Lecture 16A
Meshes

Lecture 16 is split in two parts

Lighting is Lecture 16B

It might be Lecture 19

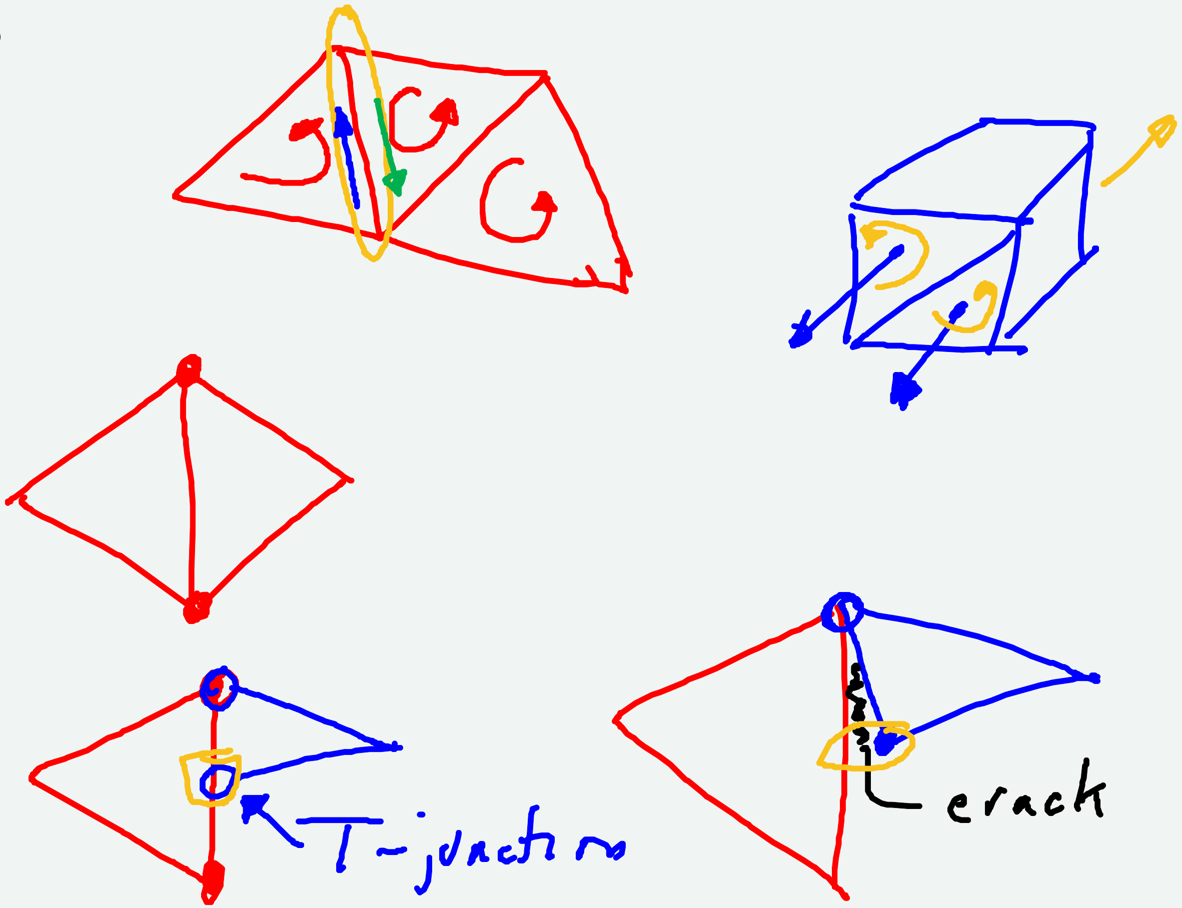
Meshes

Collections of Triangless

- Vertex Sharing
- Vertex Re-Use
- Index Set Representations

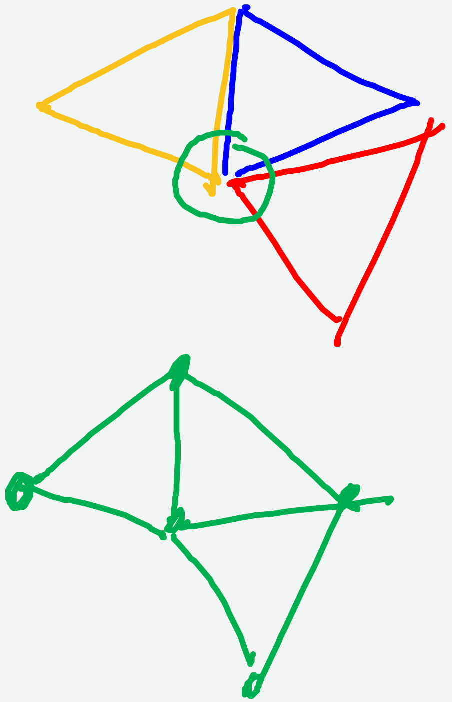
Good Meshes

- Consistency of Handedness
- Avoid Cracking
- Avoid T-Junctions



Why Not Polygon Soup?

- more efficient
- easier to maintain
- easier to check for problems



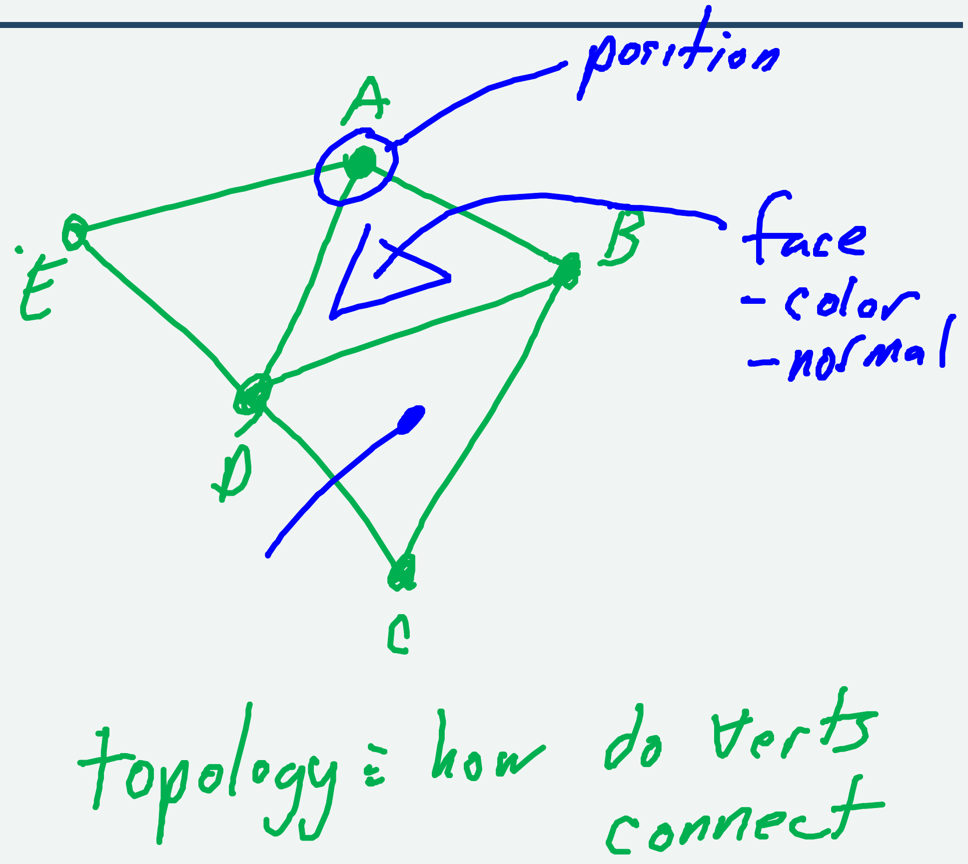
Mesh Properties

Vertex Properties (Barycentric Interpolation)

- vertex colors
- vertex normals (?)

Face properties (constant over face)

- face colors
- face normals
- not actually supported anymore

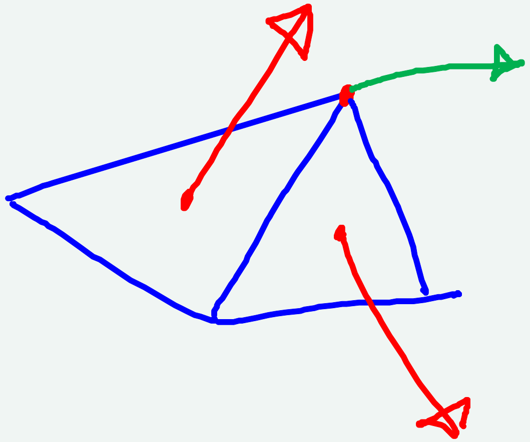


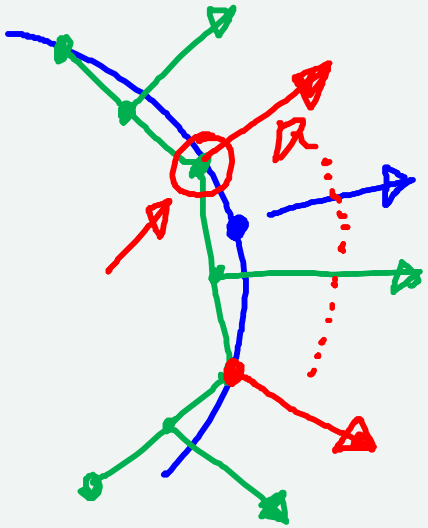
Why vertex normals?

Normals (in math) are a property of a surface (not a point)!

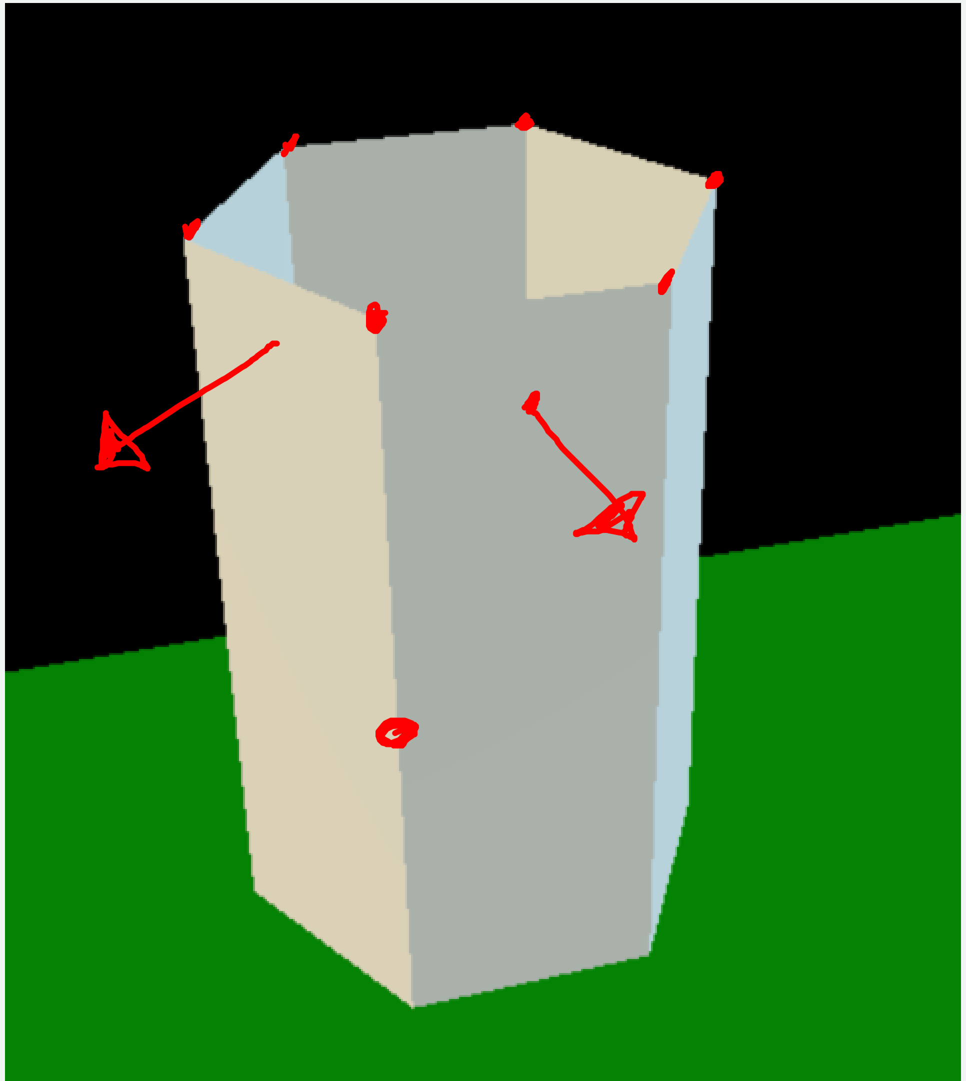
- A triangle has a normal

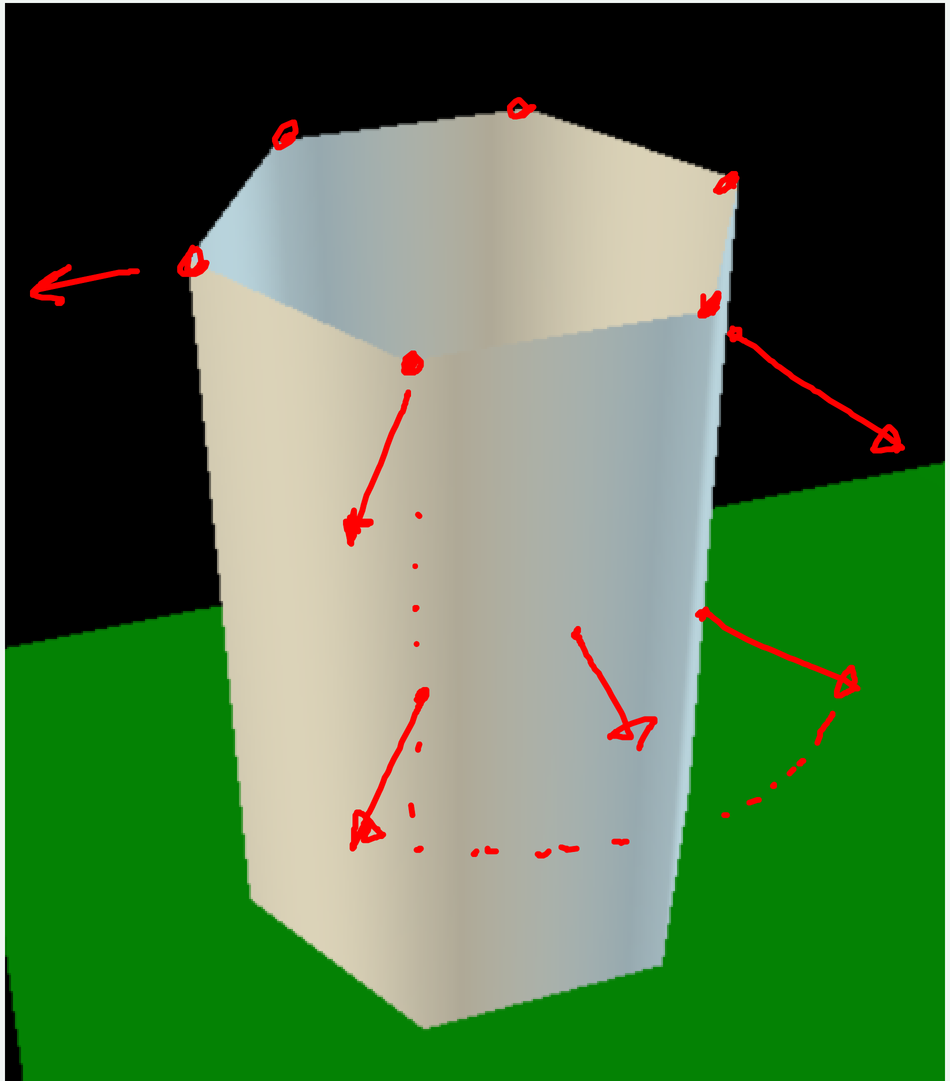
Normals in graphics… might be fake



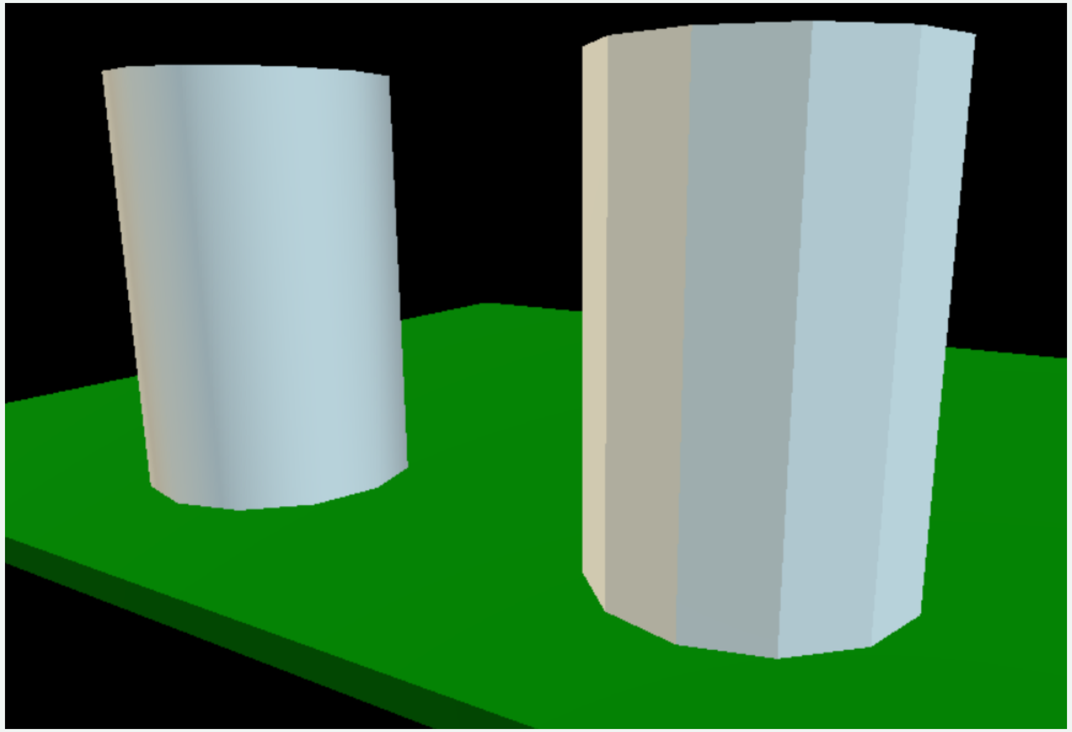


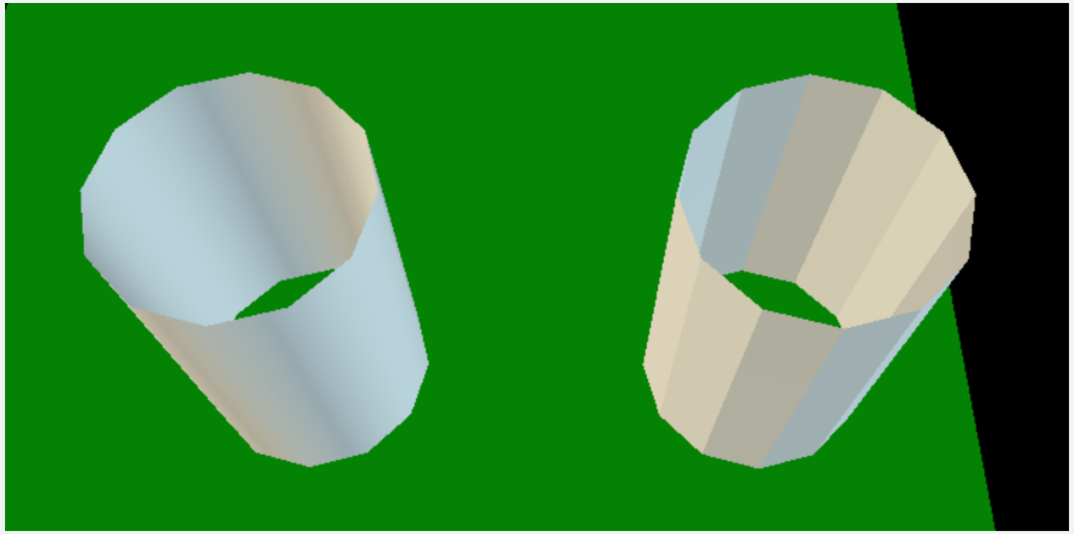
Fake Normals





Fake Normals





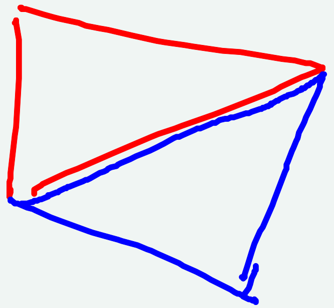
Why vertex normals?

Normals (in math) are a property of a surface (not a point)!

Normals in graphics often are associated with vertices

- Fake smooth surfaces (normals in between faces)
- it's the way hardware works

But what if we really want triangles (not smooth)?



Vertex Splitting

Position is the same - what about other properties?

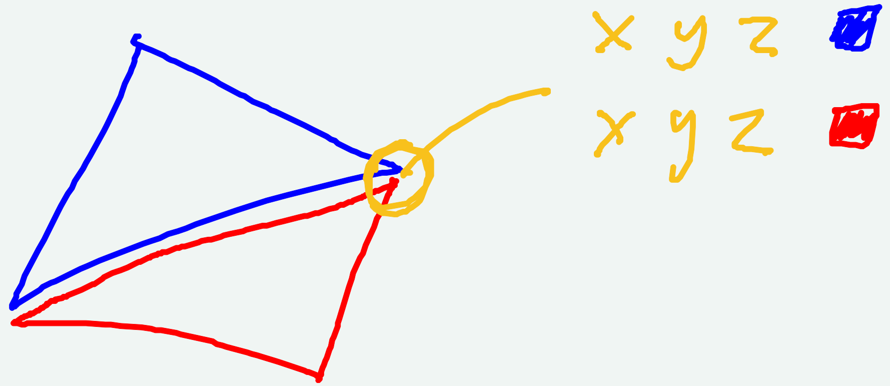
Underlying hardware: a vertex has the same properties

What if each triangle is a different color?

~~THREE takes care of this for us~~

~~- proprties are on **faces**~~

THREE's old data structures did this for us



Good Triangles

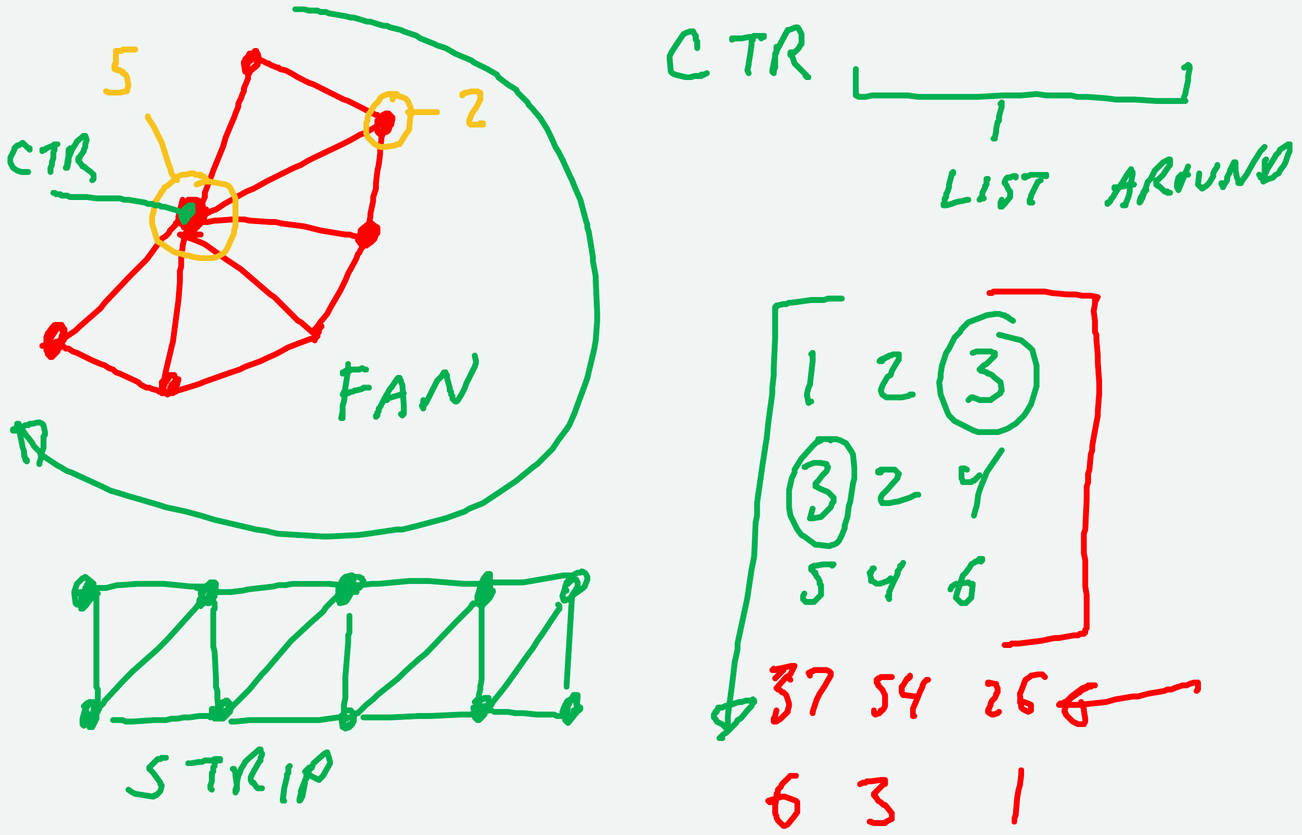
- not too small
- not too elongated

Mesh Operations / Representation

Efficient Display and Storage

- Compact
- Maps well to hardware
  - strips / fans
  - caches
  - format issues

Efficient Manipulation



Why Fancy Mesh Structures?

Fancy Mesh Data Structures

- connection between triangles
- who shares vertices / edges?
- make **Mesh Surgery** easier
- keep consistent when changing

What they do... (in book, if you want to know)

- keep track ofo edges
- find neighbors quickly

What will we do?

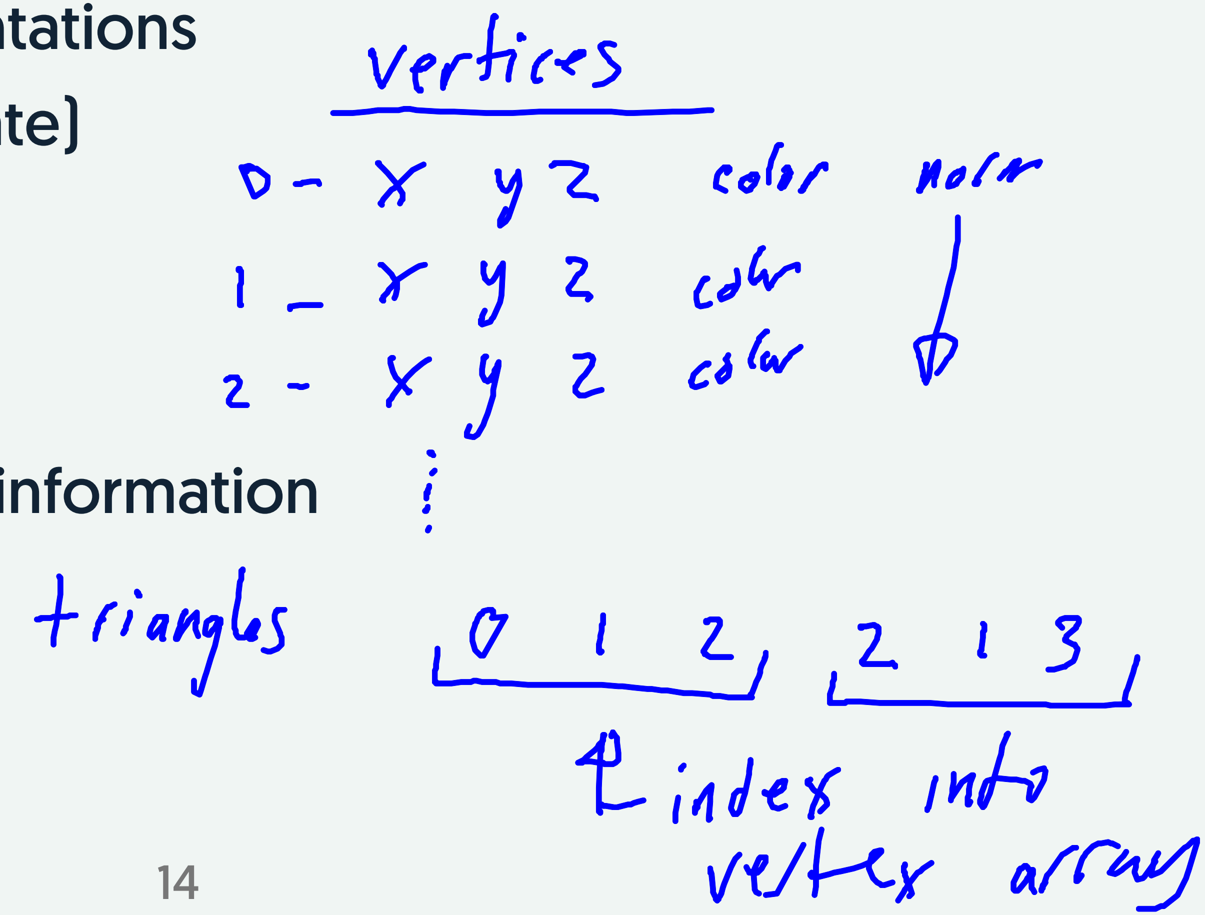
Polygon soup

Vertex sharing / indexed representations

Uniform patterns (when appropriate)

- grids
- strips / fans

Topology is separate from vertex information



In THREE

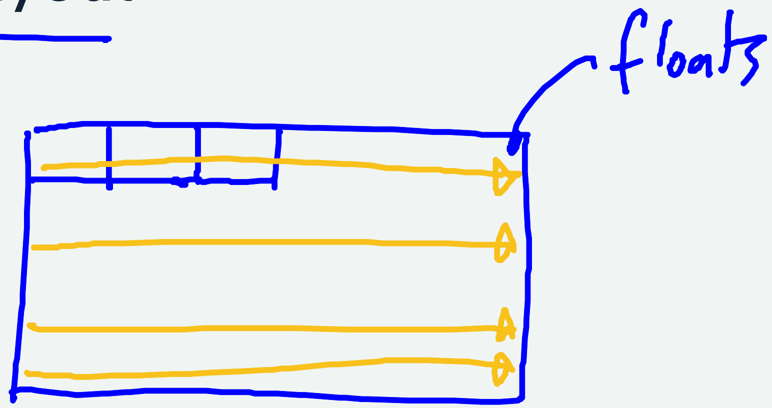
- ~~Geometry - basic "mesh" class~~ ⇒ convert
  - ~~list of vertices~~
  - ~~list of faces - faces have vertex information~~
  - ~~simple JavaScript data structures~~
- BufferGeometry
  - similar content
  - efficient representations (typed arrays)
  - designed for easy transmission to hardware
  - Need to understand **buffers** first

Buffers?

Blocks of memory

Organize for efficient transmission and use

- fixed data type (not dynamic types)
- fixed layout



Attribute Buffers

vertex ← attribute per vertex

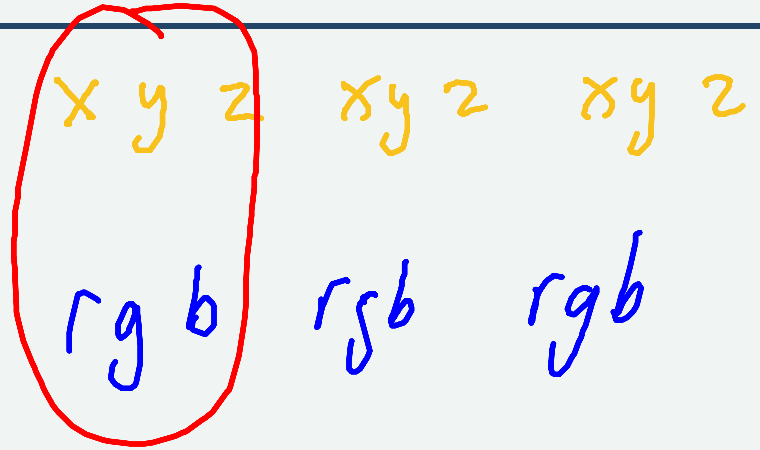
- fixed data type (e.g., Float32)
- fixed item length (e.g., 3 for 3D point)
- THREE calls the `BufferAttributes`

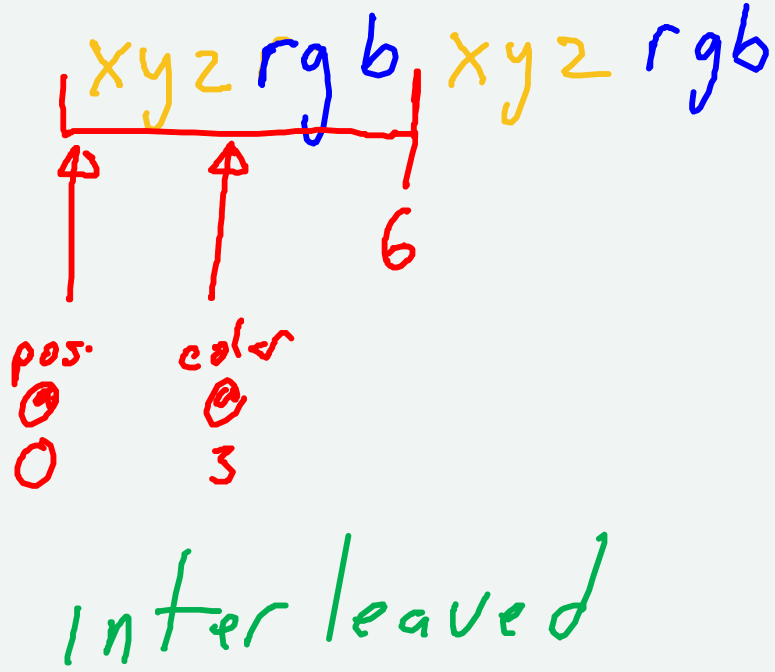
```
const mem = new Float32Array([1, 2, 3, 4, 5, 6 ,7, 8, 9]);
const buf = new T.BufferAttribute(mem,3);
```

Note:

- `Float32Array` type
- 3 values per vertex

Interleaved vs. Non-Interleaved Buffers





Buffer Geometry

- Used to make a mesh
- Attach buffers

```
const mem = new Float32Array([1, 2, 3, 4, 5, 6 ,7, 8, 9]);
const buf = new T.BufferAttribute(mem,3);

const geom = new T.BufferGeometry();
geom.setAttribute("position",buf);
```

9 numbers
3 vertices
3 numbers per vertex

Whatever attributes the material will want/need

```
const geom = new T.BufferGeometry();

const mem = new Float32Array([/* 4 verts * 3 vals/vert = 12 numbers*/] );
const buf = new T.BufferAttribute(mem,3);
geom.setAttribute("position",buf);

const cmem = new Float32Array([ /* 12 numbers */]);
geom.setAttribute("color", new T.BufferAttribute(cmem,3));

const nmem = ... /** set up array of normals */;
geom.setAttribute("normal", new T.BufferAttribute(nmem,3));

// and so on...
```

rgb

Triangles from vertices

1. Triangle soup

   (v0,v1,v2), (v3, v4, v5), ...

2. Indexed

   `setIndex` - takes a list of vertex numbers (integers)

   technically its a buffer (3 verts/triangle, 1 integer per vertex)

How are colors combined?

- The material can have a color(s)
- ~~The face can have a color~~
- The vertices can have colors
- The texture can provide a color (next week)

In THREE:

- material ~~chooses face colors or~~ a single color or vertex colors
- multiply colors together component-wise

Aside... Colors in THREE

Everything is `class Color`

Internally...

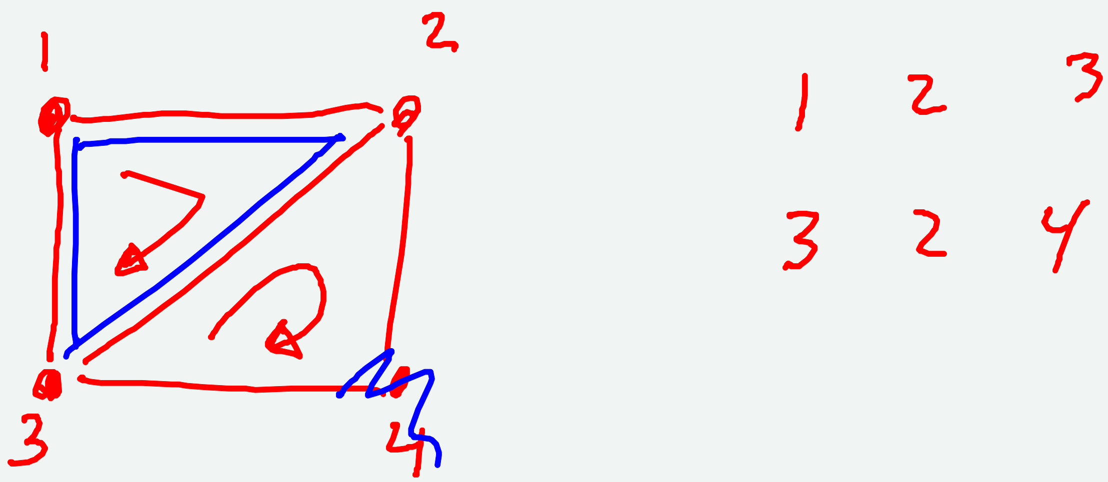
- it stores RGB

Externally

- get / set any way you like
- `.setRGB` (three numbers 0-1), `.setStyle` (CSS string)

Vertex Colors

```
let material =
   new T.MeshStandardMaterial({vertexColors:T.VertexColors});
```



Barycentric Interpolation

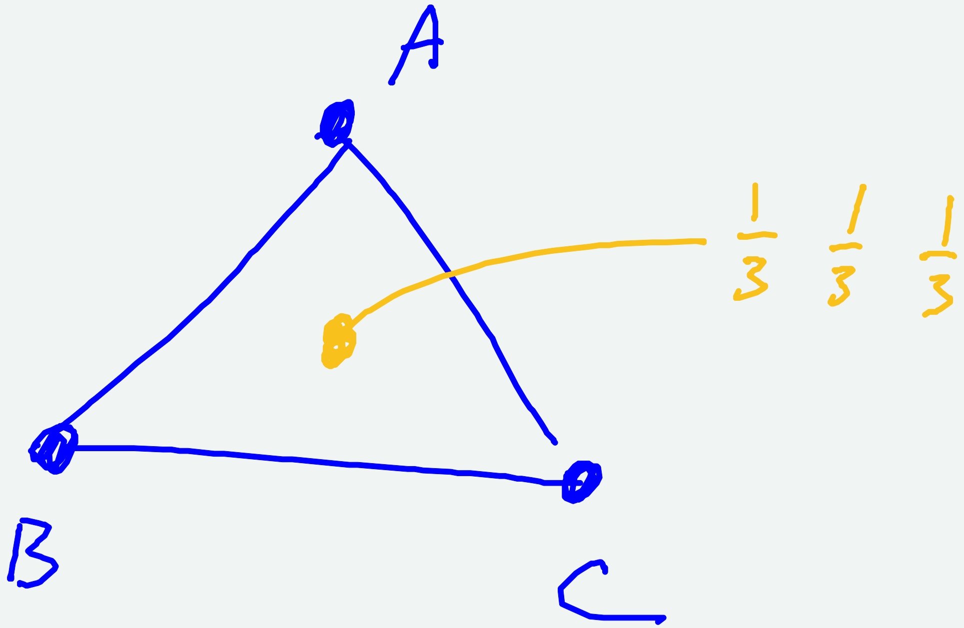
Barycentric interpolation (over a triangle)

$$\mathbf{p} = \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C}$$

where $\alpha + \beta + \gamma = 1$

Gives a **coordinate system**

- for the triangle ($\alpha, \beta, \gamma \in 0 - 1$)
- for the plane



Interpolating Colors (and other Vertex Properties)

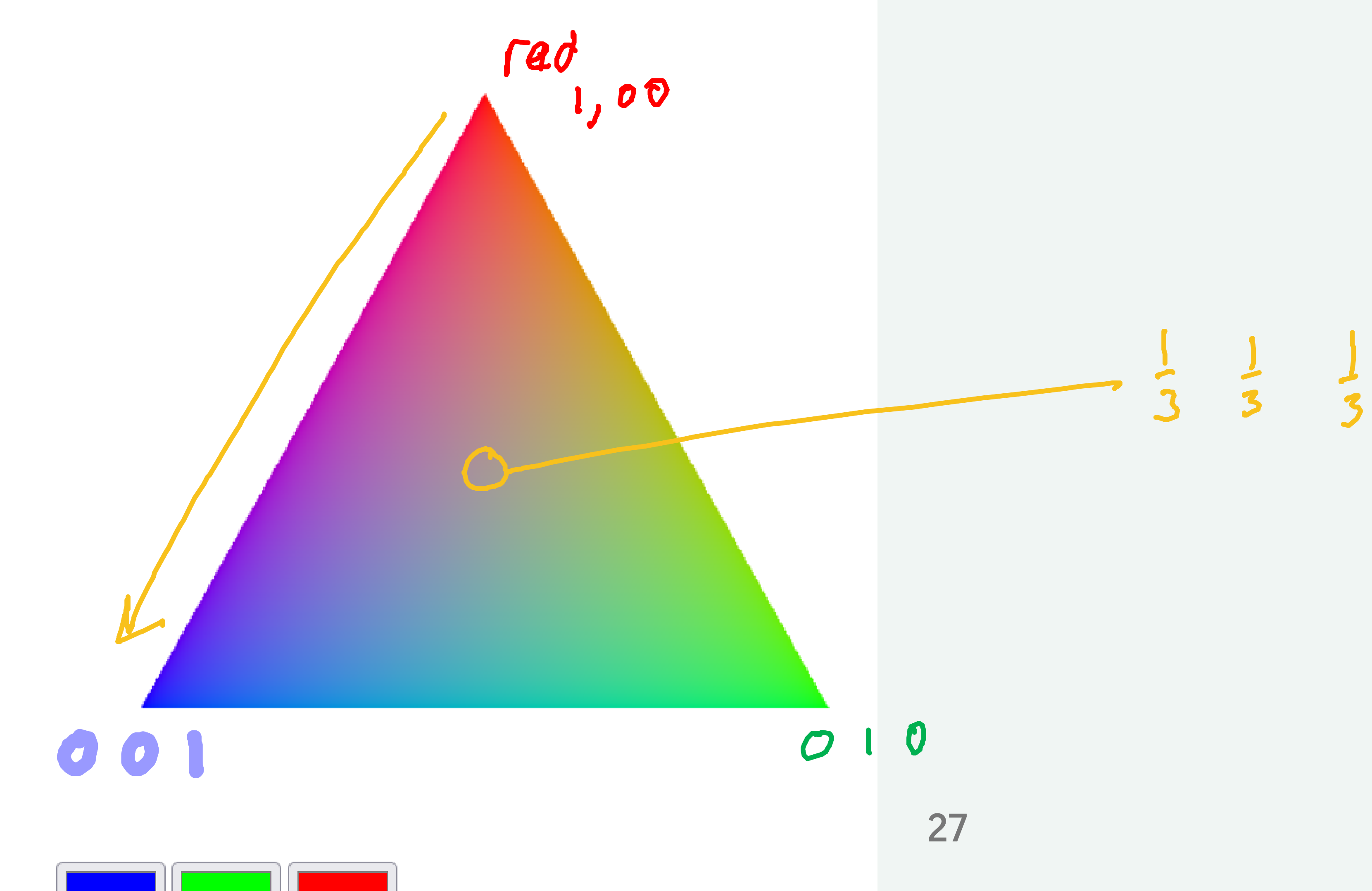
Barycentric interpolation

$$\mathbf{p} = \alpha \mathbf{A}_{\mathbf{pos}} + \beta \mathbf{B}_{\mathbf{pos}} + \gamma \mathbf{C}_{\mathbf{pos}}$$

so...

$$\mathbf{color} = \alpha \mathbf{A}_{\mathbf{color}} + \beta \mathbf{B}_{\mathbf{color}} + \gamma \mathbf{C}_{\mathbf{color}}$$

Barycentric Color Interpolation

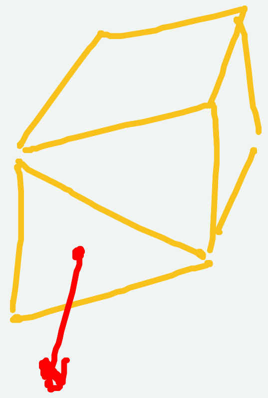


About those normals...

Triangles have an outward facing normal vector

We can compute this by the cross product

- if the vertices are ordered correctly



Why Specify Normals?

- specify outward direction if it isn't obvious
- fake normal directions (pretend a triangle is something else)

Normals

Triangles should have an **outward** facing normal

- cross product **if** the vertices are ordered correctly

We can compute them (THREE can do it for us!)

- requires correctly ordered triangles
- sometimes we "fake" the normals

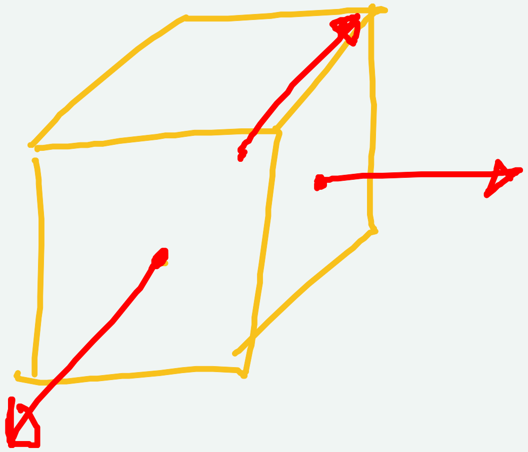
Outward Normals?

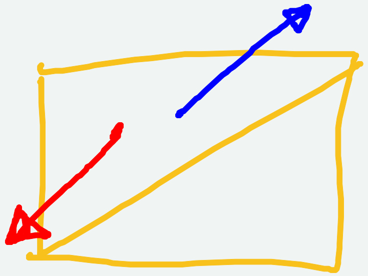
Assumes there is an inside and outside

- front and back of a triangle

By default, THREE only draws the front of a triangle

- need to tell the materials otherwise

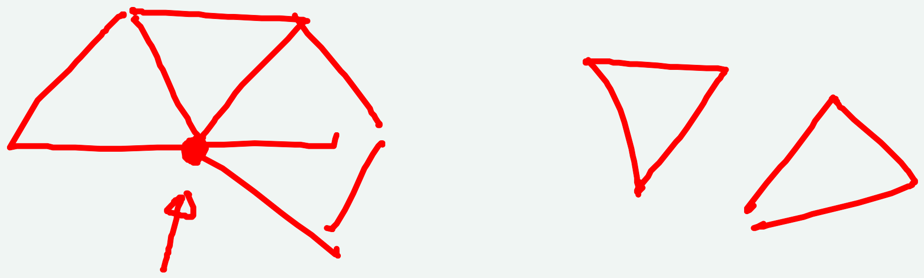




Normals in THREE

Old Style `Geometry` :

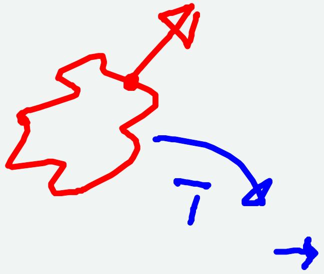
- face normals (auto splitting)
- vertex normals
- compute face normals



New `BufferGeometry` :

- compute normals averages the triangles around the vertex

Transforming Normals



If we transform the **points of a triangle** what happens to its **normal**?

It is a **different** transformation!

- only the 3x3 matrix part (normals are vectors, translations don't matter)
- **adjoint** of the 3x3 part of the transform

The adjoint is the **inverse transpose**

For a rotation, the inverse transpose is the matrix itself

- this is only true for rotations!

Uses of Normals

1. Backface Culling

   THREE.js does backface culling by default

   use `side: THREE.DoubleSide` with your materials for planes

2. Lighting

Uses of Normals

1. Backface Culling

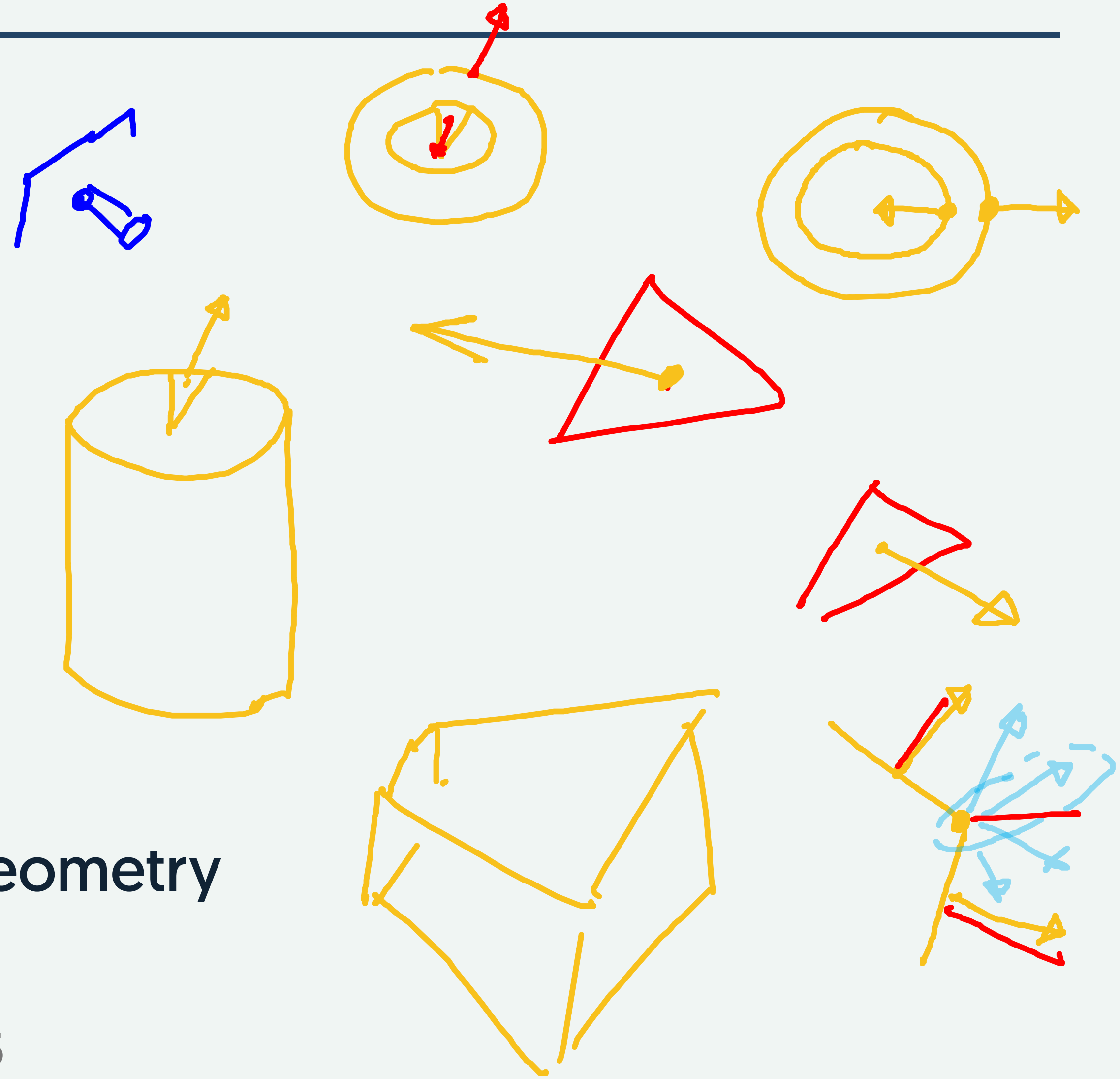
   THREE.js does backface culling by default

   use `side: THREE.DoubleSide` with your materials for planes

2. Lighting

Mesh Summary

- Good Meshes
    - avoid cracks and T-Junctions
    - avoid bad triangles
    - consistent normals
- Data Structures for Efficient Sharing
- Vertex Properties / Vertex Splitting
- Basic Data Structures
- Buffers, AttributeBuffers and BufferGeometry
- Normals



JavaScript Tip

Inheritance is important for Workbook 7

You will make your own **subclasses** of the framework class

(there is a tutorial on the course web, will post to Piazza)

Classes in Javascript

```
class Parent {
    constructor(a,b) {
        this.a = a;
        this.b = b;
        this.c = 10;
    }
    method() {
        console.log(this.a,this.c)
    }
};
```

```
let thing1 = new Parent(1,2);
thing1.method(); // prints 1,10
```

SubClasses in Javascript

```
class Parent {
    constructor(a,b) {
        this.a = a;
        this.b = b;
        this.c = 10;
    }
    method() {
        console.log(this.a,this.c);
    }
};
```

```
class Child extends Parent {
    constructor(b) {
        super(3,b);
        this.c = 20;
    }
}
```

```
let thing1 = new Parent(1,2);
thing1.method(); // prints 1,10
```

```
let thing2 = new Child(5);
thing1.method(); // prints 3,20
```

SubClasses in Javascript

Child class **extends** parent class

Child class has its own constructor

Child constructor calls parent

`super()` - takes parent arguments

`this` doesn't exist until `super()`

```
class Child extends Parent {
    constructor(b) {
        super(3,b);
        this.c = 20;
    }
}
```

Child class uses parent methods

(unless it overrides them)

```
let thing2 = new Child(5);
thing1.method(); // prints 3,20
```

Why do you need to know this?

The CS559 Software Framework uses this!

You define types (subclasses) of `GrObject`

`GrObject` has a list of THREE `Object3D`

You pass the `GrObject` constructor the `Object3D` it should contain

```
class BasicSphere extends GrObject {
    constructor() {
        let geom = new T.SphereGeometry();
        let mat = new T.BasicMaterial({color:"green"});
        let mesh = new T.Mesh(geom,mat);
        super("Basic Sphere", mesh);
        this.mesh = mesh;
    }
}
```