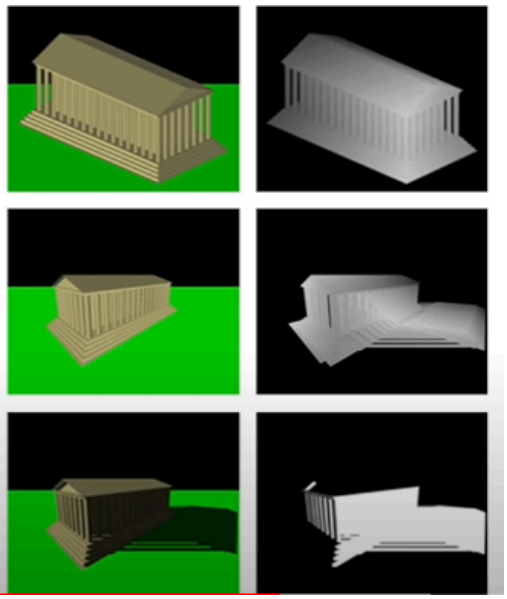
Idea: texture images on faces of a cube represent environment

This will fix distortion, easier to render form just six cameras

But harder to acquire directly in real world, more computationally expensive.

Shadow mapping

- Step 1: **render scene from light, store depth from light in shadow map**
- Step 2: render scene from camera
  - Transform each visible surface coordinate (x, y, z) into point in light-space coordinate frame (x', y', z')
  - "**Shadow test**": look up shadow map value at (x', y'); **if depth value z' < shadow map value, point is not in shadow**
- When shading shadowed points, can simply set to black (or darker) color



The good

- Simple and efficient to compute shadows in rasterization pipeline
- Can pre-compute shadow map for static objects, re-use across camera viewpoints
  Question: what if things move?

The bad

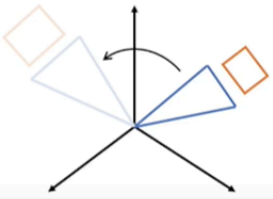
- Shadows have hard edges (more advanced shadow mapping improves on this)
- Spatial and depth resolution of shadow map limits shadow quality (e.g., aliasing when shadow map texels have large footprint)

Shadow Map Antialiasing

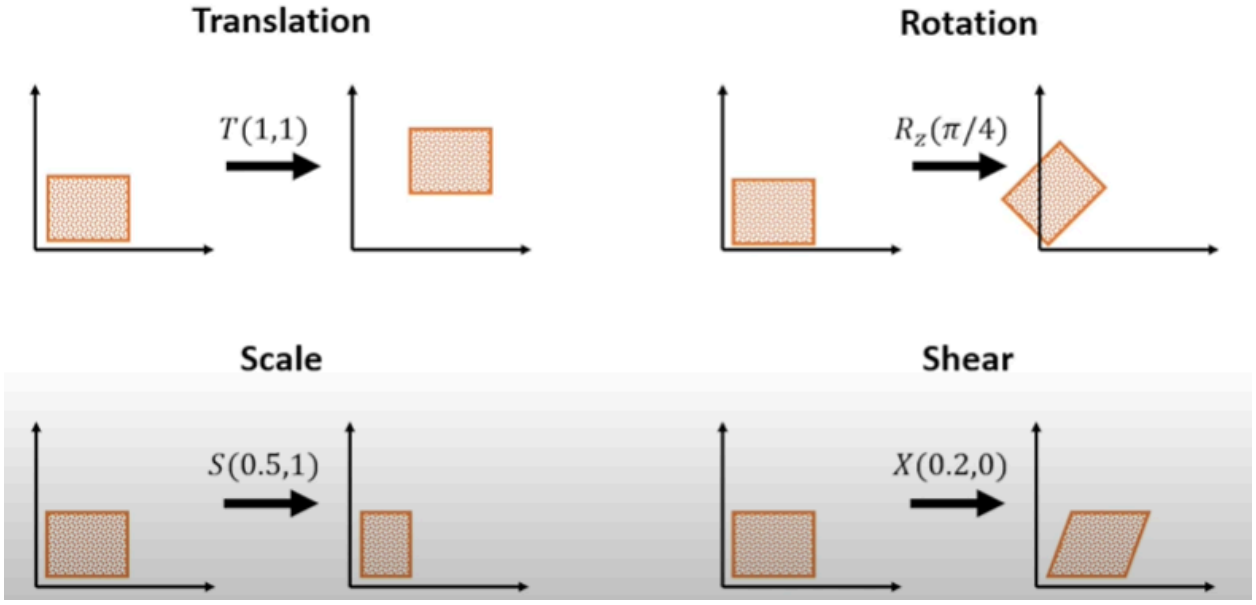
G10 - Transformations and scene graphs

Transformations allow us to replicate objects and place them in other locations within the scene efficiently.

- **Spatial transformation**: $f: \mathbb{R}^n \rightarrow \mathbb{R}^n$ that maps each point $\mathbb{R}^n$ to a point in $\mathbb{R}^n$
- Fundamental in many graphics operations
  - Positioning 3D objects
  - Projecting using a camera model
  - Animating objects over time
  - Mapping 2D textures onto 3D objects
  - ...
- Many useful transformations are **linear maps**
  - Transform lines to lines i.e. $f(\mathbf{ax} + \mathbf{y}) = \mathrm{a}f(\mathbf{x}) + f(\mathbf{y})$
  - Keep origin fixed i.e. $f(0) = 0$



We can also represents as matrices and have a product of matrices represent several transformations (uniform transformations)

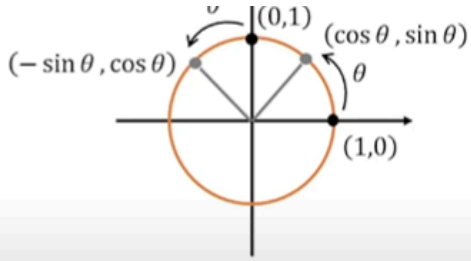


How would we implement a rotation in matrix form?

Rotation by angle theta rotates any point (x,y) around origin by theta, keeping point on circle passing through (x,y)

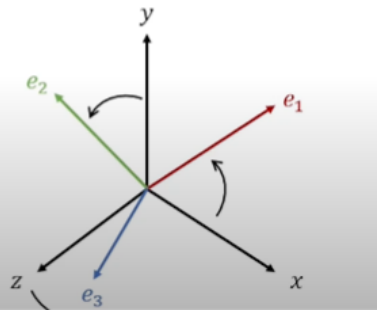
- Consider what happens to points (1,0) and (0,1)
- Map to $(\cos\theta, \sin\theta)$ and $(-\sin\theta, \cos\theta)$ respectively
- For arbitrary point $\mathbf{x} = (x, y)$:

$$R_\theta(\mathbf{x}) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



We extend this to 3D by keeping one axis fixed and then rotate along missing axis in later matrix products. (R_x, R_y, R_z)
Rotation matrices are orthonormal (R^T*R = I → r^T = R^-1)
This means it maps basis axis' to an orthonormal basis e_1, e_2, e_3

$$\overbrace{\begin{bmatrix} - & e_1^T & - \\ - & e_2^T & - \\ - & e_3^T & - \end{bmatrix}}^{R^T} \overbrace{\begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix}}^{R} = \overbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{I}$$



Now scaling. Maps vector to scalar multiple of vector. This is easy.

$$\begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Now translations. Adds offset to a point (x,y) This isn't a linear transformation, it is an affine transformation.

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Solution: add an extra dimension (homogeneous coordinates)

- A 3D point (x, y, z) becomes $(x, y, z, 1)$
- Any homogeneous coordinate point (dx, dy, dz, d) with $d \neq 0$ represents (x, y, z); "de-homogenize" by **dividing all elements by d to get equivalent 3D point**
- Then, translation can be represented in 4x4 matrix form as:

$$\begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Also extend to 4x4 matrix operating on homogeneous coordinates!

Translation

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around z axis

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

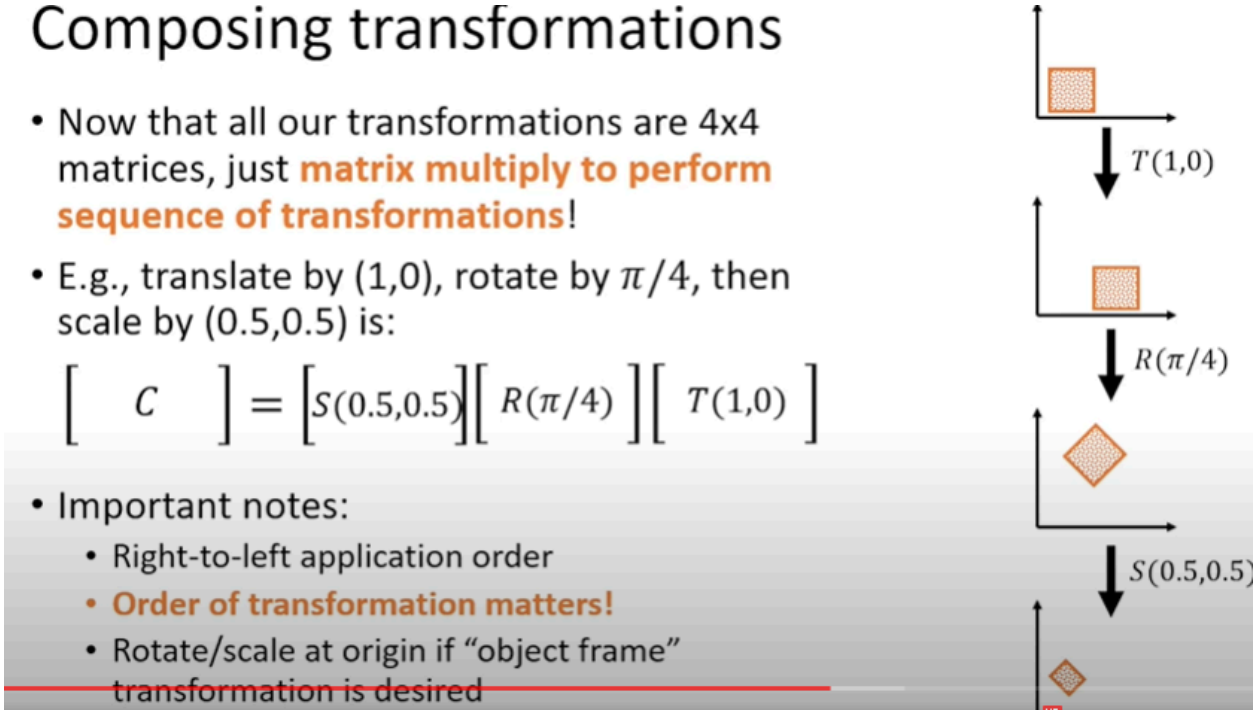
Scale

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around y axis

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now that all of our transformations are 4x4 matrices, we can just multiply matrices for days to get our sequence of transformation.



Another advantage f homogenous coordinates is that we can distinguish direction vectors and points

- Use (x,y,z,1) for points
- Use (x,y,z,0) for direction vectors

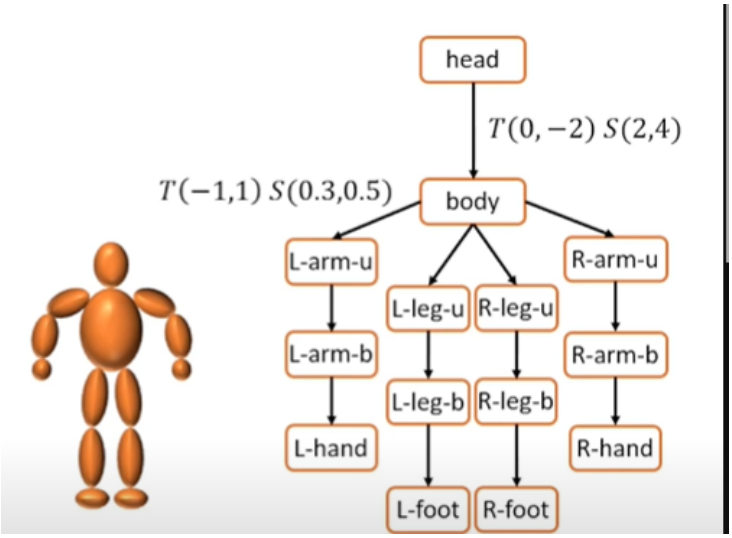
Why? We don't want to translate or scale elements of direction vectors as their direction would be shifted

Scene graphs

Organize objects and transformations with a "Scene graph"

- Scene Graph ~ hierarch of transformed objects
- Nodes store objects
- Edges store 4x4 matrix transformations

Traverse from root and apply transformations along edges



They also allow us to use object instancing. Key idea: nodes in scene graph can use indirection (pointers)

- Point to one copy of 3D geometry, each object node is an "instance"
- Then transform as appropriate when rendering geometry for that node

Massive potential memory and speed savings!

G13 - Animation

Animation is a computational description of motion.

Three techniques of computer animation

1) Artist-driven: keyframing
2) Data-driven: motion capture
3) Physics-based: simulation

1) Keyframing
    a) Main principle: interpolation. Specify values at keyframes, interpolate in between.
        i) Easy to create and edit keyframes
        ii) But it is hard to create long and complex sequences of motion.
2) Motion capture: Use sensors to capture from real world then "replay" motion to virtual objects

a) Passive vs active sensors: optical vs infrared
    i) Easy to get complex, realistic motions.
    ii) Hard to edit and need to observe motion in the real world.

3) Simulation: Generate motion by simulating the physics of moving objects
    a) Typically: approximate complex objects with sets of simpler primitives
        i) Must develop approximations and physics models
        ii) Difference between simulation for engineering/sciences vs. simulation for visual computing: accuracy vs. efficiency
    b) A very common example of simulation is modeling fluids and smoke.
        i) Uses particle systems that have many particles (balls) interacting with simple forces like gravity and repulsion.
        ii) We can also add structure to these particle systems by adding “springs” connected to particles