Lecture 16
More Rotations
Meshes

Last Time

- 3D review
- Rotations

Today

- Rotations
- Meshes

How to represent rotations?

1. 3x3 Matrix (or 4x4) (9 numbers)
2. Euler Angles (3 numbers)
3. Axis Angle (4 numbers: vector + angle)
4. Unit Quaternion (4 numbers: magnitude 1)

How do choose?

Understanding Euler Angles

1. Single Axis Rotations
2. Ordered Rotations
3. Local vs. Global
4. Different Sets of Euler Angles

$\Leftarrow R_x R_y R_z \Leftarrow$

$\times$

Composition

One rotation after another…

- how to get the combined rotation?

Composing Rotations

In a single axis (like in 2D):

$$R_z(a) \circ R_z(b) = R_z(a + b)$$

With different axes, this does not hold!

$$R_x(a) \circ R_y(b) = R_?(?)$$

And things in between cause problems

$$R_x(a) \circ R_y(b) \circ R_x(c) \neq R_x(a + c)R_y(b)$$

$R_{x1}\ R_{y1}\ R_{z1} \quad R_{x2}\ R_{y2}\ R_{zy} = R_{x3}\ R_{y3}\ R_{z3}$

Getting Stuck

Rotate about X then Y

Rotate about Z is the same as the first rotate about X

Gimbal Lock

No matter what X is, Y=90 aligns Z with it

- There is no way to get the Y axis out of the X=0 plane
- We lost a degree of freedom

(demo EulerToy3)

Axis Angle (Euler's other theorem)

Demo: et-axisangle

Downsides:

- hard to figure out what axis
- hard to compose

Rotation Matrices

- hard to interpret
- easy to "drift"
- hard to insure it's a rotation
    - Gramm Schmidt Orthonomalization

orthogonal
normal

Unit Quaternions

4 numbers:

- Axis angle: $\underline{\underline{\underline{\theta}}}, \hat{\mathbf{n}}$ vector
- Unit quaternion: $\underline{\cos(\frac{\theta}{2})}, \underline{\sin(\frac{\theta}{2})}\hat{\mathbf{n}}$
- Will have magnitude 1

Why?

What is a Quaternion anyway?

4-dimensional complex number

Consider 2D complex numbers ($a + b\mathbf{i}$)

- we can do arithmetic on them
- multiplication is meaningful

4D Complex Numbers?

Don't worry... you can look up:

- formulas to multiply
- formulas to convert to Matrix form
- formulas to interpolate (and preserve unit-ness)

efficient ways to apply transformation

but you should know...

- these formulas exist
- multiplication preserves unit-ness
- multiplication composes transformations

Why is this better? (or is it?)

- No Gimbal Lock (but antipodes)
- Represents orientations
- Close things are close (except for sign flips)

But Really:

- Easy to compose
- Easy to interpolate (not linear interpolation)
- Other nice math (interpolation)
- 3x3 rotation matrices are a pain
- Easy to fix drift

Convert to Quaternions

(Other direction is MUCH harder)

Axis angle $(\theta, \hat{\mathbf{v}})$ -> $(\cos(\frac{\theta}{2}), sin(\frac{\theta}{2})\hat{\mathbf{v}}))$

x, [1,0,0]

Euler Angles XYZ (x,y,z)

- make a quaterion for each $(\cos(\frac{x}{2}), sin(\frac{x}{2}[1,0,0])))$
- multiply the quaternions together

THREE.js and rotations

Internally, stores quaternions

- it provides all conversions
- it does conversions automatically (beware errors!)
- it provides good quaternion functions
- it gives you operations using other forms
  - axis angle, euler angle, matrix,

You never **need** to see the quaternions… unless you want to

THREE and Rotations

State (variables / orientation)

matrix (normalMatrix, ...)
position
scale
quaternion
rotation

Transforms (motions / rotations)

applyMatrix4
translate (x,y,z, onAxis, ...)
applyQuaternion
rotate (x,y,z, onAxis, ...)

`lookAt` , `setFrom` are special (a method that sets) an absolute orientation

Internally...

The **quaternion** is used for everything

If you do something else, it is converted to the quaternion

If you apply a matrix it must be **decomposed** into rotate, translate, scale

```
        applyMatrix: function ( matrix ) {
                this.matrix.multiplyMatrices( matrix, this.matrix );
                this.matrix.decompose( this.position, this.quaternion, this.scale );
        }, // in Object3D.js
```

Internally

```
translateX: function () {
        var v1 = new Vector3( 1, 0, 0 );
        return function translateX( distance ) {
                return this.translateOnAxis( v1, distance );
        };
}(),
translateOnAxis: function () {
        // translate object by distance along axis in object space
        // axis is assumed to be normalized
        var v1 = new Vector3();
        return function translateOnAxis( axis, distance ) {
                v1.copy( axis ).applyQuaternion( this.quaternion );
                this.position.add( v1.multiplyScalar( distance ) );
                return this;
        };
}(),
```

Old School JavaScript hidden constant

```
translateX: function () {
        var v1 = new Vector3( 1, 0, 0 );
        return function translateX( distance ) {
                return this.translateOnAxis( v1, distance );
        };
}(),
```

A Special Rotation: LookAt

Point the Z axis towards a point

- Useful for cameras
- Useful for other objects

Note this is not unique

- Only specifies 2 dergees of freedom

Up Vector!

Lookfrom / Lookat / Up

- In Three
  - `position` of object center
  - `lookat` method
  - `up` vector (object property)

Internally, it will convert to quaternion

Geometric Derivation

1. Point z at target

   normalize(at - from) $(\widehat{\text{at} - \text{from}})$

2. Find x (right) as $\widehat{up} \times z$
3. Find y (local up) as $z \times x$

Notice: we have built a rotation matrix!

It has all the right properties

We never figured out angles

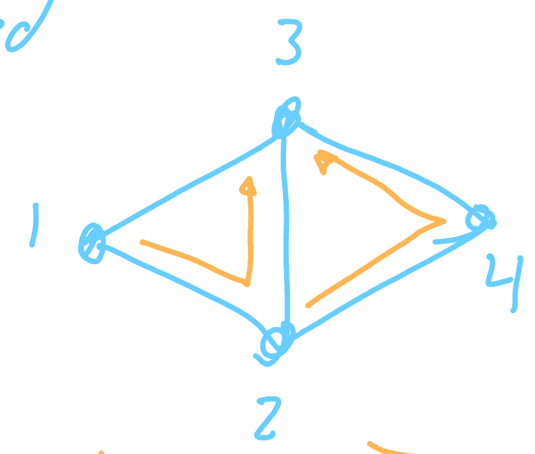
Rotations Summary: What you need to know

1. Basic facts (rigid, orthonormal, composition, ...)
2. Single Axis Rotations
3. Euler Angles - be able to think about them
   - local vs. global
   - how things compose (and complexities)
4. Axis Angle forms - understand what they are
5. Quaternions
   - basic facts - and know they are inside THREE
6. Lookfrom/Lookat/VUp
7. Use in THREE (including centers)

Meshes

Collections of Triangless connected

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



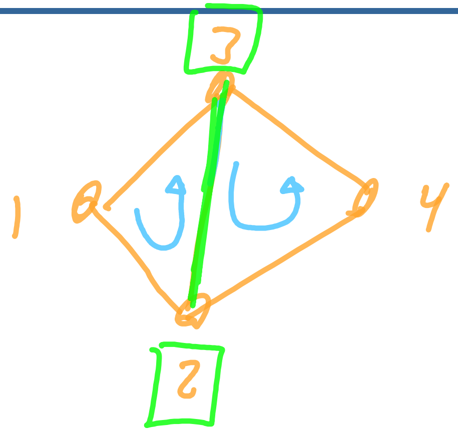
123 243

triangles
123
243

vertices
1
2
3
4

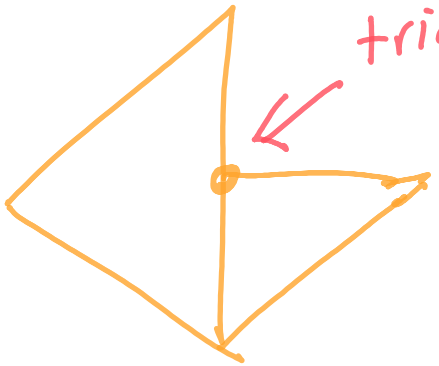
Good Meshes

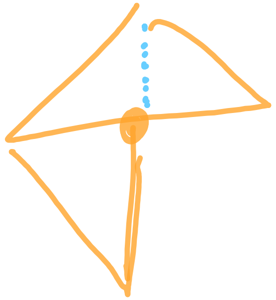
- Consistency of Handedness
- Avoid Cracking – Share vertices
- Avoid T-Junctions



triangles connect @ corners

triangles connect at edge





Why Not Polygon Soup?

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

Mesh Properties (in THREE)

Information about **Meshes** (the whole object)

transformation
material

Information about **Faces**

not supported anymore - just which vertices

face colors

Information about **Vertices**

positions, normals, colors, ...

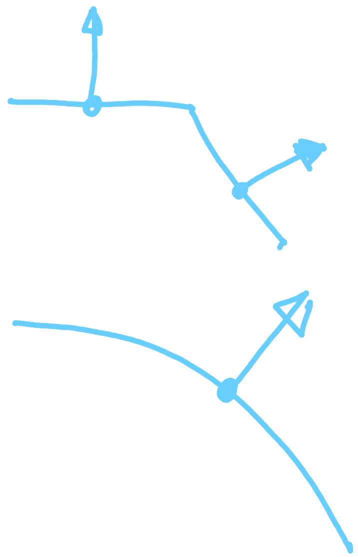
if vertex coloring

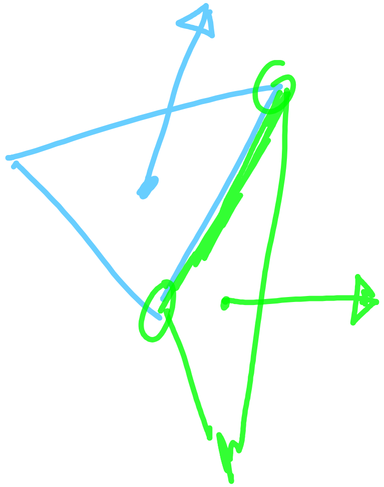
Why vertex normals?

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

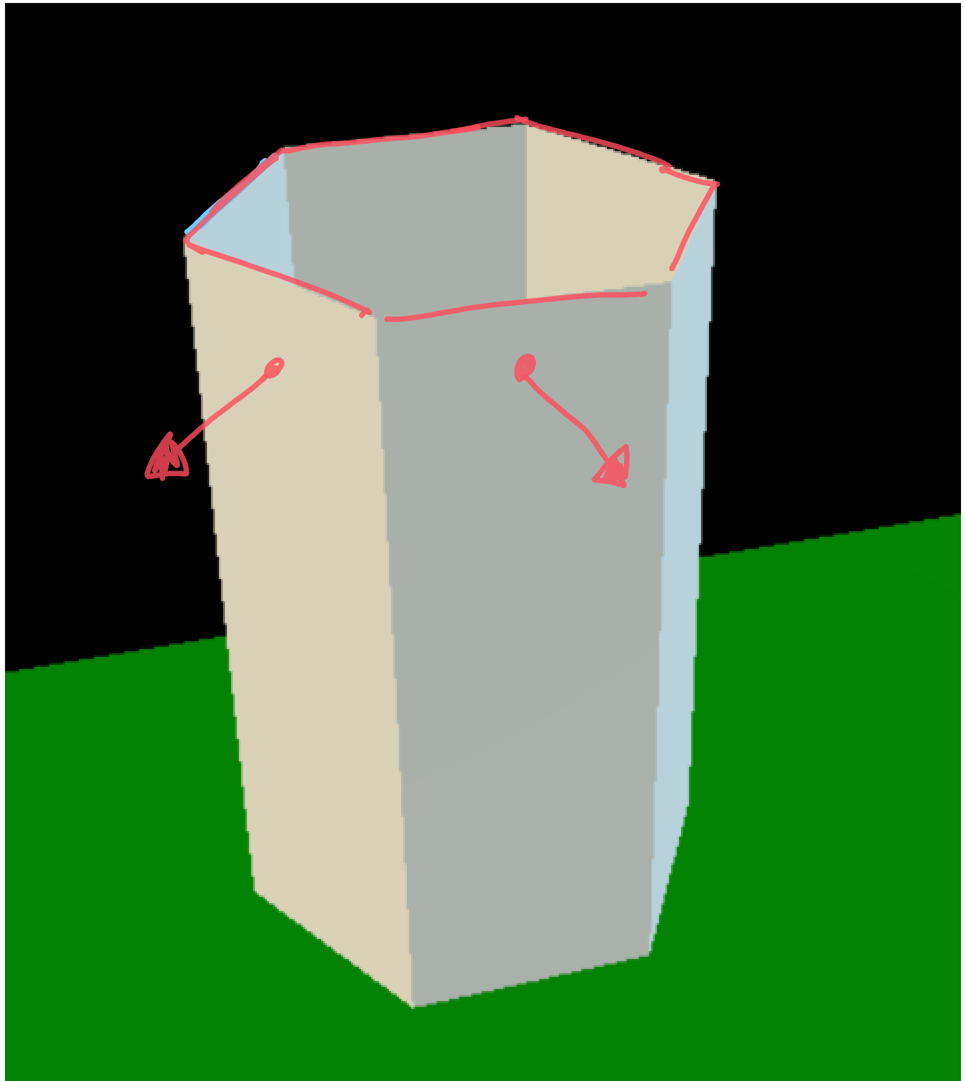
- A triangle has a normal

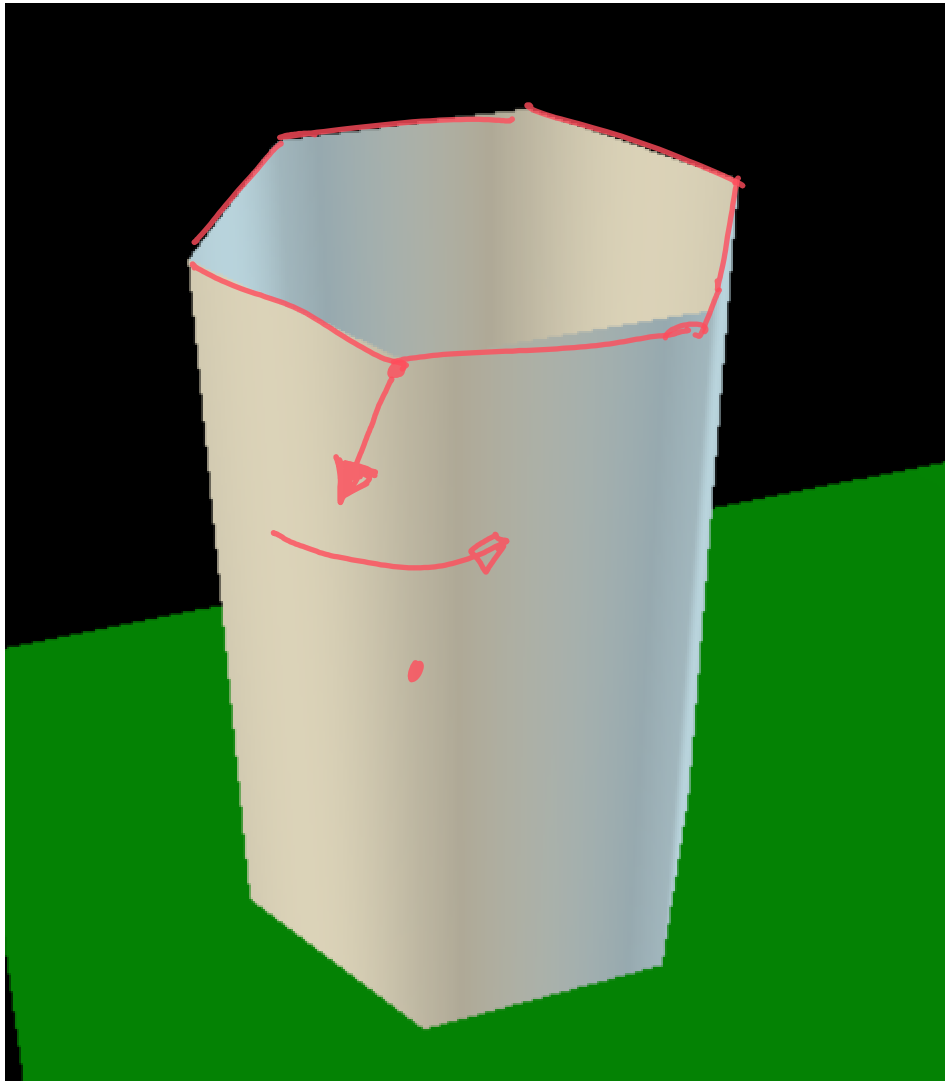
Normals in graphics… might be fake



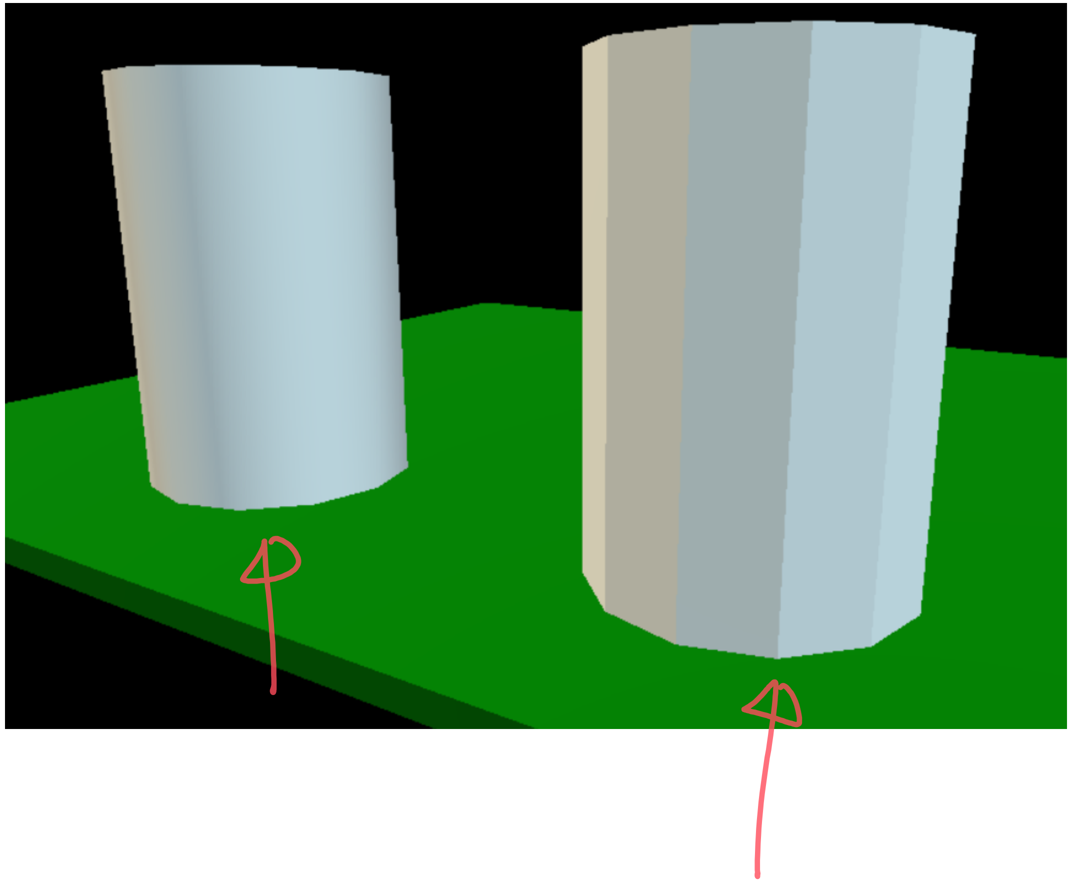


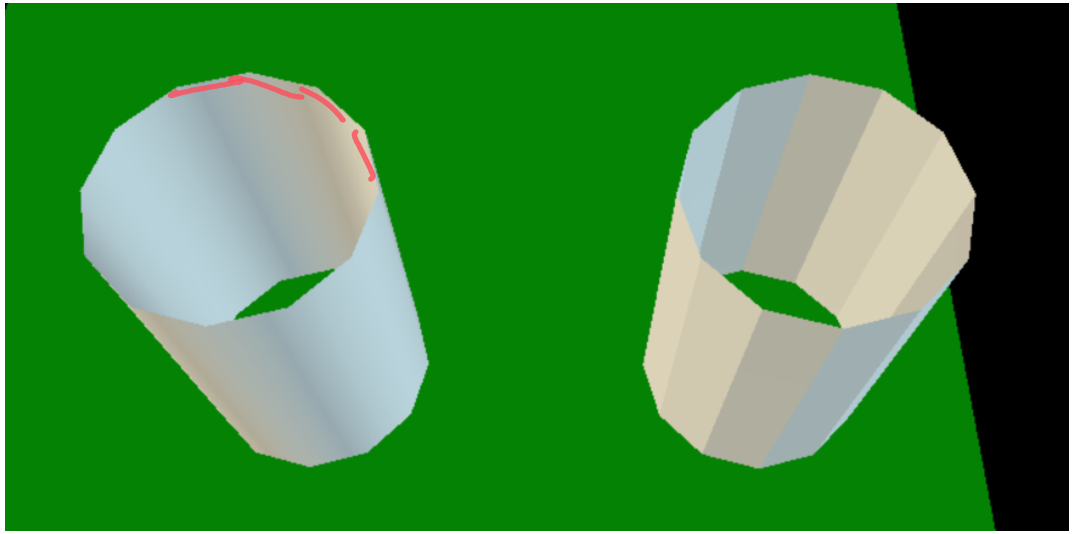
Fake Normals





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

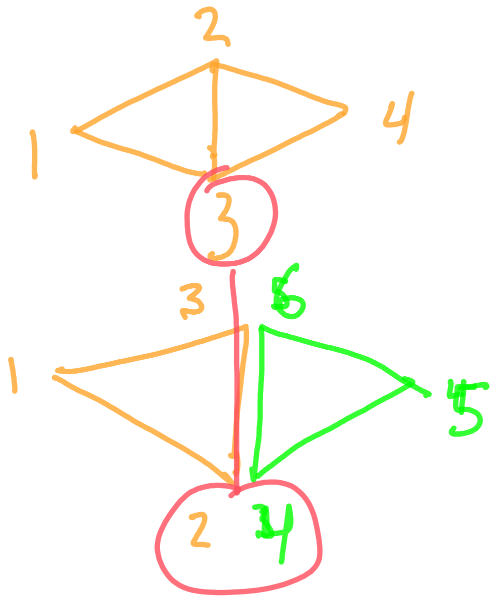
But what if we really want triangles (not smooth)?

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?



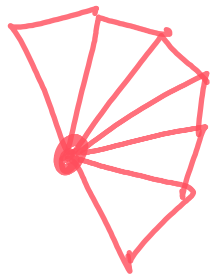
Good Triangles

- not too small
- not too elongated

Mesh Operations / Representation

Efficient Display and Storage

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



Fan

Efficient Manipulation (Fancy Data Structures)

- not in class

In THREE

- 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

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

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...
```

Triangles from vertices

0 1 2
3 4 5
6 7 8

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)

.setIndex([0, 1, 2, 3, 4, 5