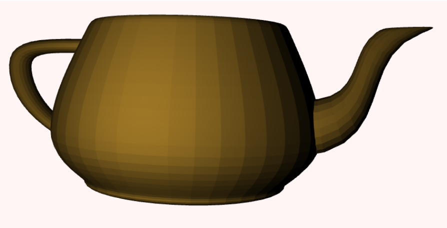
Chapter 11: Shading and the fragment pipeline

K. Jünemann
Department Informations- und Elektrotechnik
HAW Hamburg

Content

Shading



How are these different shadings generated?

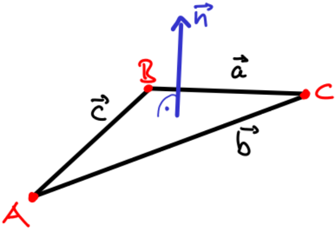
Shading

Blinn-Phong color computations need per-pixel surface normals! (here called fragment normals)

Hierarchy of normal vector calculations:

- *Face normals*: normal vector assigned to entire face
- *Vertex normals*: normal vector assigned to vertices of a face
- *Fragment normals*: interpolation of vertex normals across face (*Phong shading*)

Shading: face normals



- Easy to compute from vertex coordinates:

$$\vec{n} = \frac{\vec{c} \times \vec{b}}{|\vec{c} \times \vec{b}|} \quad \text{with} \quad \vec{c} = \overline{AB},\ \vec{b} = \overline{AC}$$

- Face normals *not* stored in `Geometry` objects!

Shading: vertex normals

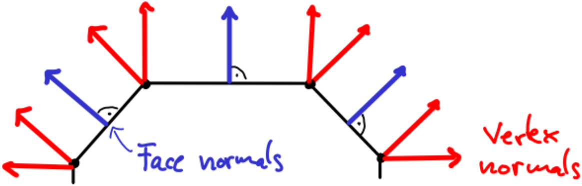
Vertex normals are stored in `normal` attribute of `Geometry` objects:

```
const normals = geo.getAttribute('normal');
```

- normal vector can be set by hand (with `setAttribute` method) or be computed with `computeVertexNormals` method.
  - `computeVertexNormals` computes normal vector as *average* of all normals of faces a vertex is part of.
- Indexed geometries: a vertex has a single normal vector
- Non-indexed geometries: vertex copies can have different normal vectors

Shading: vertex normals

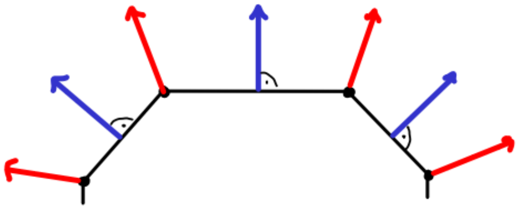
- *Option 1*: *Flat surface normals*
  Vertex normals are identical to face normals.



  - Each vertex has several *different* normal vectors.
  - Suitable for non-smooth objects (cube, pyramid, etc.).
  - Can be achieved with `computeVertexNormals` for *non*-indexed geometries.

Shading: vertex normals

- *Option 2*: *Smooth surface normals*
  Vertex normals are set to *average* of face normals over faces the vertex is part of.

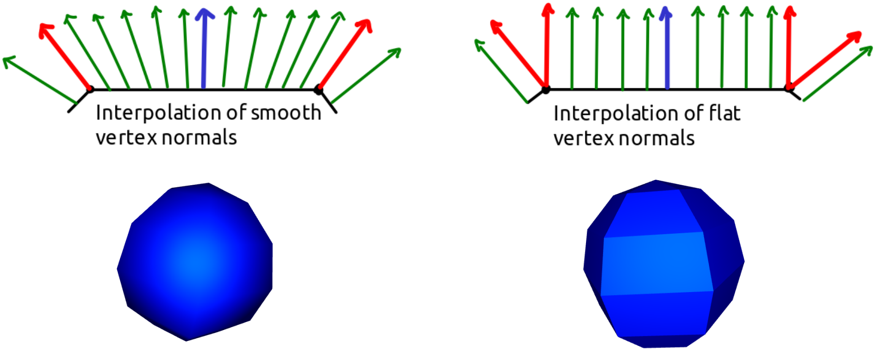


  - All normals assigned to a vertex are identical.
  - Suitable for smooth objects (sphere, torus, etc.).
  - Can be achieved with `computeVertexNormals` for indexed geometries.

Shading: fragment normals

Phong shading assigns each fragment of a face a normal vector by interpolation of vertex normals.

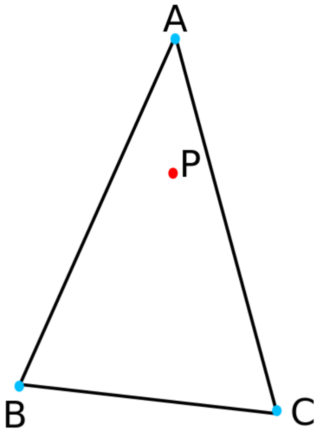
- Default in `MeshPhongMaterial` and `MeshStandardMaterial`



- fragment normals *not* stored in geometry objects
- (re)computed for every frame.

Shading: how to interpolate across a face

Digression: *Barycentric coordinates* and interpolation across a face



Each point P of a face can be uniquely written as

$$P = \alpha A + \beta B + \gamma C$$

with *non-negative* coefficients α, β, γ and

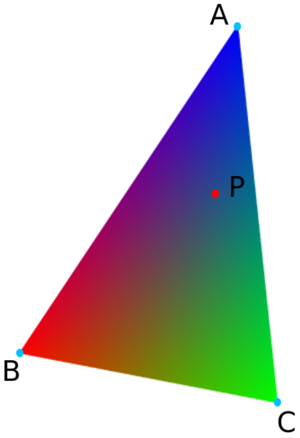
$$\alpha + \beta + \gamma = 1.$$

(α, β, γ) are called *barycentric* coordinates.

Shading: how to interpolate across a face

Barycentric coordinates can be used to interpolate *anything* across a face.

- implemented deep down in the graphics engine



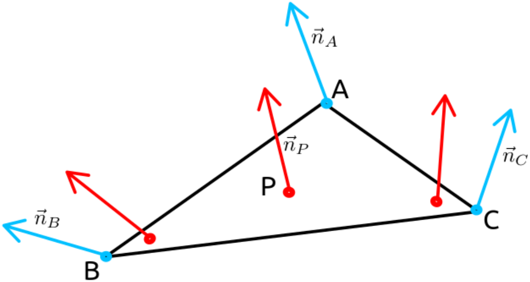
Example 1: color interpolation

color of a point P with barycentric coordinates (α, β, γ):

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} = \alpha \underbrace{\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{blue}} + \beta \underbrace{\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}}_{\text{red}} + \gamma \underbrace{\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}}_{\text{green}}$$

Shading: how to interpolate across a face

Example 2: normal vector interpolation



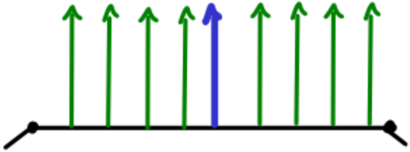
Normal vector $\vec{n}_P$ at point P with barycentric coordinates (α, β, γ):

$$\vec{n}_P = \alpha \vec{n}_A + \beta \vec{n}_B + \gamma \vec{n}_C$$

- $\vec{n}_A$, $\vec{n}_B$, $\vec{n}_C$: vertex normals

Shading: flat shading

Flat shading: a shortcut to achieve flat fragment normals!



- Easier: tell renderer to use face normals for all color computations
- Saves face interpolation procedure!
- Turn this on with `Material.flatShading = true`
  - default is *false*: Phong shading

Shading: Garoud shading

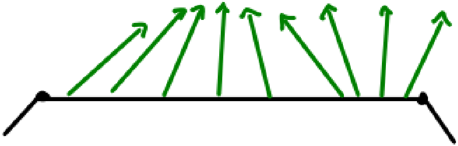
An alternative shading method: *Garoud shading*

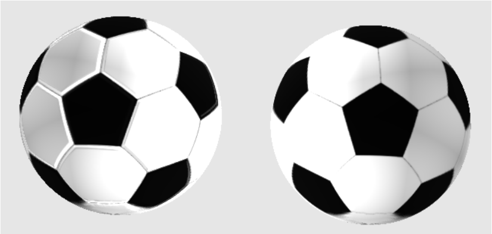
- Step 1: calculate color at each vertex using vertex normals
- Step 2: interpolate vertex color across face
  - Quality less good
  - More efficient than Phong shading
  - Implemented in `MeshGaroudMaterial` (examples folder)
  - Has no specular reflection
  - Garoud shading in general has problems with focussed specular highlights

Shading: normal and bump maps

Yet another way to define fragment normals

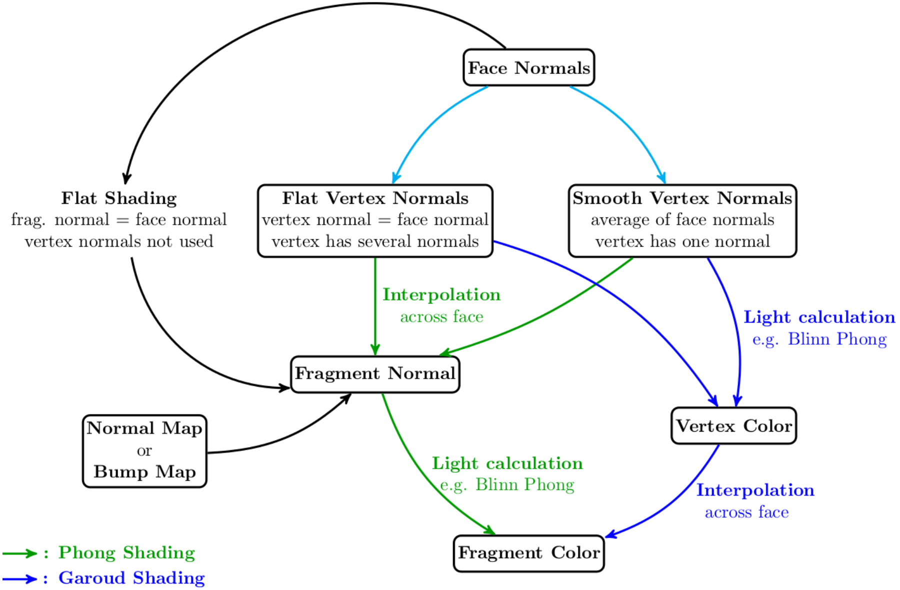
- *Normal maps*: user defined fragment normals
- *Bump maps*: fragment normals computed from height map





- see next chapter on textures

Shading: Summary



Shading: configuration in `three.js`

- *Phong shading:*
  - By default used by:
    - `MeshPhongMaterial`
    - `MeshStandardMaterial`
    - `MeshPhysicalMaterial`
  - Geometries must contain suitably defined vertex normals.
    - can be (re-)computed with `computeVertexNormals` method of `BufferGeometry` class.
- *Garoud shading:* used by
  - `MeshGaroudMaterial`
- *Flat shading:*
  - Turn this on with `Material.flatShading = true` (default: `false`)
  - Vertex normals are ignored in this case

Shading: configuration in `three.js`

More on `computeVertexNormals`: result depends on definition of geometry (see chapter 5):

- Indexed geometry:
  - Vertex is part of *several* faces
  - average of these face normals is computed for each vertex (average is weighted by face area)
  - result: *smooth* vertex normals
- Geometry with duplicated vertices:
  - each vertex is part of *just one* face
  - vertex normal is same as face normal
  - result: *flat* vertex normals

Shading: storage of vertex normals

A vertex normal consists of 3 numbers per vertex:

- stored as a buffer attribute called *normal*
- vertex coordinates are a buffer attribute called *position*
- other buffer attributes: *uv* (see chapter 12) and *color*
- set with `BufferGeometry.setAttribute` function
  - `setFromPoints` method is wrapper for `setAttribute('position', ...)` method.

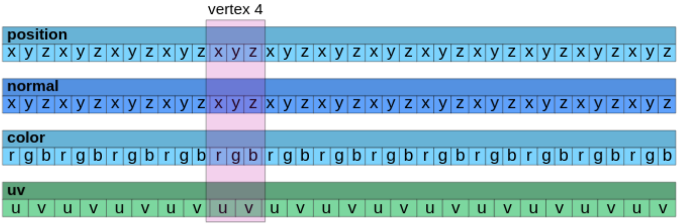
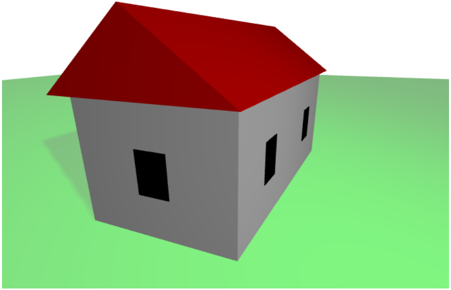


image from *Three.js Fundamentals* chapter about BufferGeometry

Shading

Example: Add suitable vertex normals to the roof of our house:



To visualize normal vectors use

- `VertexNormalsHelper`

The graphics pipeline (2): fragment part

Goal: Calculate color of each pixel on the screen!

Given: A list of faces (i.e. triangles), each with 3 vertices

Additional information:

- Fragment processing takes place *after* vertex processing: for each vertex, we know
  - its position on the viewport (i.e. display screen)
  - its z_{ndc} coordinate
- Various 'attributes' are attached to vertices (e.g. normal vectors, a color)

The graphics pipeline (2): fragment part

For each face, do the following steps:

Step 1: Rasterization: split face into fragments

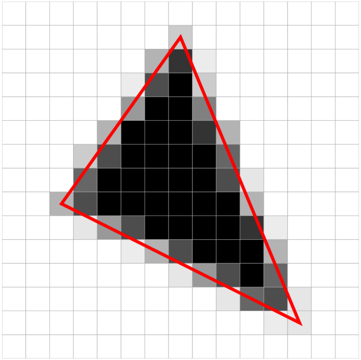
Step 2: Interpolation of vertex information

Step 3: Discard hidden fragments

Step 4: Calculate color of each fragment

The graphics pipeline (2): fragment part

Step 1: *Rasterization* splits a face into fragments



Source: Wojciech mula, Polish Wikipedia

Important effect: some pixels only partly located inside face (see below)

The graphics pipeline (2): fragment part

Step 2: Interpolation (has been discussed above)

Important quantities to interpolate

- The normalized device coordinate z_{ndc}
- In case of Phong shading: The normal vector
- In case of Garoud shading: The color of the vertex
- The position (e.g. in camera space)

The graphics pipeline (2): fragment part

Step 3: Discard hidden fragments

- Each fragment has an interpolated z_{ndc} value
- Each screen pixel has an associated *z-buffer*

z-buffer algorithm:

For each fragment of all objects in our scene:

    `if`

        z-buffer at screen position of current fragment
        contains value smaller than current z_{ndc} value

    `then`

        discard current fragment

    `else`

        keep current fragment *and* store current z_{ndc} value

- simple but inefficient since *all* faces are processed $\Longrightarrow$ optimized versions exist.

The graphics pipeline (2): fragment part

Step 4: Calculate color of each fragment

- Phong or flat shading: use
  - position of current fragment
  - position of light sources and camera
  - normal vector at current fragment
  - material properties

  to calculate color of each fragment.
- Garoud shading: color computation done at vertex level (3 times per face $\implies$ faster!)
  Fragment color is interpolated vertex color.

The graphics pipeline (2): fragment part

Programmable shaders: WebGL provides two entry points into pipeline:

- *Vertex-Shader*: calculates vertex position in homogeneous clip space
  - receives various matrices and other data from Javascript (uniforms and attributes)
  - sends quantities to be interpolated to Fragment Shader
- *Fragment-Shader*: calculates color of fragment based on data received from vertex shader as interpolated values

Shaders are written in *OpenGL Shading Language* (glsl):

- Similar to C with built-in vector and matrix types
- executes directly on GPU
- accessible in `three.js` through `ShaderMaterial` and `RawShaderMaterial`

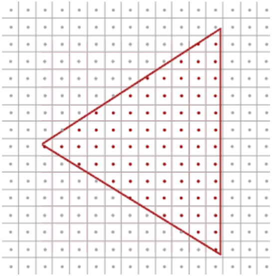
Aliasing

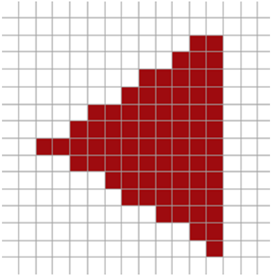
Rasterization asks the following question:

Is a fragment part of a face?

What type of answer can we expect?

Option 1 (*Aliasing*): Just *yes* or *no*

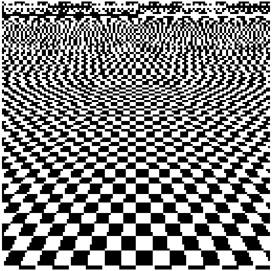


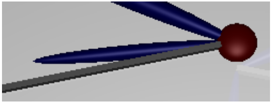


Source: `https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing`

Aliasing

Aliasing leads to suboptimal image quality:

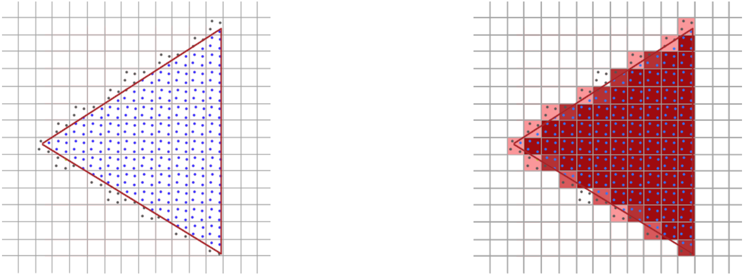




Source (left image): `https://en.wikipedia.org/wiki/Spatial_anti-aliasing`

Aliasing

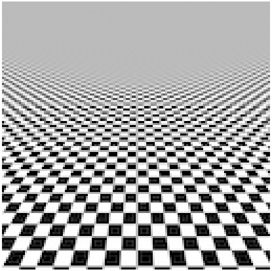
Option 2 (*Anti-Aliasing*): If a pixel belongs to a face is quantified by a number between 0 and 1: its fraction of area inside face) $\implies$ Color value is interpolated accordingly.

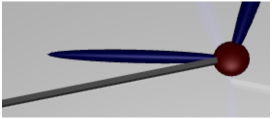


Source: `https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing`

Aliasing

Anti-Aliasing requires more computations but leads to better image quality:





Source (left image): `https://en.wikipedia.org/wiki/Spatial_anti-aliasing`

Anti-Aliasing is turned off in `three.js` by default. Turn it on with

```
renderer = new THREE.WebGLRenderer({canvas,
                                    antialias:true});
```