Lecture 5: Mesh Shape Representation

DHBW, Computer Graphics

Lovro Bosnar

1.2.2023.

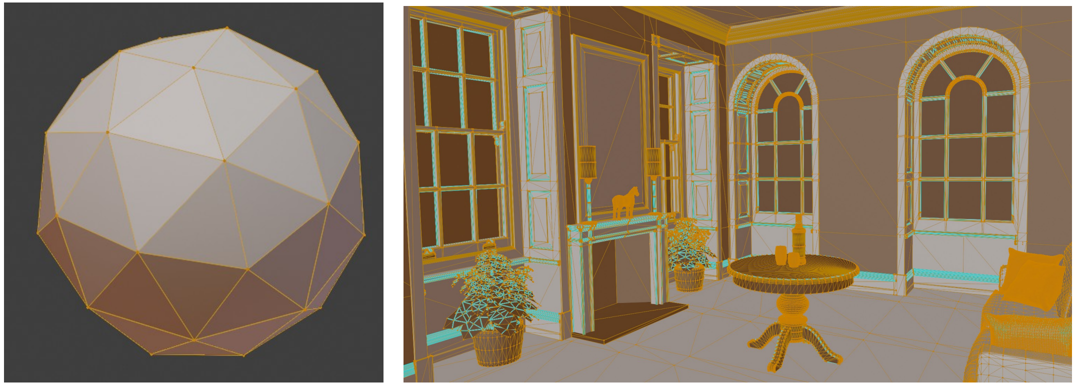
Syllabus

- 3D scene
  - Objects
    - Shape
    - Material
  - Lights
  - Cameras
- Rendering
- Image

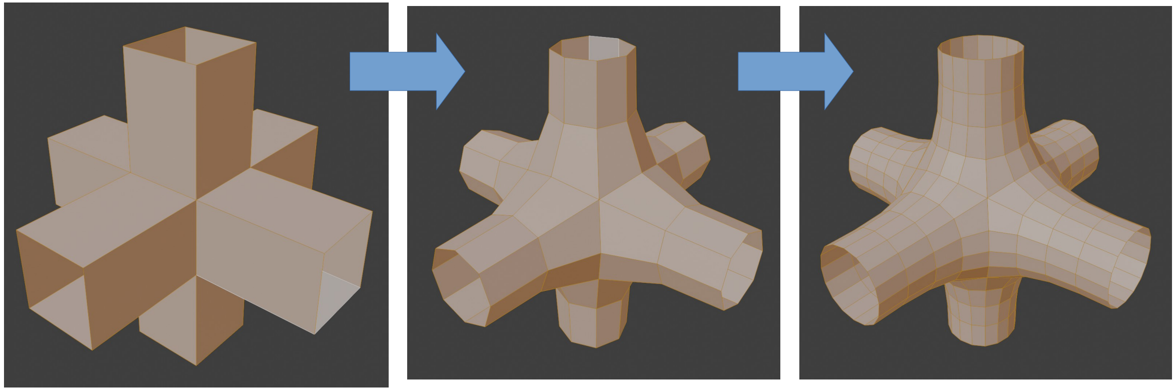
- Object shape
  - Polygon mesh
  - Subdivision surfaces
  - Mesh and rendering
  - Mesh data structures
  - Mesh modeling and acquisition

Recap: shape representations

- Points
  - Point clouds
  - Particle systems
- **Surfaces:**
  - **Polygonal mesh**
  - **Subdivision surfaces**
  - Parametric surfaces
  - Implicit surfaces
- Volumetric objects/solids
  - Voxels
  - Space partitioning data-structures
- High-level structures
  - Scene graph



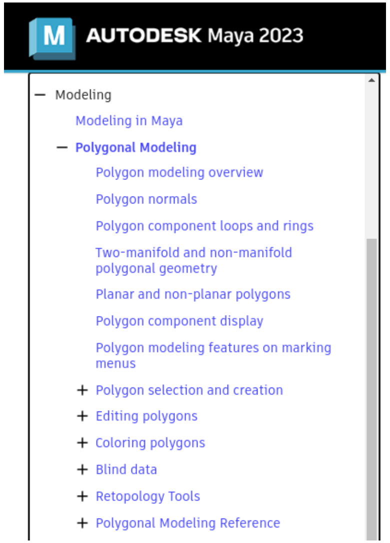
Examples of polygonal mesh

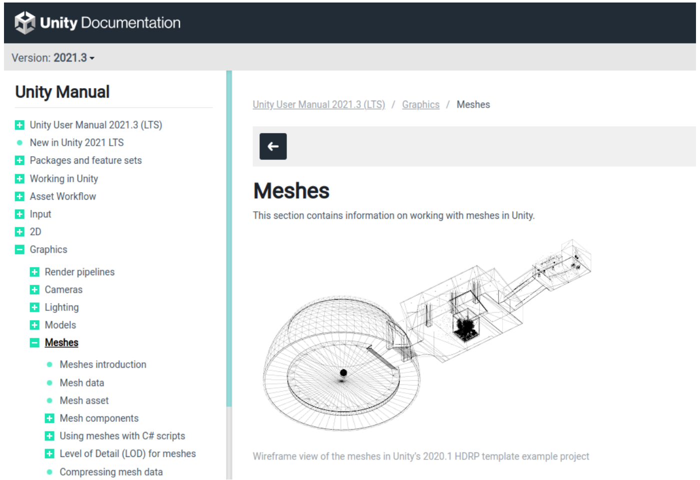


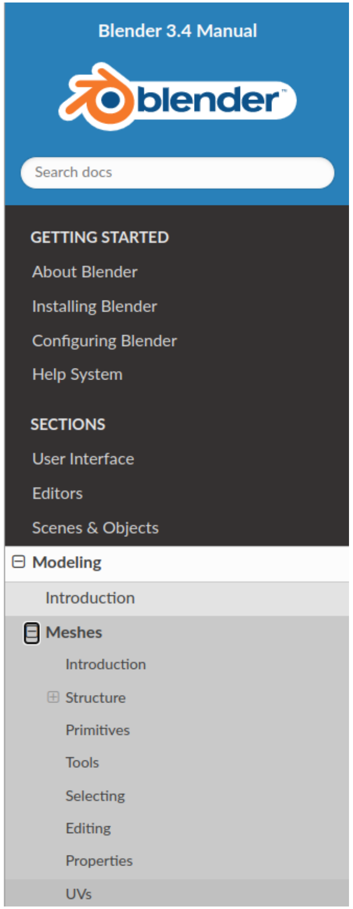
Subdivision surfaces

Polygon meshes

- **Polygon mesh** (shortly mesh) representation is one of the most oldest, popular and widespread geometry representation used in computer graphics
  - Very often, in professional DCC tools or game engines* we can find mesh representation that is used either for modeling or for rendering







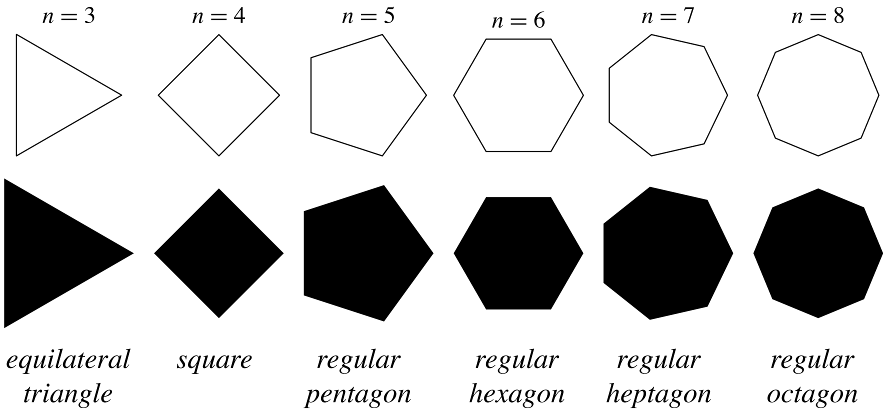
Blender: https://docs.blender.org/manual/en/latest/modeling/meshes/
Maya: https://help.autodesk.com/view/MAYAUL/2023/ENU/?guid=GI
Houdini: https://www.sidefx.com/docs/houdini/nodes/lop/mesh.html
Unity: https://docs.unity3d.com/Manual/class-Mesh.html
Unreal: https://docs.unrealengine.com/4.26/en-US/WorkingWithContent/Types/StaticMeshes/

* Very often, mesh is commonly used for transporting models and scenes from DCC tools to game engines. DCC tools enable modeling using different shape representations, but in a lot of cases, all shapes are transformed to mesh representation and exported to other programs.

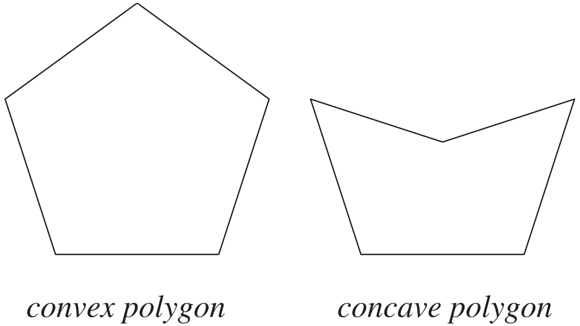
Polygon mesh

Polygon

- Polygon is planar shape defined by connecting array of points.
- **Vertices** (vertex, singular) - individual points
  - In 2D they are defined using two coordinates,e.g., (x,y)
  - In 3D they are defined using three coordinates, e.g., (x,y,z)
- **Edges** - lines connecting two vertices are called.
- **Polygon (face) –** inner part of a connected and closed line of edges
  - Order of connecting vertices (**winding direction): clockwise** or **counterclockwise**
  - Face orientation, defined by **normal**, depends on winding direction



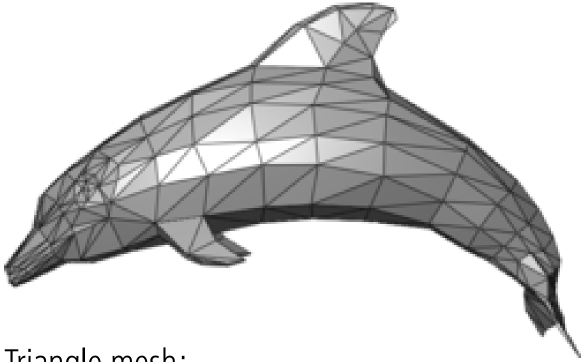
Regular polygons: https://mathworld.wolfram.com/RegularPolygon.html



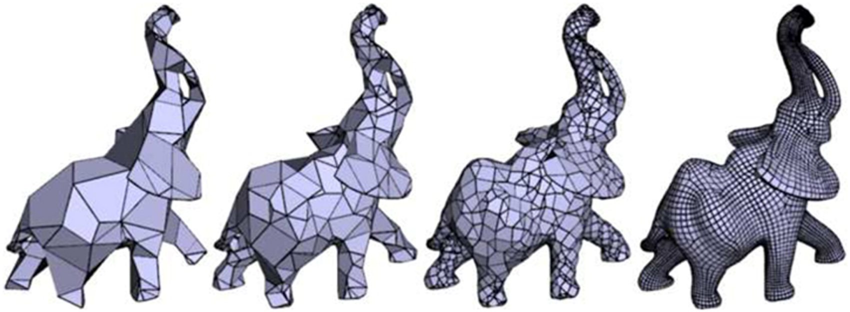
Convex polygon: connection of any two points completely lies inside polygon: https://mathworld.wolfram.com/ConvexPolygon.html

Polygon mesh

- **Polygonal (surface) mesh: boundary representation of an object**
  - 2D surface embedded in 3D space
- Assumption: objects are hollow and can be represented by an approximation of their surface using polygons (faces)
  - Note that this representation is not good for example smoke and volumetric effects.
- Typical polygons: **triangles** and **quads**



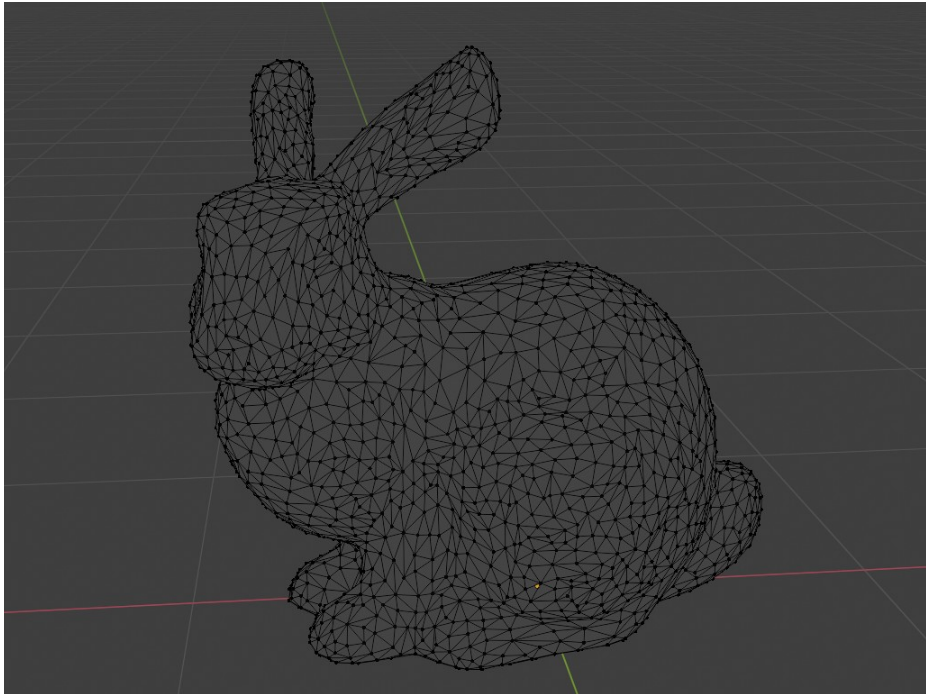
Triangle mesh: https://en.wikipedia.org/wiki/Polygon_mesh

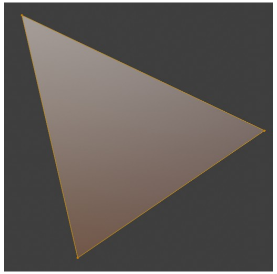


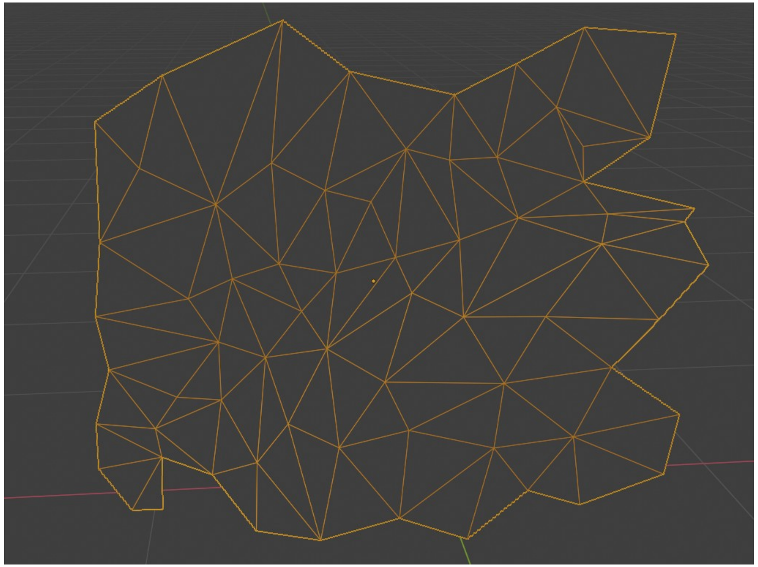
Quad mesh: https://www.sciencedirect.com/science/article/abs/pii/S0097849312000623

Triangle mesh

- Polygon with three vertices → **triangle polygon**.
- **Triangle mesh** consists of many triangles joined along their edges to form a surface





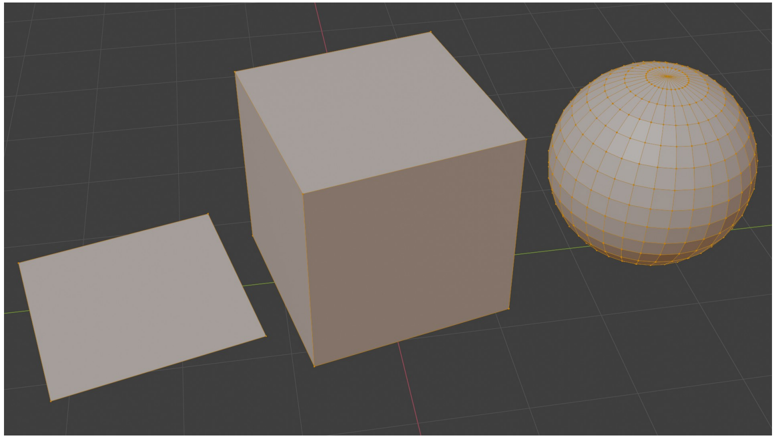


Triangle mesh

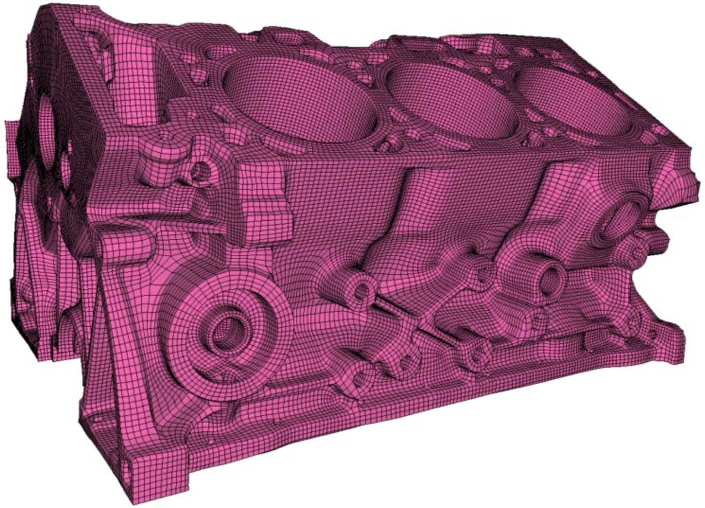
- Triangle mesh is foundational and most widely used data-structure for representation of a shape in graphics
- Triangle is fundamental and simple primitive:
  - All vertices lie in the same plane – **always coplanar**
  - GPU graphics rendering pipeline is optimized for working with triangles
  - Easy to define ray-triangle intersections needed for ray-tracing-based rendering
  - Easy to subdivide in smaller triangles
  - Texture coordinates are easily interpolated across triangle
- Different shape representations used in modeling and acquisition can be transformed to triangle mesh
- Triangle mesh has nice properties:
  - Uniformity: simple operations
    - Subdivision: single triangle is replaced with several smaller triangles. Used for smoothing
    - Simplification: replacing the mesh with the simpler one which has the similar shape (topological or geometrical). Used for level of detail

Quad mesh

- Polygon with four vertices → quadrilateral polygon, shortly **quad**
- **Often used as a modeling primitive**
- Problem: not all quad vertices lie on a plane (not necessary coplanar)
  - More complex ray intersection testing
- For rasterization-based rendering converted to triangle mesh
- Ray-tracing rendering can define ray-plane intersection



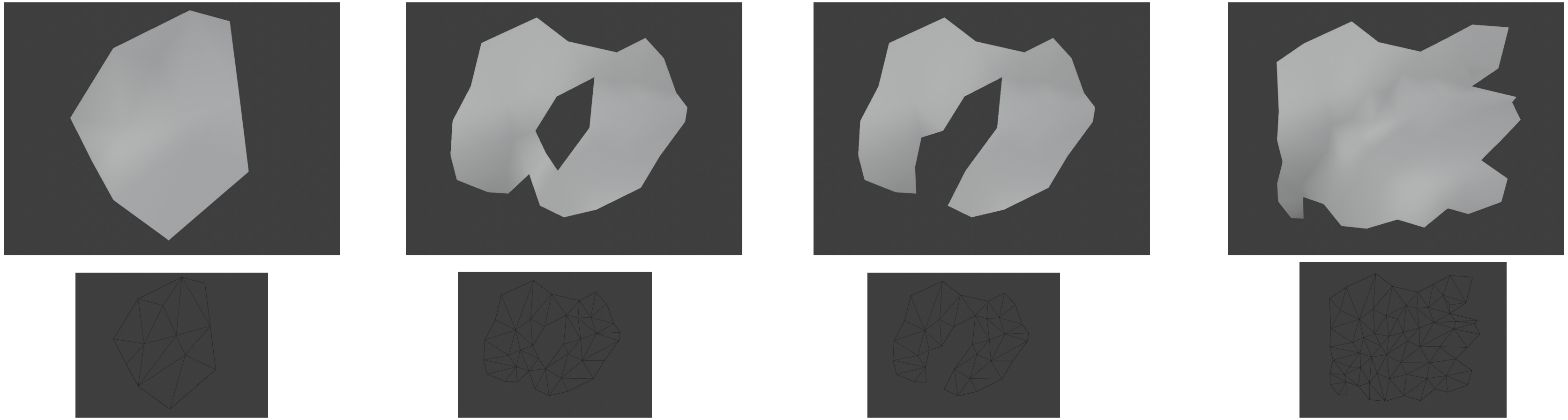
Blender mesh: https://docs.blender.org/manual/en/latest/modeling/meshes/index.html



https://geometryfactory.com/products/igm-quad-meshing/

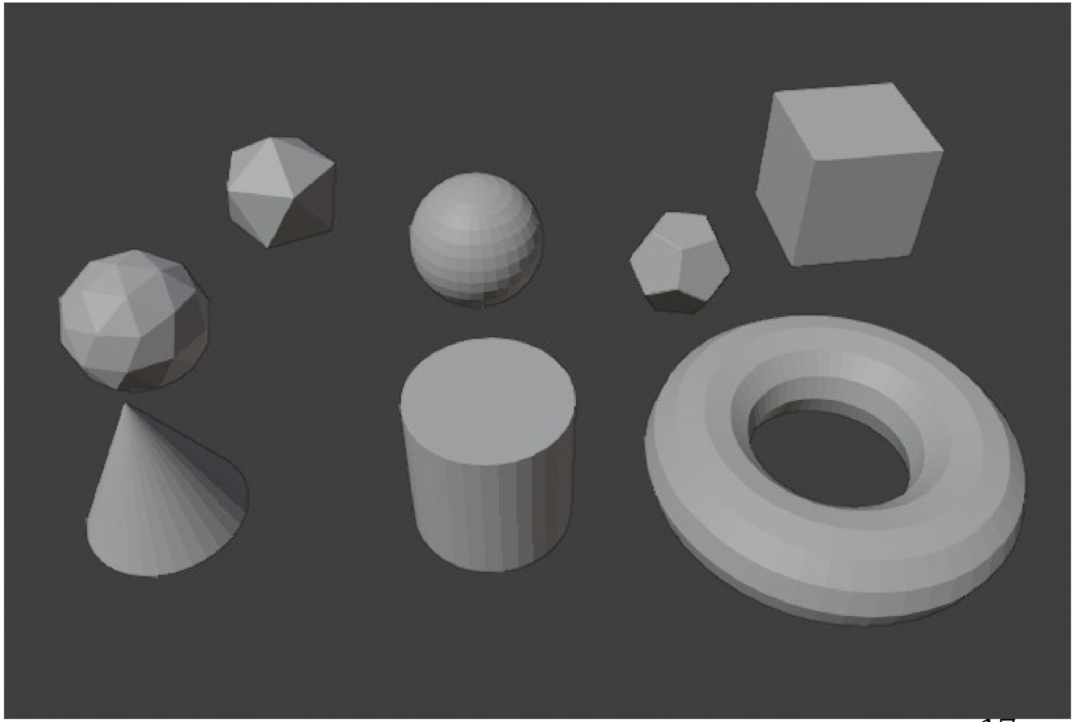
General polygon

- **General** polygon: polygon with more than four vertices
  - Polygons can be convex or concave, and more complex, they may also have holes.
- **Face (atomic element)** of mesh can be any polygon.
  - Good practice: keep atomic elements as simple as possible (so that computation is easier) and combine those atomic elements into more complex shapes, e.g., convex or concave meshes or meshes with holes.



Polygon mesh representation

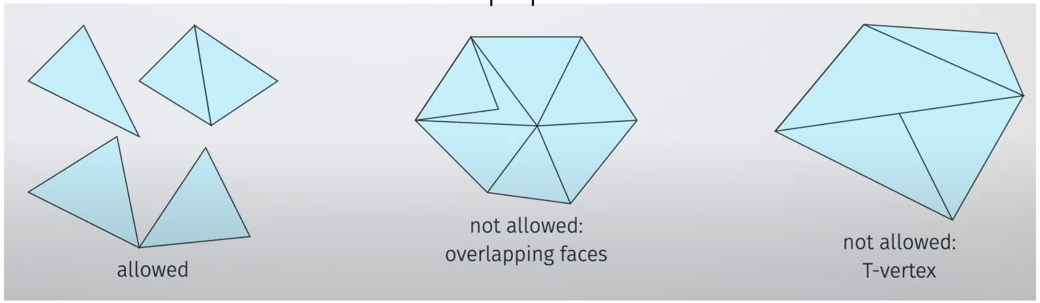
- Basic information needed for representing and storing any polygonal mesh:
  - **Vertex positions** represented by coordinates **(x,y,z) → geometry**
  - **Vertex connectivity → topology**



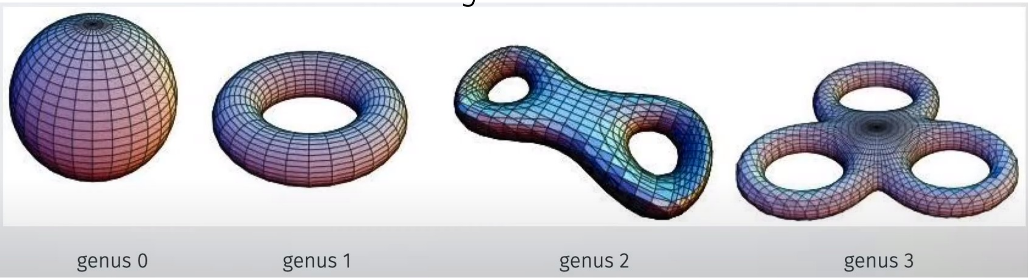
Properties of meshes

- **Manifold mesh**
  - **Mesh is manifold if the intersection of two polygons is empty, a common vertex or a common line**
  - Desired property of many algorithms and general in graphics
- **Mesh genus**
  - Intuition: "number of holes"
  - Important for computational topology
- **Meshes as graphs**
  - Vertices, edges and faces
  - Application of graph algorithms, e.g., shortes edge path between two vertices

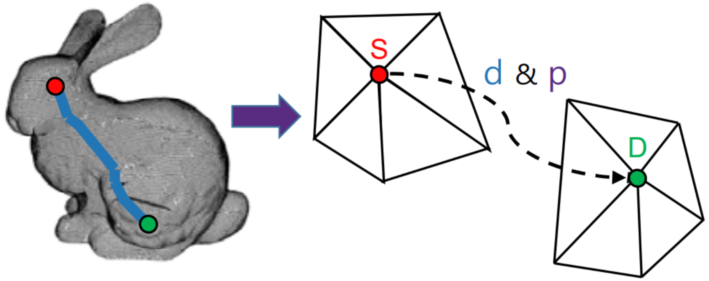
Manifold properties



Mesh genus



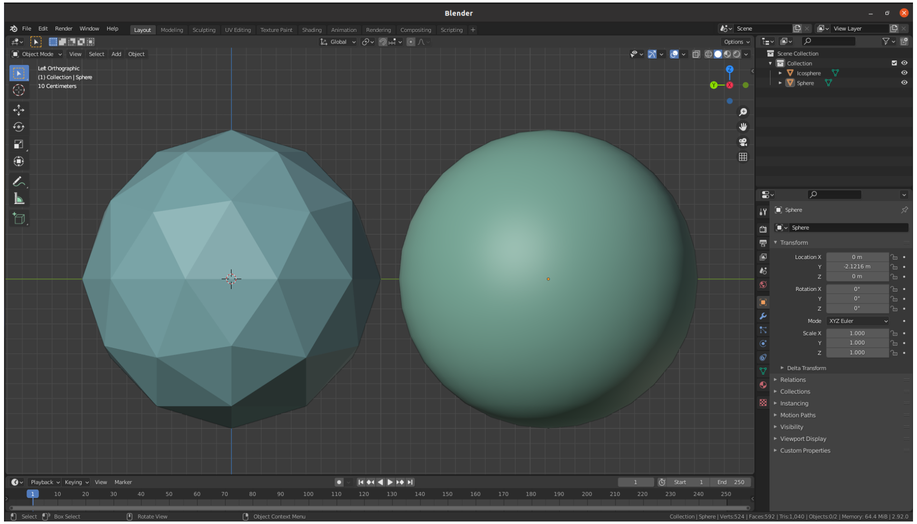
https://www.youtube.com/watch?v=V3Npa0uZYIE&list=PL4TptkuzgxxUVZ-_DiO33kp4_rkoAy1BC&index=6&ab_channel=ChristophGarth

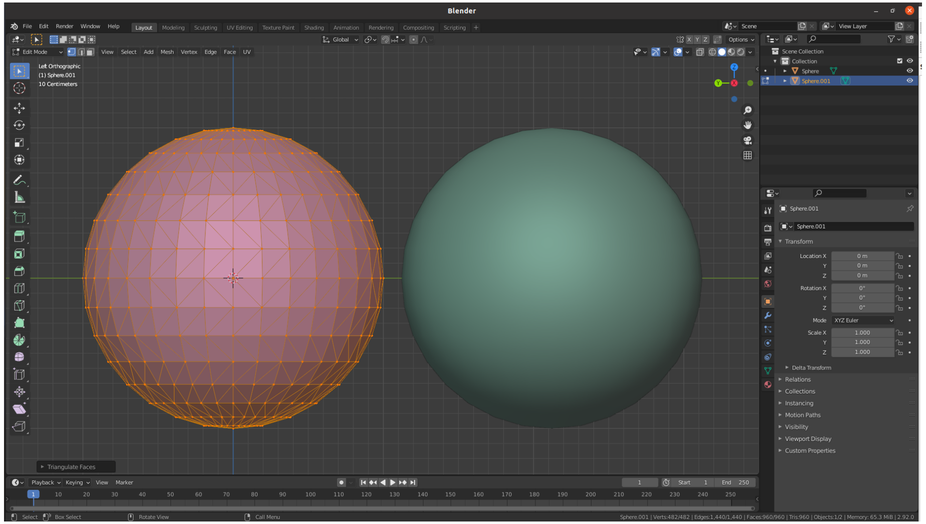


Mesh as graph: http://www.jatit.org/volumes/Vol95No18/12Vol95No18.pdf

Polygonal appoximation

- Example: ideal sphere vs polygon sphere
- **Polygonal approximation**: piece-wise linear (flat) approximation of continuous surface with polygonal mesh

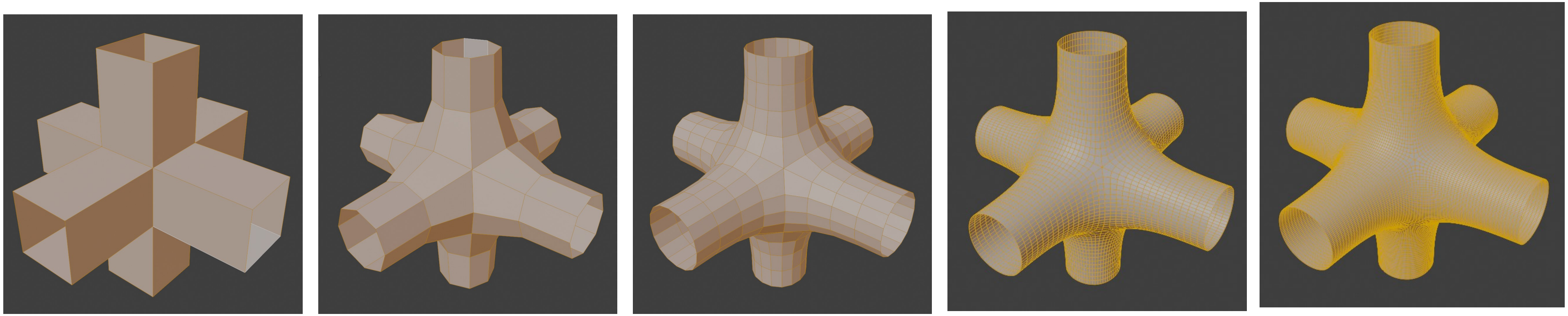




Subdivision surfaces

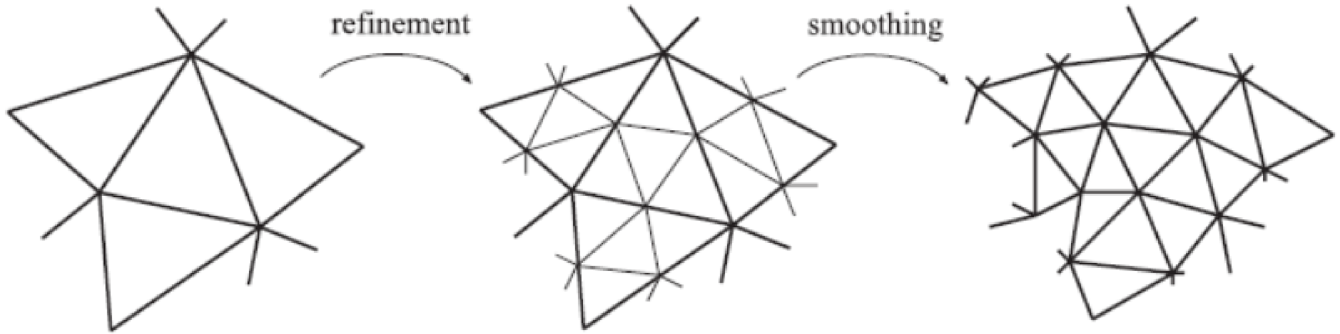
Representing smooth surfaces

- **Subdivision surfaces**: methods for defining smooth and continuous surfaces from meshes with arbitrary topology
  - Infinite level of detail: as many as needed triangles can be generated



Subidivsion surfaces

- Initial input is polygonal mesh: **control mesh (cage)**
- Two-phase process:
  - First, **refinement phase**, creates new vertices and reconnects them to create new smaller polygons
  - Second, **smoothing phase**, computes new positions for some or all vertices in the mesh

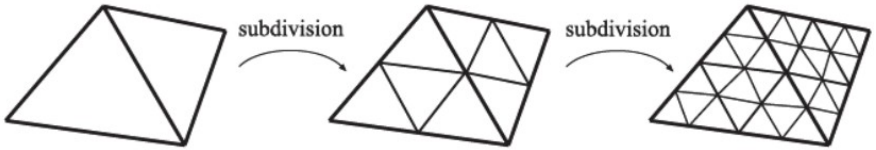


https://www.realtimerendering.com/

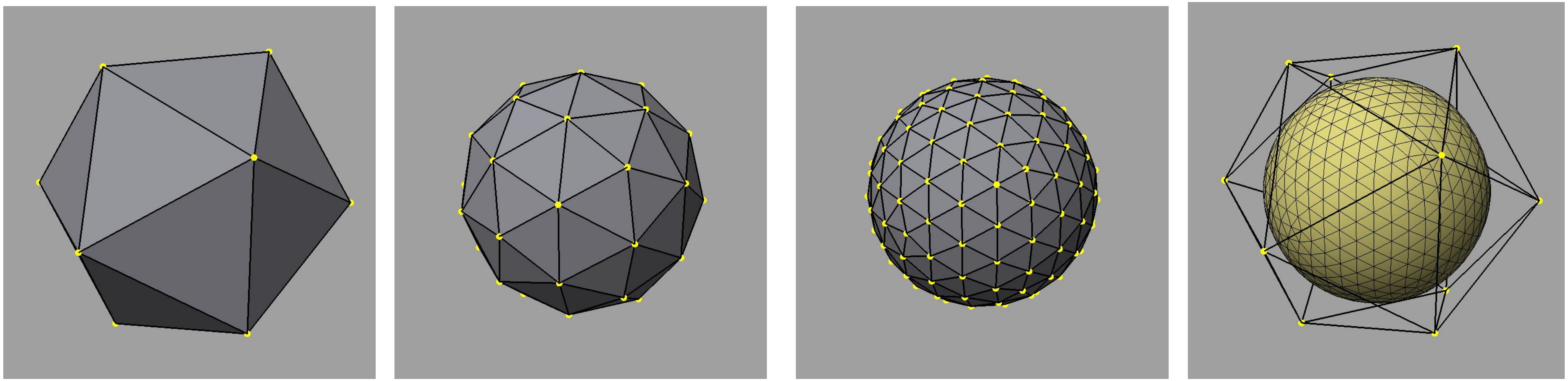
Subidivsion surfaces

- Different **subdivision schemes** are determined by **Refinement and Smoothing**:
  - **Loop subdivision**
  - **Catmull-Clark subdivision**
  - Doo-Sabin subdivision
- Choice of rules gives different surfaces:
  - Level of **continuity**
  - **Approximative or interpolating** surface
- Subdivision schemes characterization:
  - **Stationary vs non-stationary**: always use same rules at every subdivision step or change rules
  - **Uniform vs non-uniform**: use same rules for all vertices and edges or change rules
  - **Triangle vs polygon**: only triangles or arbitrary polygon

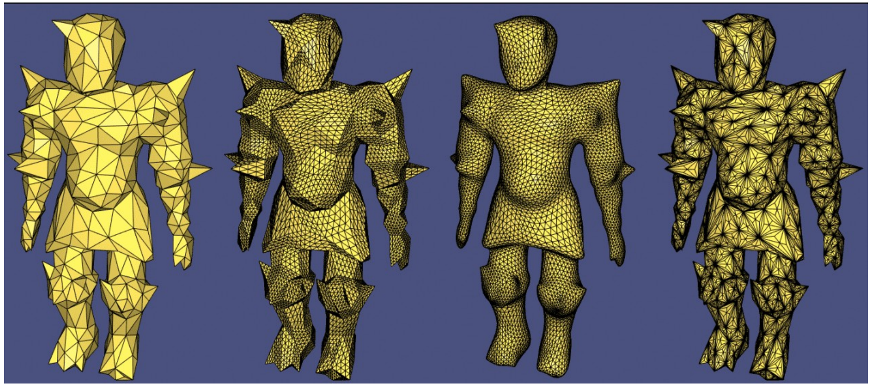
Loop subdivision



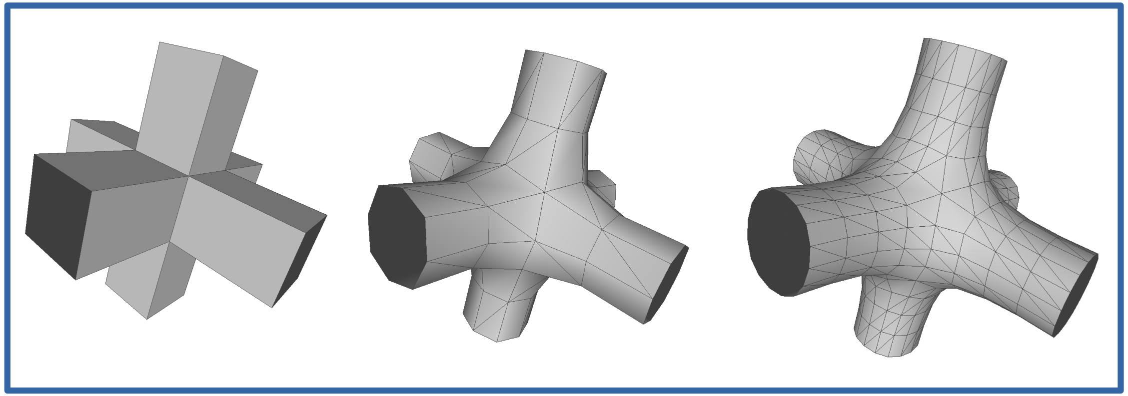
- Stationary, triangle based
- Starts with control mesh, updates each existing vertex and creates new vertex for each edge
  - At each step, triangle is subdivide into four new triangles. After n steps → 4^n triangles
- Resulting surface is approximative → smooth but shrinked
  - To reduce shrinking, more vertices must be defined in control mesh
  - Final surface is contained in convex hull of the original control points



Loop subdivision



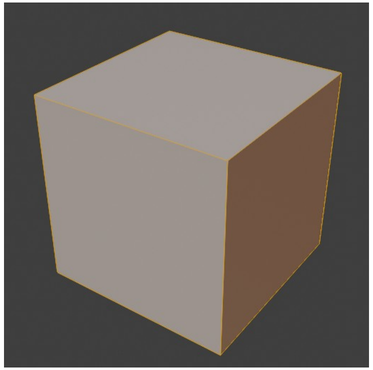
libigl subdivision: https://libigl.github.io/tutorial/ (original, upsample, **loop** and barycentric.

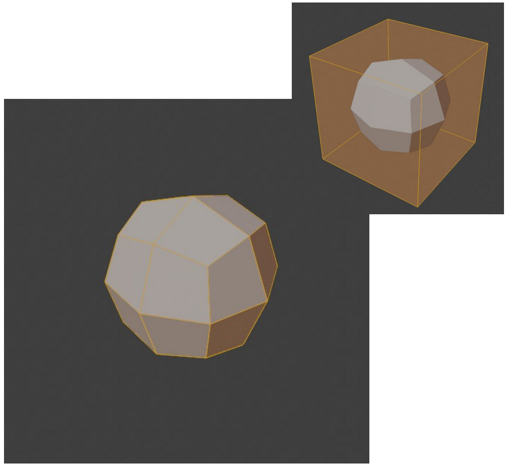


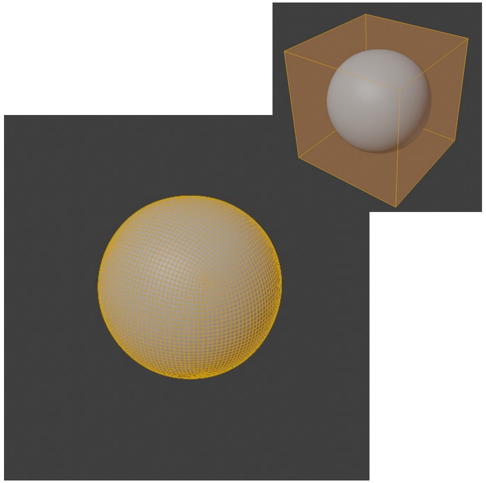
Meshlab loop subdivision:
https://pymeshlab.readthedocs.io/en/latest/filter_list.html#meshing_surface_subdivision_loop
https://github.com/cnr-isti-vclab/meshlab

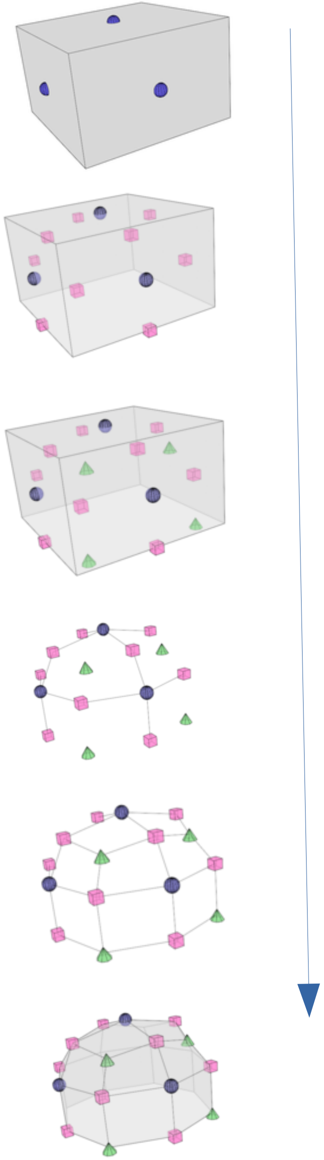
Catmull-Clark

- Method of arbitrary polygons
- Starts with control mesh, recursive process defining rules for*:
  - Computing new face point
  - Computing new edge point
  - Updating vertices, creating new edges and faces









*https://graphics.pixar.com/library/Geri/paper.pdf

https://en.wikipedia.org/wiki/Catmull%E2%80%93Clark_subdivision_surface

Catmull-Clark subdivision

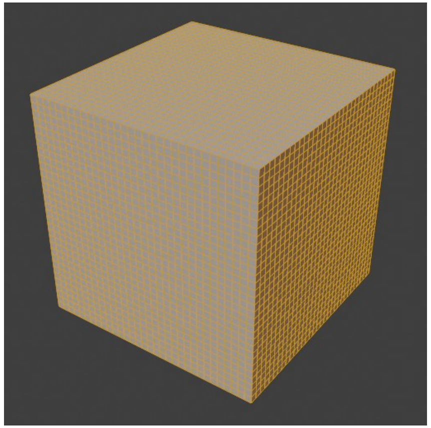
- Most commonly used subdivision surface method
  - Used in all Pixar feature films from Toy Story 2 onward
  - Used from making models for games
- Tends to generate more symmetrical surfaces

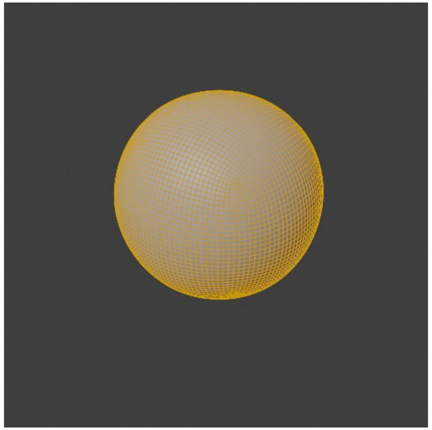
Catmull-Clark Subdivision in Blender (OpenSubdiv backend): https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/subdivision_surface.html

OpenSubdiv: State of the art subdivision library for production: https://graphics.pixar.com/opensubdiv/docs/intro.html

Surface details

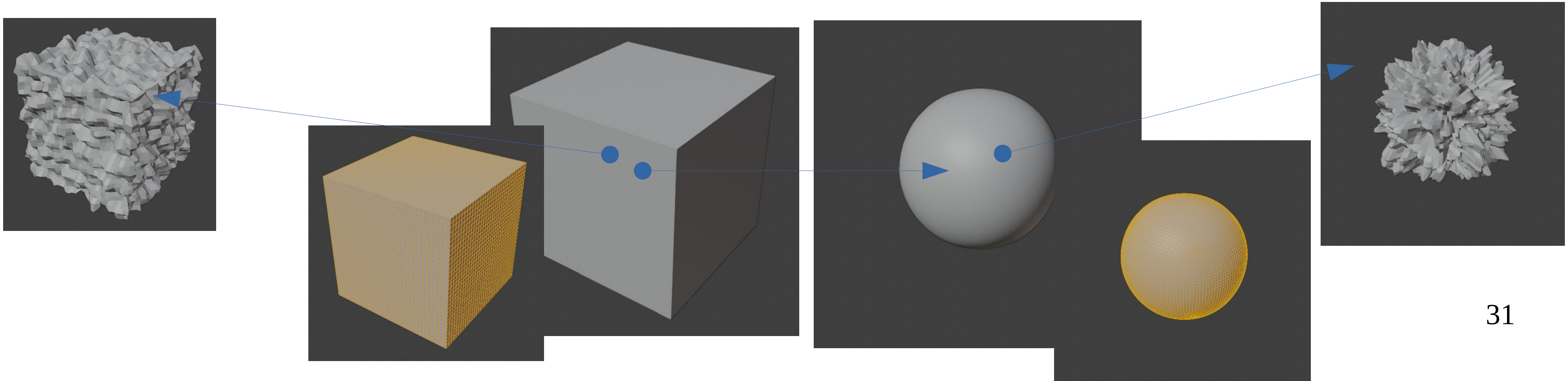
- Flat and curved surfaces are too smooth and thus not realistic while lacking detail
- Detail on surface can be added with:
  - Further geometrical enhancements: displacement
  - Material modeling:
    - Scattering function parameters variation: e.g., color patterns
    - small scale surface variations: surface normal perturbation





Displaced subdivision

- Detail can be added on subdivided surface using **displacement mapping**
- Often displacement mapping moves vertices (v) in direction of surface normal vector (N) for distance (d):
  - `v' = v + d * N`
- Generally, displacement can be done using arbitrary vector (M)
  - `v' = v + M`
- Scalar distance values (d) or vector values (M)used for displacement are described with image or procedural texture

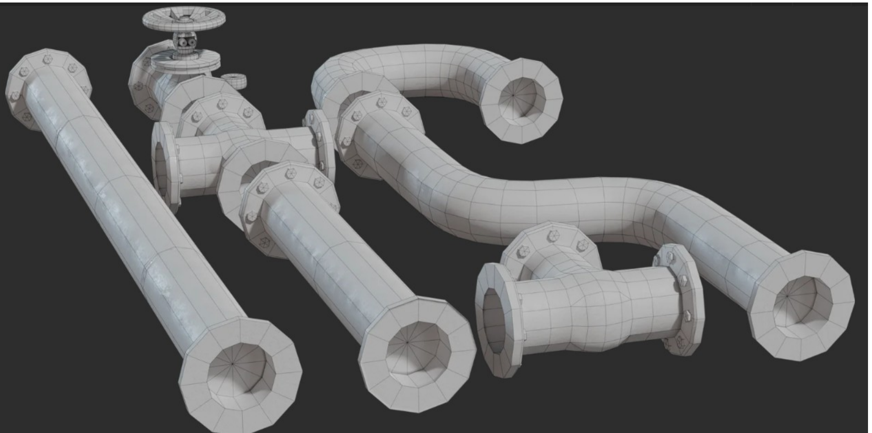


Examples of polygonal meshes

https://polyhaven.com/

Examples of polygonal meshes

https://polyhaven.com/



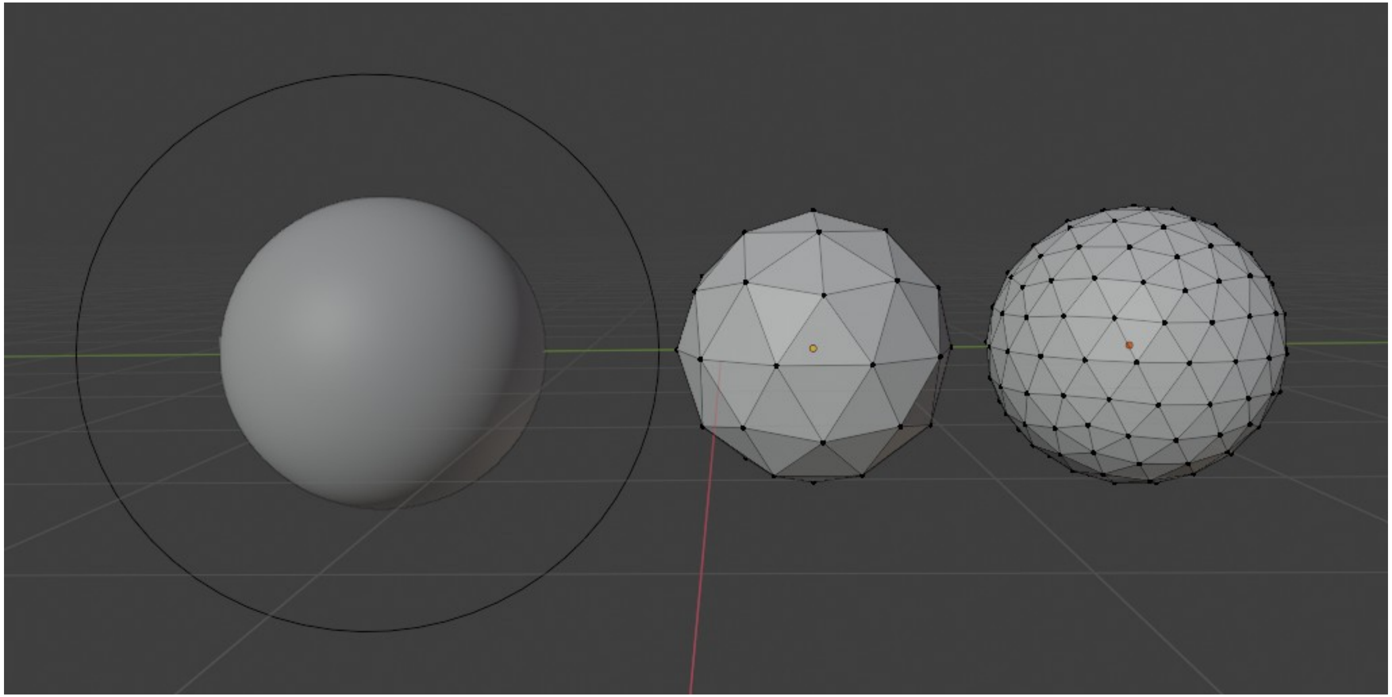
Mesh and rendering

Mesh and rendering

- Rendering uses object shape to solve **visibility problem**
  - What is visible from camera
  - Which surfaces are visible to each other → which might reflect light onto each other (light transport, shading)
- **Rasterization and ray-tracing based rendering are optimized for working with triangulated mesh**
- Any shape representation can be transformed into triangulated mesh using **tessellation**, specifically, **triangulation**:
  - Uniform
  - Adaptive

Uniform tessellation

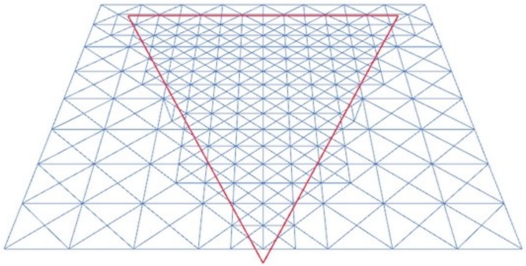
- uniformly sized triangles are created for whole surface.

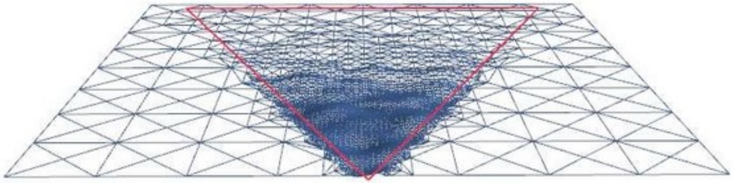


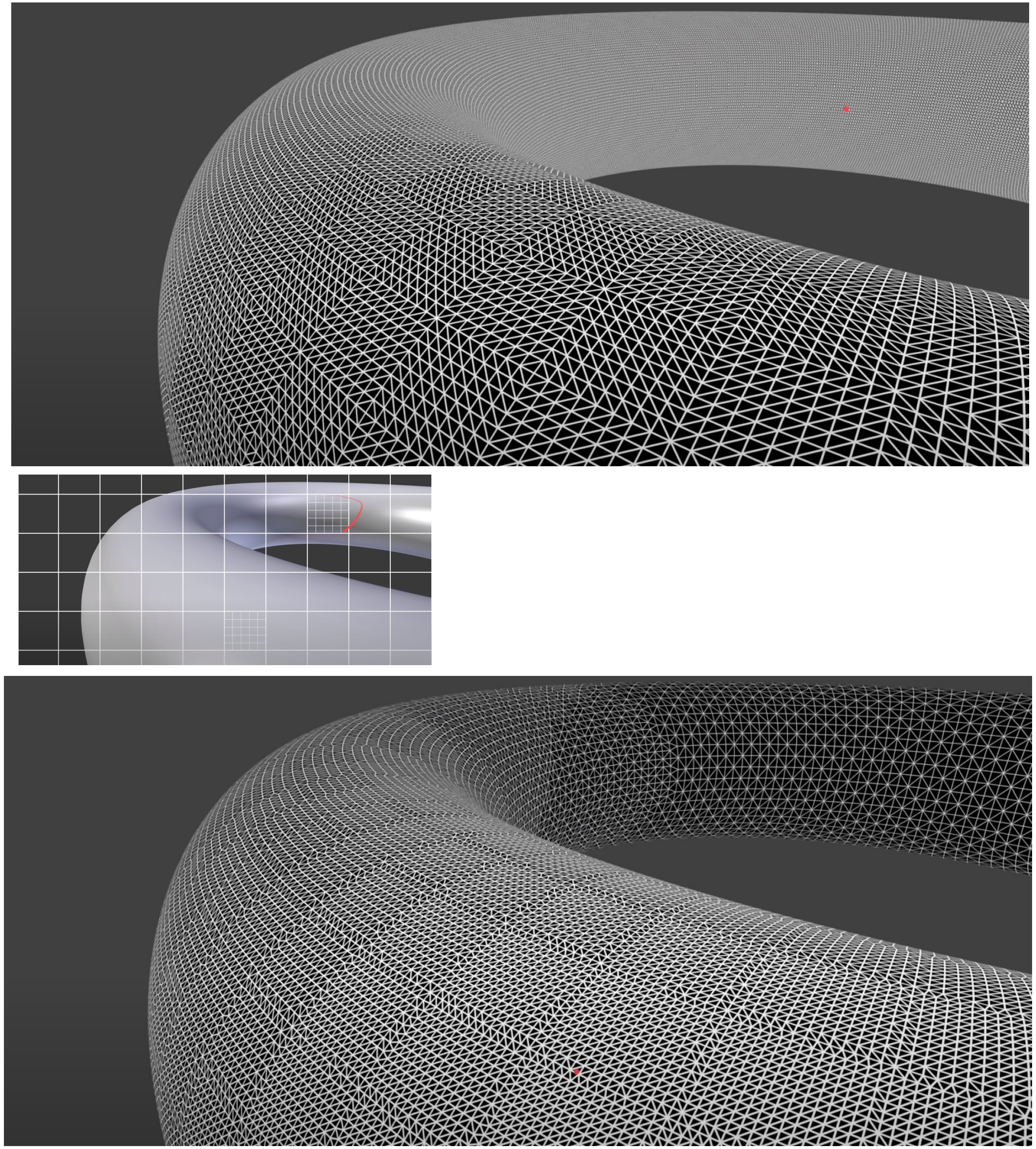
Uniform tessellation of implicit representation of a sphere.

Non-uniform (adaptive) tessellation

- Triangles of varying sizes are created for surface, based on:
  - Distance to surface



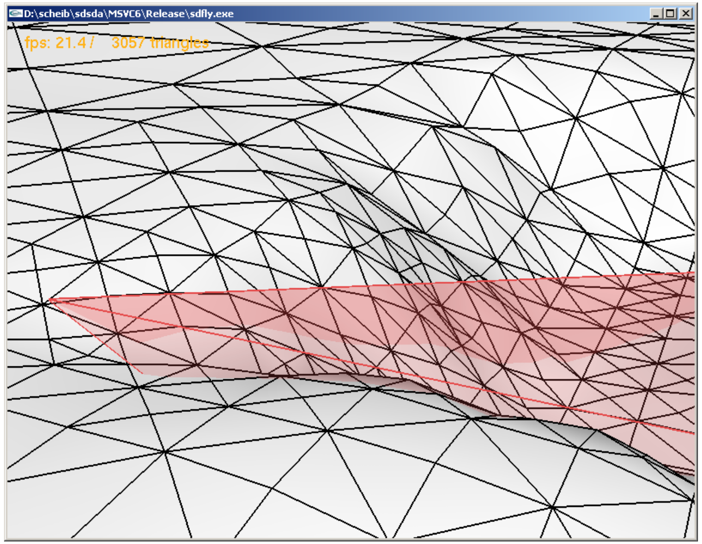


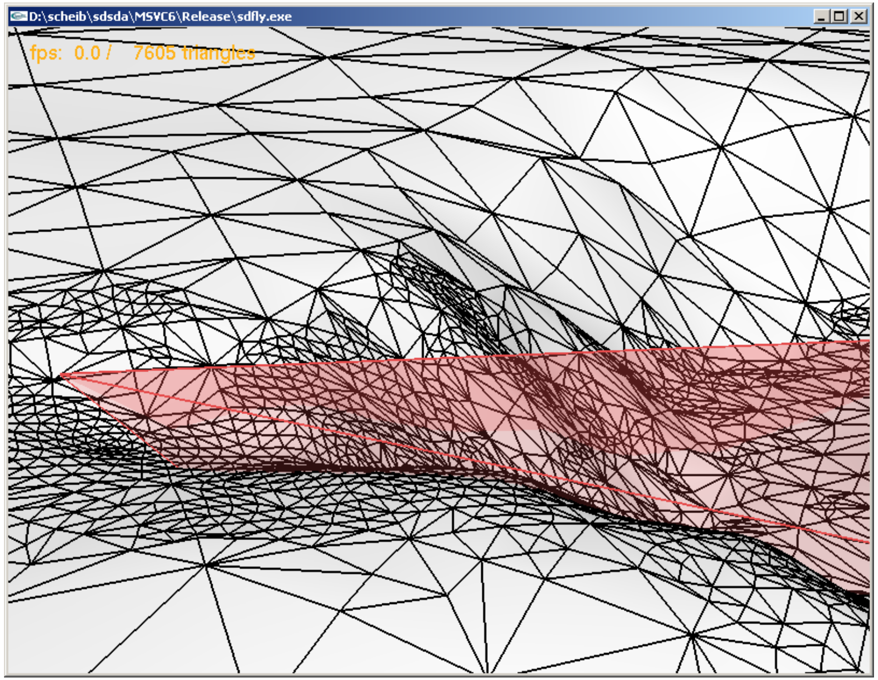


https://www.hindawi.com/journals/tswj/2014/979418/
https://www.blenderguru.com/tutorials/introduction-microdisplacements

Non-uniform (adaptive) tessellation

- triangles of varying sizes are created for surface, based on:
  - Distance to surface
  - Amount of surface details (surface curvature)

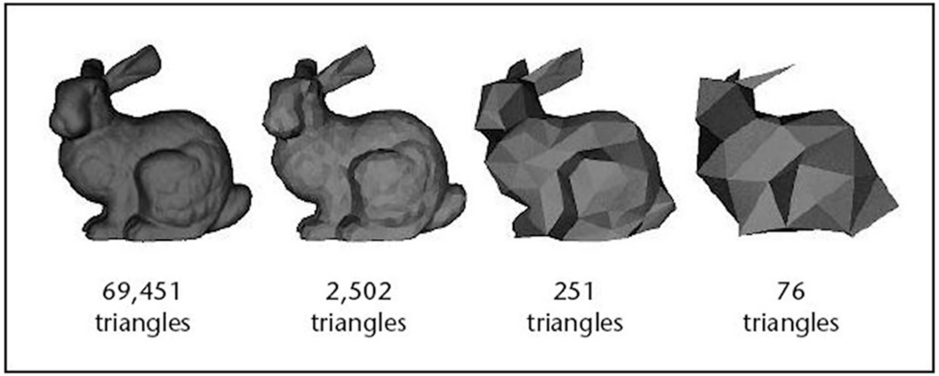




Curved parts of surface are represented with more triangles while flat parts can be represented with less triangles: https://gamma.cs.unc.edu/SDSDA/

Level of detal

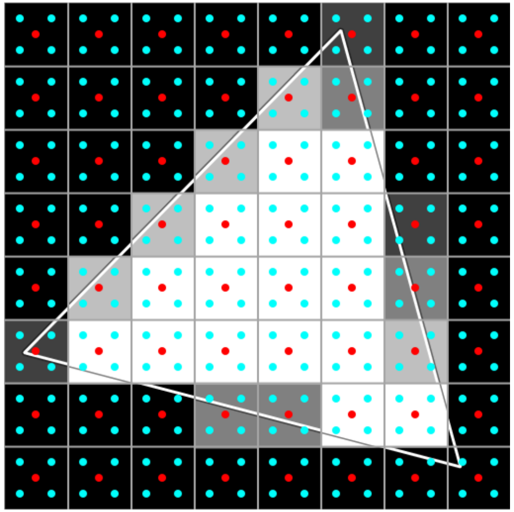
- Depending on camera distance, mesh can contain more or less details
- **Static**: multiple object shapes with different amount of polygons (polygonal resolutions) are stored and used depending on camera distance.
- **Dynamic**: amount of polygons in object shape is computed and used on the fly
  - https://www.gamedeveloper.com/programming/real-time-dynamic-level-of-detail-terrain-rendering-with-roam



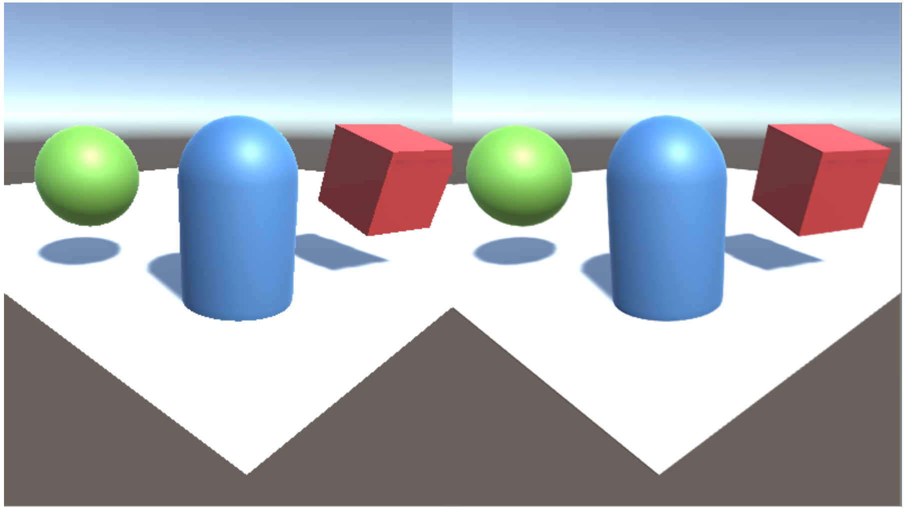
Static level of detail: https://3dstudio.co/3d-lod-level-of-detail/

Triangulated mesh and rendering

- Triangle surface is continuous, to be displayed on raster image discretization must be performed
  - Due to discretization, aliasing problems might appear
  - Solution: multiple rays/samples per pixel



https://www.beyond3d.com/content/articles/122/4



https://docs.unity3d.com/560/Documentation/Manual/PostProcessing-Antialiasing.html

Vertex attributes

- Besides vertex positions (geometry) and connectivity information (topology), additional information is used for rendering.
- This information is stored per vertex - **vertex attributes***:
  - Normal
  - Texture coordinate
  - Color
  - Any kind of information that can be encoded and used for rendering: weights, temperature, etc.

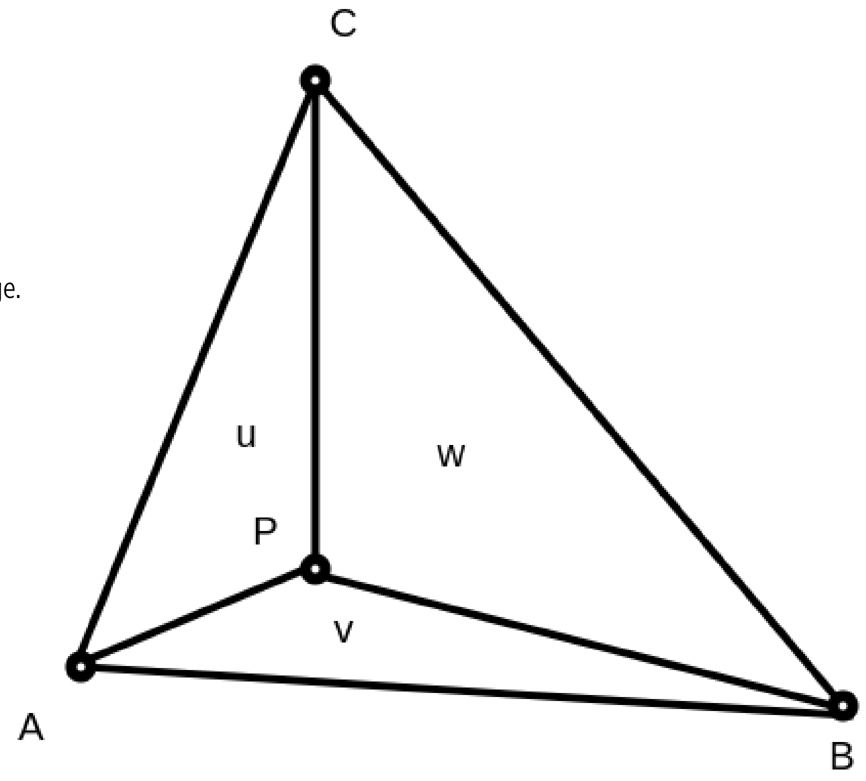
* Those are in general called primitive variables. Primitive is generic term in computer graphics which describes object which is understandable by program.

Vertex attributes

- Vertex attributes are, during rendering (shading), interpolated across triangle or quad
  - Example ray-tracing: ray hitting inside of a polygon
  - Example rasterization: color of pixel inside polygon
- Interpolation depends on polygon types:
  - **Triangles: barycentric interpolation**
  - **Quads: bilinear interpolation**

Barycentric interpolation

- Express position of any point located on triangle with three scalars.
  - `P = uA + vB + wC`, where `A,B,C` are triangle vertices
  - `u,v,w` are barycentric coordinates
    - `u + v + w = 1` and
    - `0 <= u,v,w <=1`, otherwise P is outside of triangle. If equal then P is on triangle edge.
- Areal coordinates: `u,v,w` are proportional to area of sub-triangles defined by P
  - `u = Area(Triangle(CAP)) / Area(Triangle(ABC))`
  - `v = Area(Triangle(ABP)) / Area(Triangle(ABC))`
  - `w = Area(Triangle(BCP)) / Area(Triangle(ABC))`
  - `Area(Triangle(ABC)) = ||(B−A)x(C−A)||/2`
- Barycentric coordinates are very useful for interpolating vertex data across triangle surface, e.g., normals, colors, etc.
  - We know P and A,B,C. Thus, calculate, u,v and w and use this factors for interpolating data

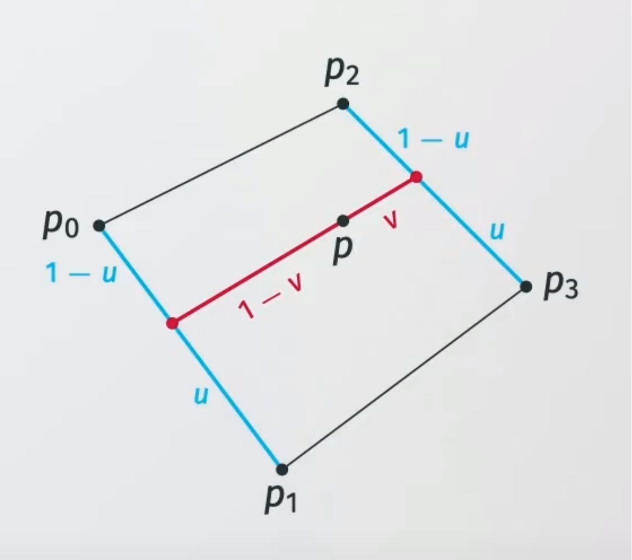


Bilinear interpolation

- Any point on quad can be found with bilinear interpolation
  - Thus, and value stored per vertex of quad can be interpolated in the same way

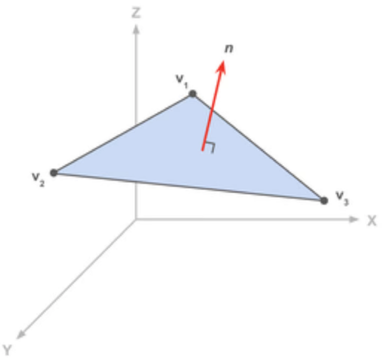
$$f(p) = (1-v)\left((1-u)f(p_0) + uf(p_1)\right) + v\left((1-u)f(p_2) + uf(p_3)\right)$$

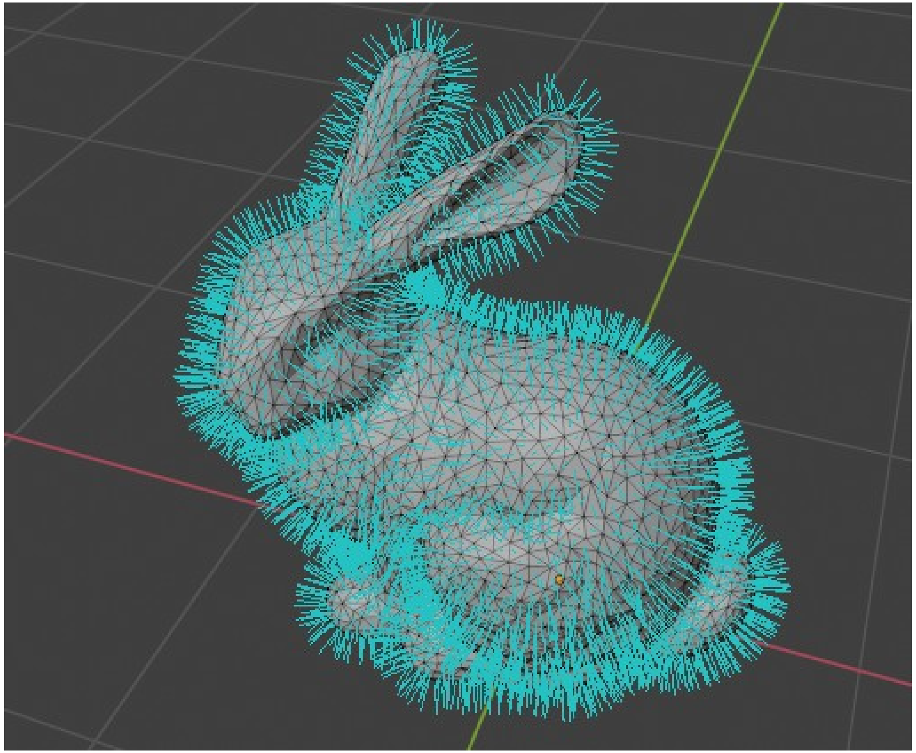
$$u, v \in [0,1]$$



Normals

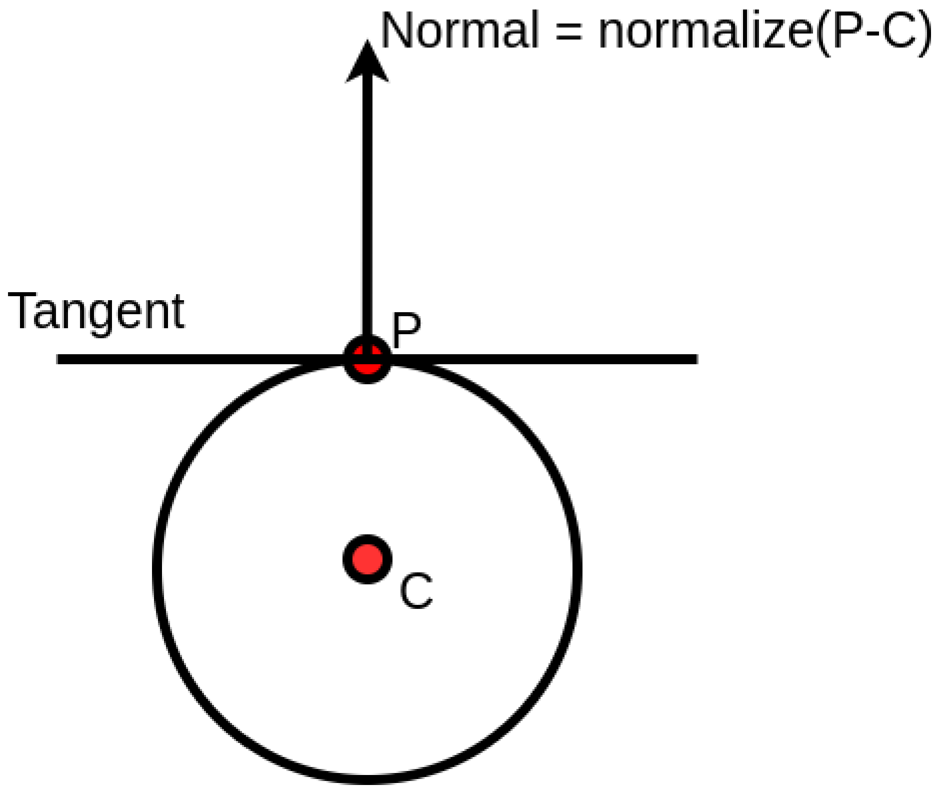
- Orientation of surface in each point is determined by normal vector.
- Normal vector is **core information for rendering (shading) and modeling**.
- Mesh normal can be defined per:
  - face
  - per vertex





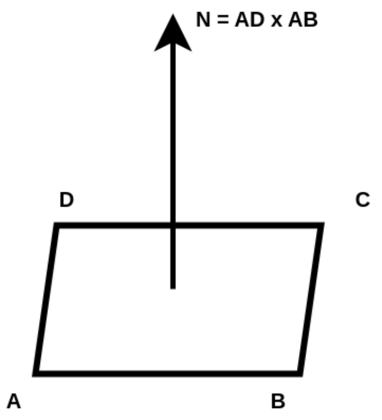
Computing normals

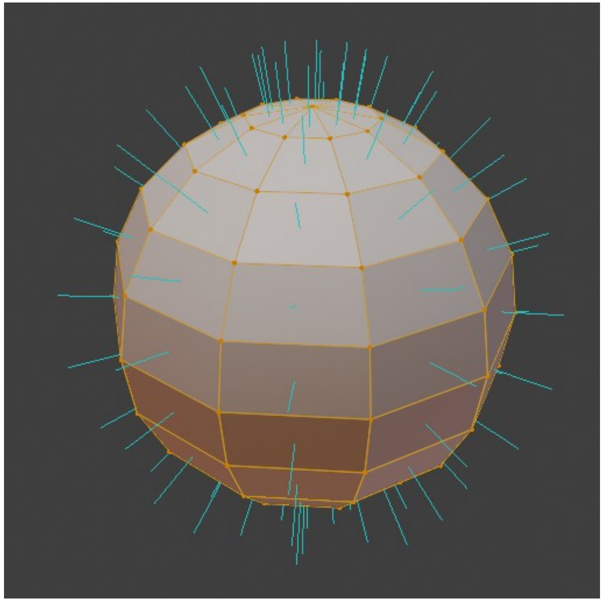
- Normal in surface point is **vector perpendicular to tangent** in that surface point
- Computation of normals depends on shape representation
  - Sphere normal
  - Mesh face normals can be calculated using triangle or quad polygon edges.

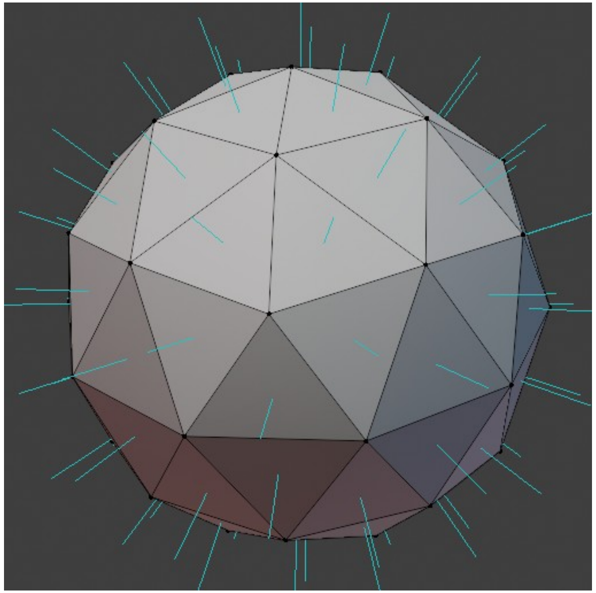


Normals: per face

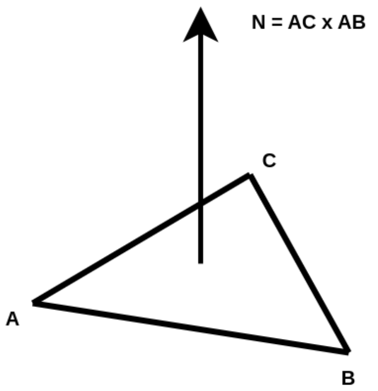
- Cross product between triangle or quad edges
- Winding order of vertices defines orientation of normal
- For right-handed coordinate system, polygons with counter clockwise winding will be front facing





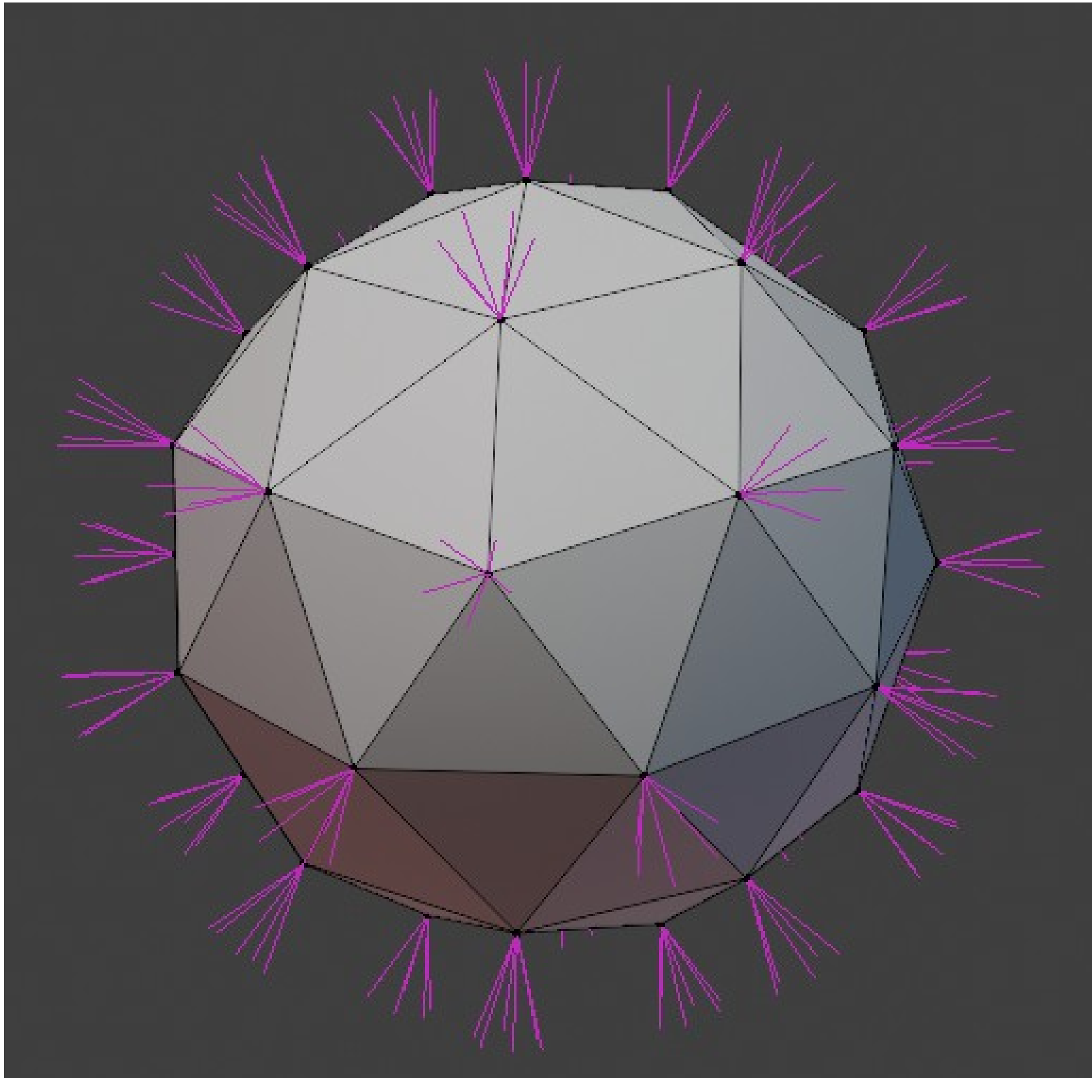


```
N = (B-A).crossProduct(C-A);
```



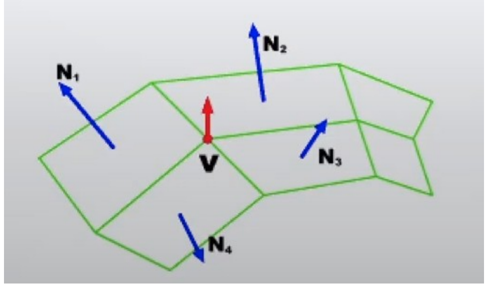
Normals: per vertex

- Multiple normals, one for each face, can be stored per vertex.

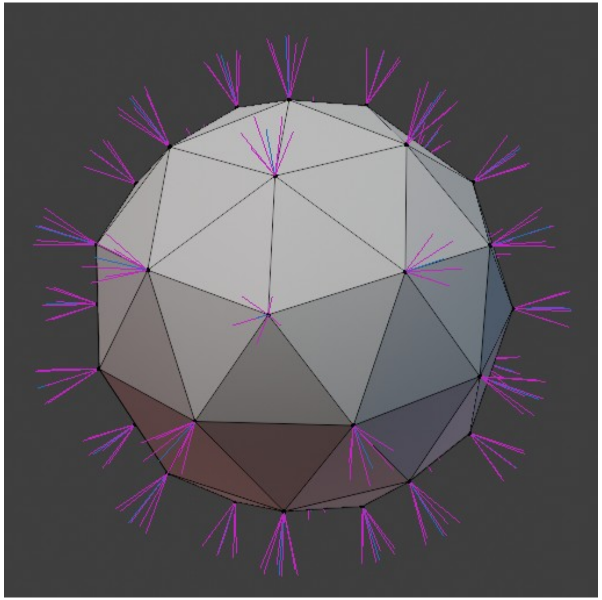


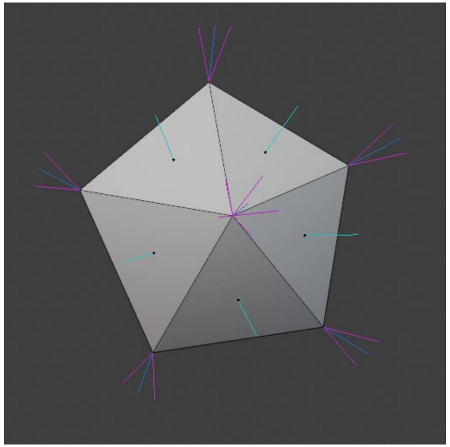
Normals: per vertex

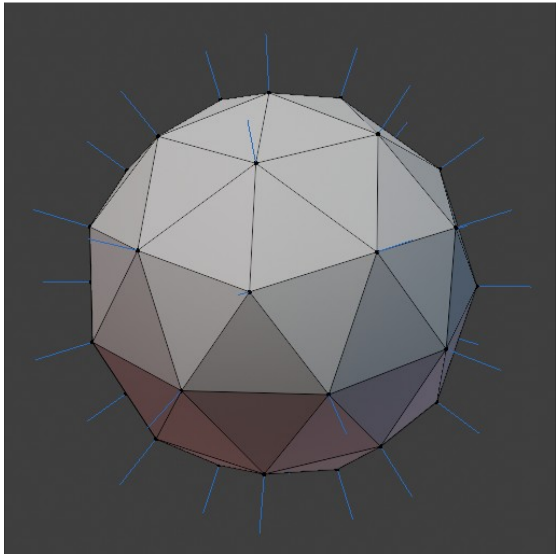
- One normal per vertex is calculated as:
  - Calculate normal for each face connected to vertex
  - Average calculated face normals



$$\vec{N}_v = \frac{\sum_{i=1}^{k} \vec{N}_i}{\left\|\sum_{i=1}^{k} \vec{N}_v\right\|}$$

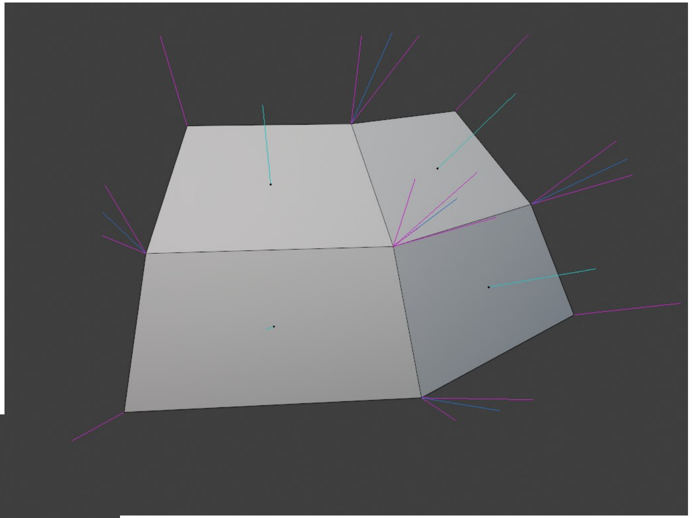


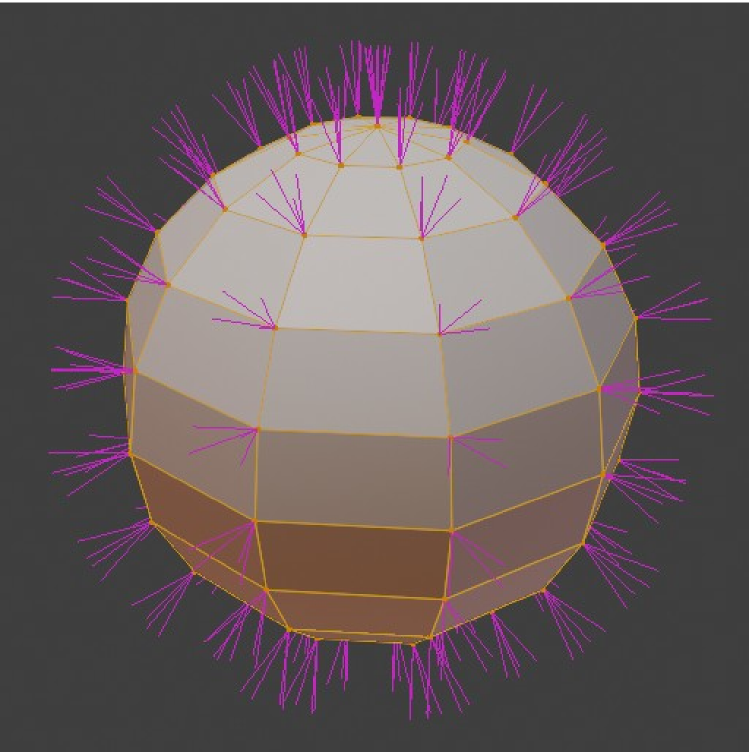


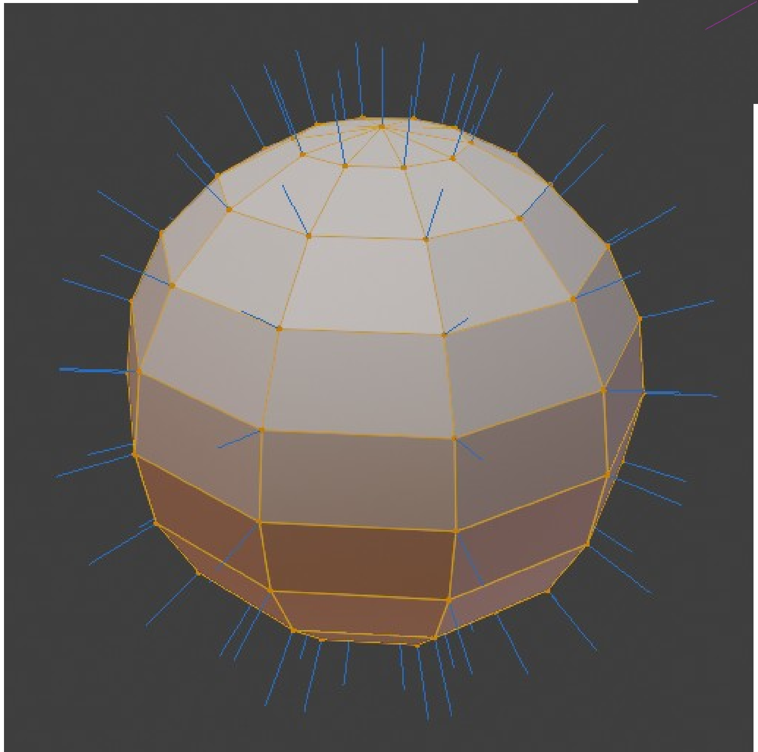


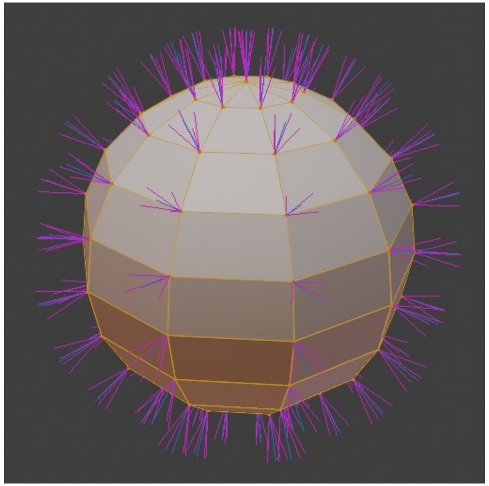
Normals: per vertex

- One normal for each face per vertex vs one normal per vertex



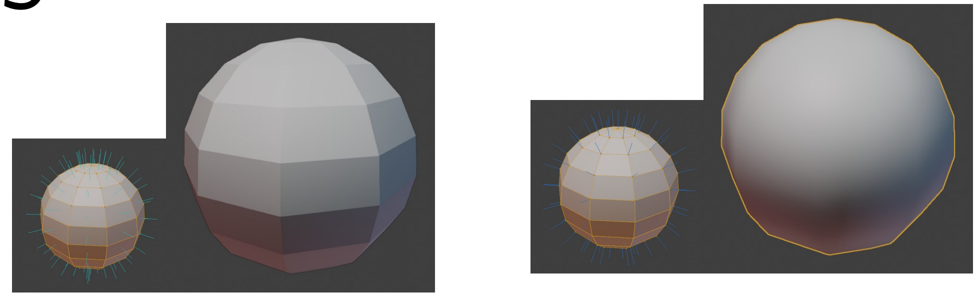


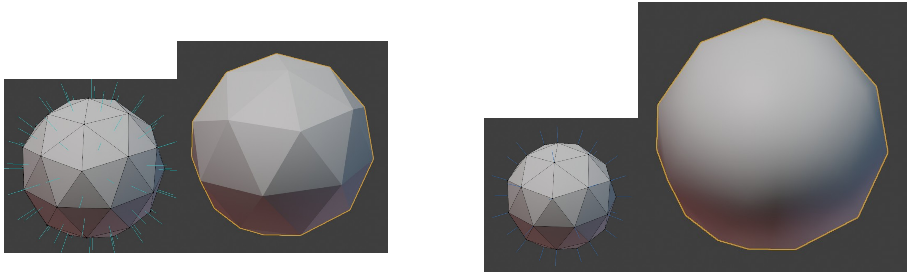




Using normals: shading

- Appearance of object depends both on shape and material.
  - Normal is main shape information used in shading: evaluation of scattering function depends on normal
- **Flat shading:** evaluation of scattering model using per-face normals (no interpolation)
- **Gouraud (smooth) shading**: evaluation of scattering model at vertices and interpolating resulting color (e.g., barycentric interpolation)
  - Problem: unnatural highlights for larger polygons since largest highlights appear at vertices of mesh and less on polygon during interpolation
  - "Vertex" light calculation
- **Phong (smooth) shading**
  - Normals per vertex are linearly interpolated over triangles (using barycentric coordinates) and then scattering model is evaluated.
  - "Per-pixel" light calculation
  - Note: phong shading vs phong lighting model





Smooth shading:

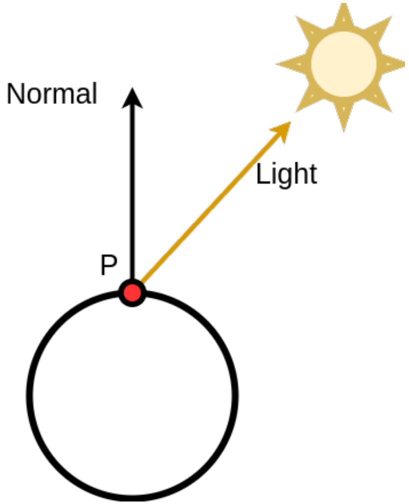
- Geometry is not changed: note faceted silhouette
- Illusion of smooth surface is created during shading

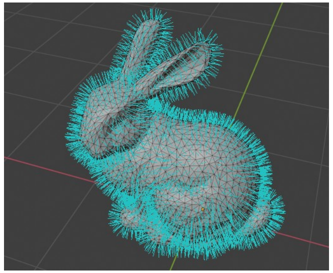
Flat vs smooth shading

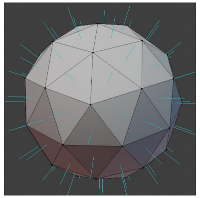
 

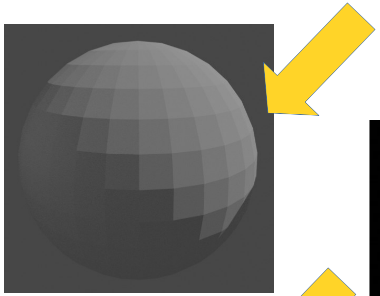
Using normals: shading

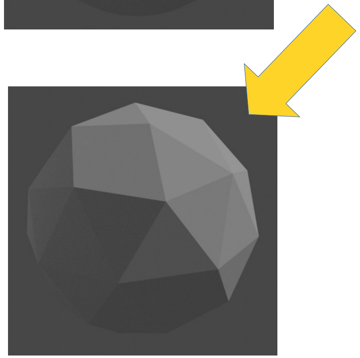
- Amount of light incident on surface depends on cosine of the angle between light direction and surface normal
  - Lambert's Cosine law
  - $\cos(\theta) = \mathrm{dot}(N, L)$

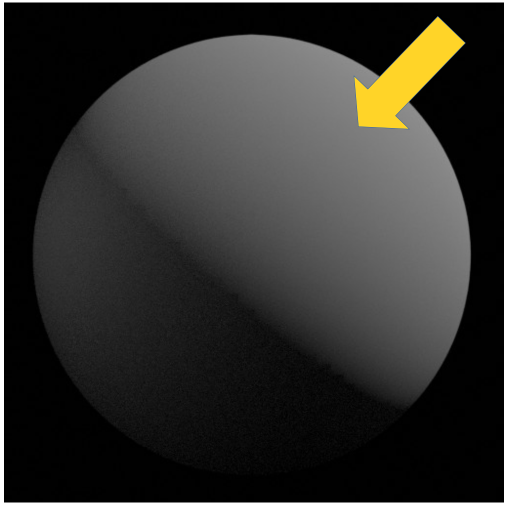


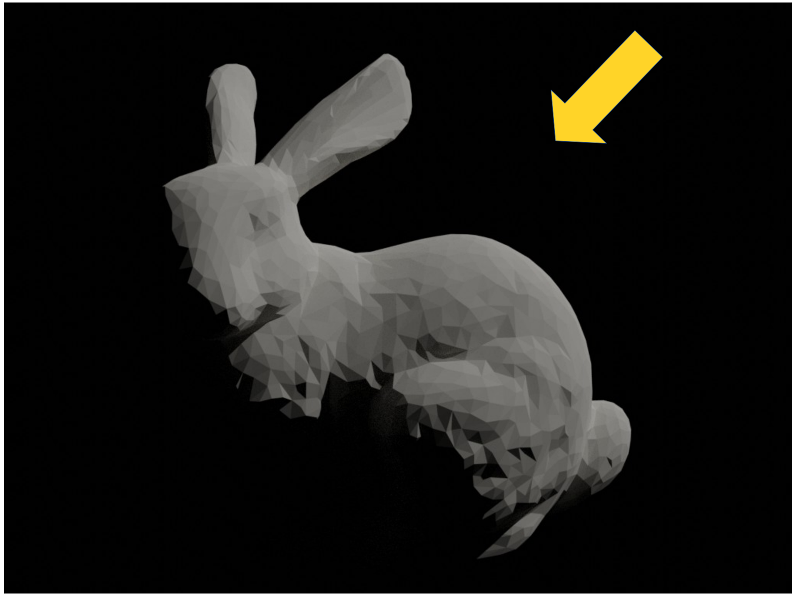






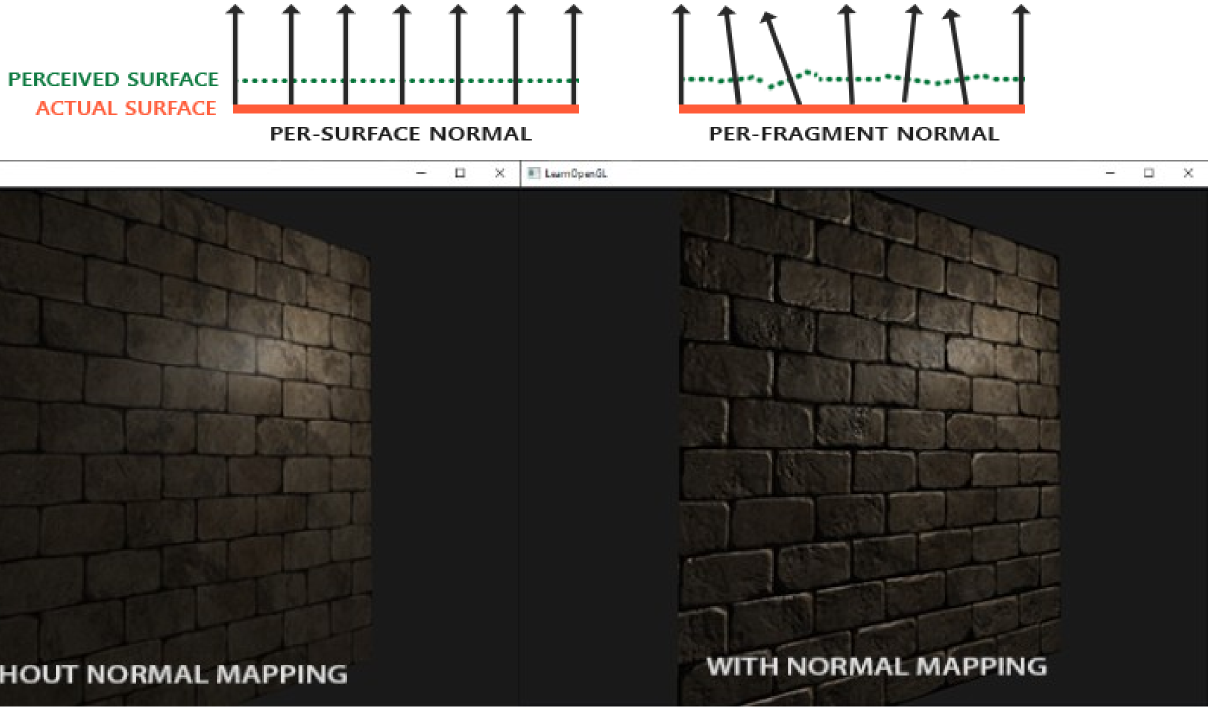






Using normals: surface details

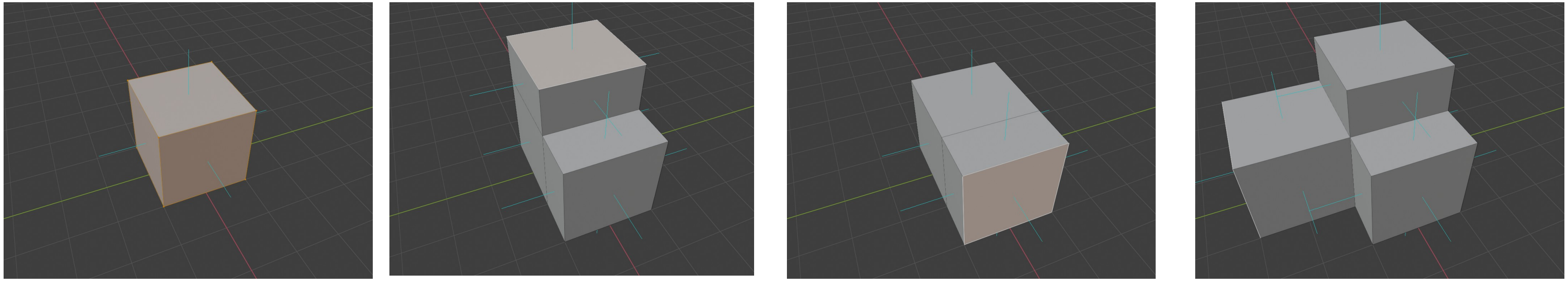
- Face or vertex normals result in smooth and too perfect surface
- These normals, during rendering, can be perturbed for each point on surface which give appearance of surface details
  - Bump/normal mapping

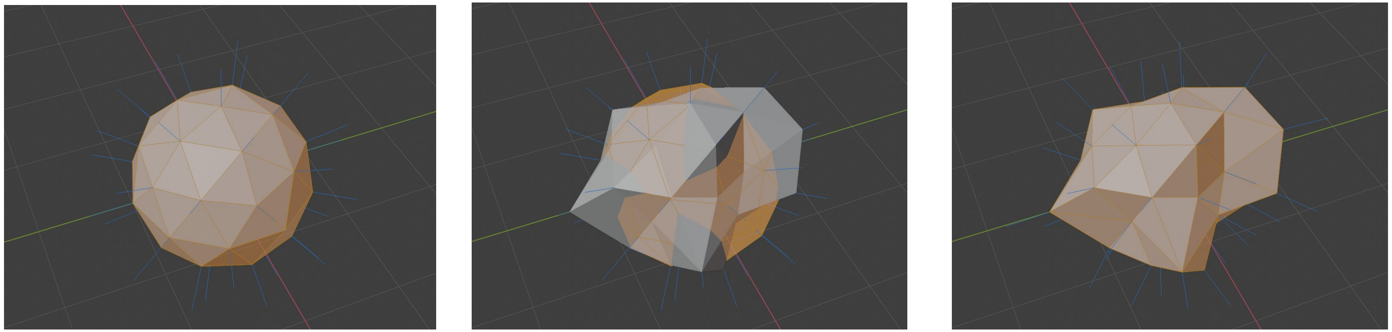


https://learnopengl.com/Advanced-Lighting/Normal-Mapping

Using normals: geometric manipulation

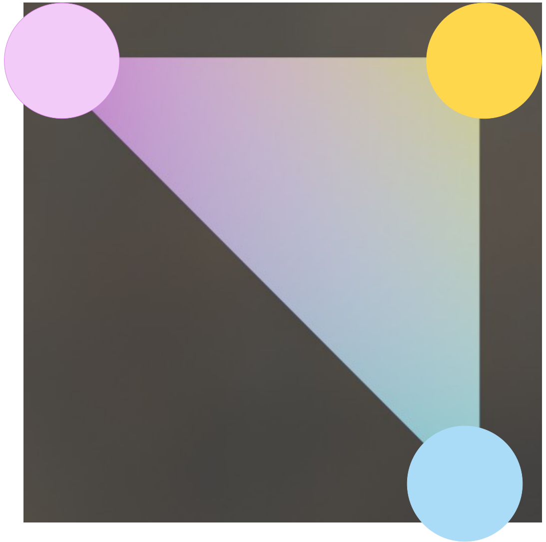
- Example: extrusion
  - Basic modeling operations which moves mesh face in direction of normal
- Example: procedural modeling
  - Using normal direction for **displacing mesh vertices**

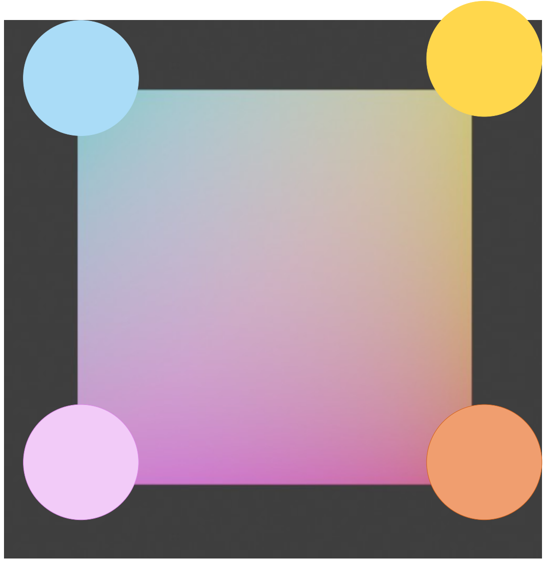


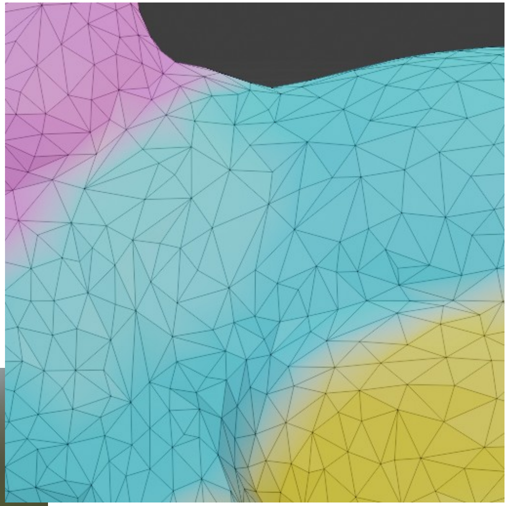


Additional mesh information: vertex colors

- Similarly as normals are defined per vertex and interpolated over triangle for shading purposes, the same can be done with color
- Simplest way to introduce variation over object surface – a form of **texture**
- This method works fine for meshes with finer structure → more colors can be assigned to more vertices.
  - When it is not possible to have fine mesh, then texture is used.



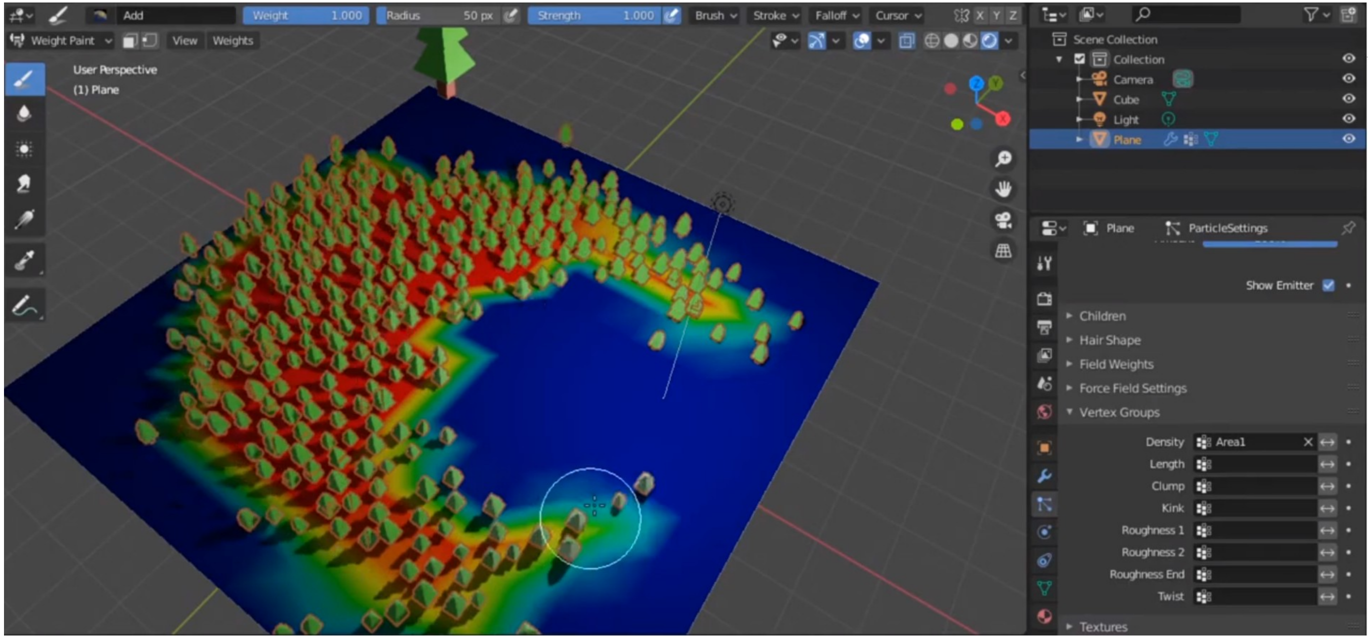




https://docs.blender.org/manual/en/latest/sculpt_paint/vertex_paint/index.html

Additional mesh information: vertex weights

- Vertex attributes can be any information that can be used for rendering
- Example: vertex weight designating density for instancing objects (e.g., particles)



https://www.youtube.com/watch?v=oPenYcM6Usw&ab_channel=HelperGround

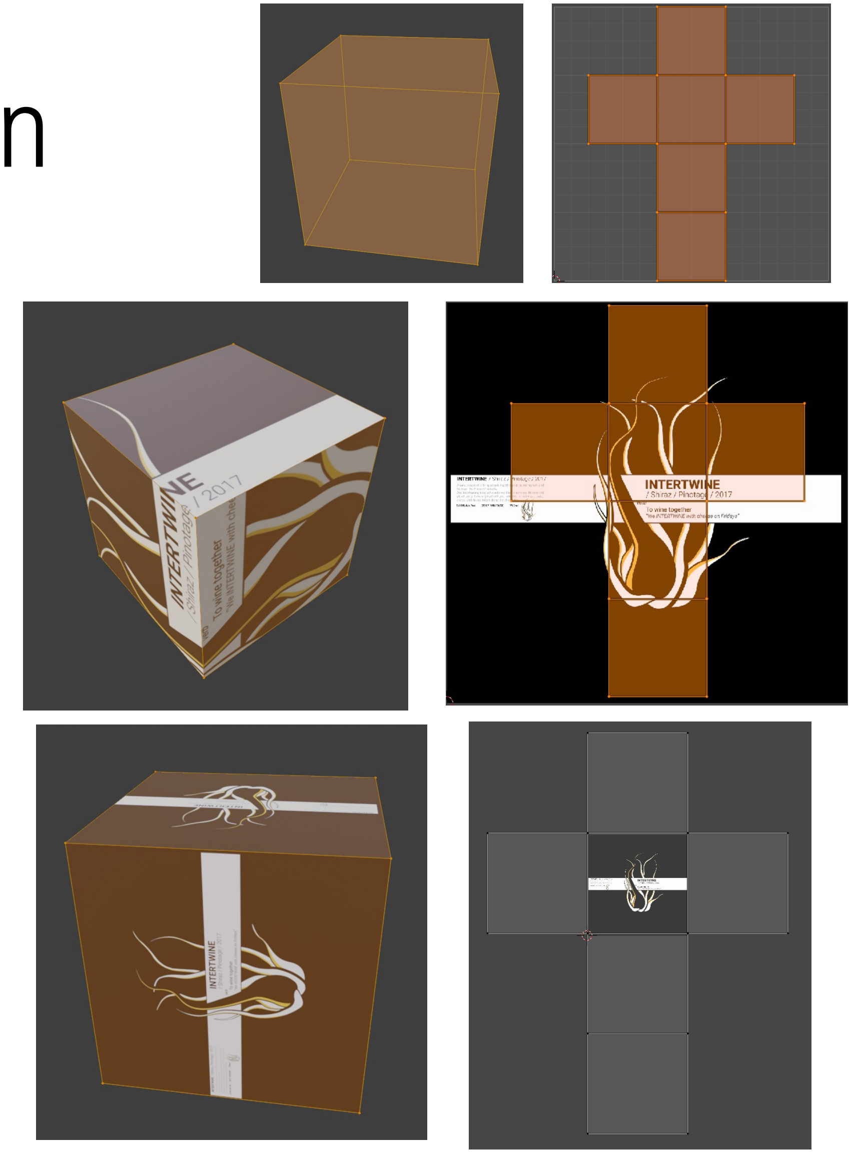
Additional mesh information: texture coordinates

- To add more details on object, often images and procedural patterns are used → **textures**.
- The problem of applying texture image on 3D object is often quite complex than on flat plane.
  - A a way of mapping a 2D image/pattern to 3D shape can be done by "unwrapping" mesh onto 2D plane.

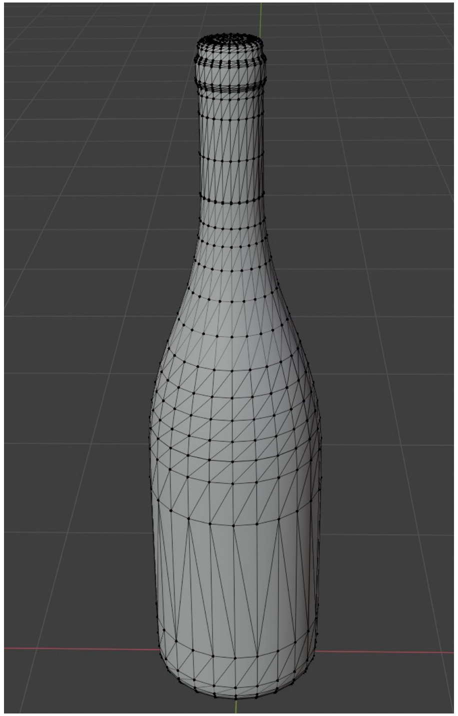
https://polyhaven.com/a/wine_bottles_01

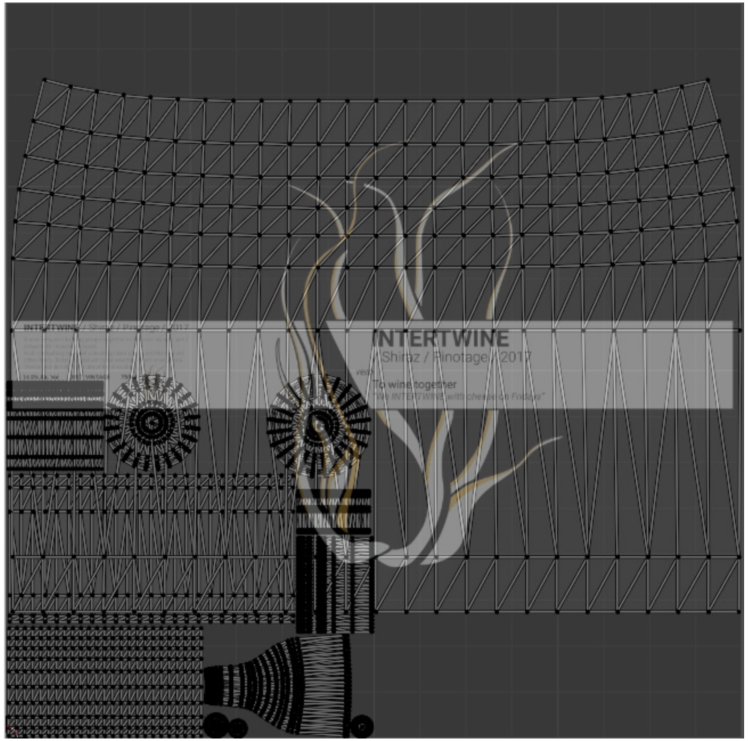
Mesh unwrapping: intuition

- Mesh is unwrapped into 2D plane we can imagine it lies in 2D coordinate system.
- Vertices of unwrapped mesh now have coordinates in this 2D coordinate system (u,v) → **texture coordinates***
  - (u,v) are in [0,1] range
  - Unwrapped faces can be manipulated, but then texture might get deformed
  - Whole unwrapped mesh can be translated or rotated or scaled to achieve different texture positioning
  - Several faces can overlap



Texture coordinates: mesh unwrap





* Note: this is one way of calculating texture coordinates. Texture coordinates can also be calculated on the fly during rendering or other methods that we will discuss later.

Polygon Mesh: datastructures, storing and transfer

Datastructures and representation

- There are multiple data structures for polygonal meshes that offer trade-off between run-time complexity and memory requirements
  - Efficient topology traversal
  - Efficient mesh updates
- Different data structures are better for rendering, storage or modeling
- Often algorithms on polygonal meshes perform:
  - Computation of normal vectors
  - Mesh smoothing, refining, simplification, compression, etc.

Mesh data structure

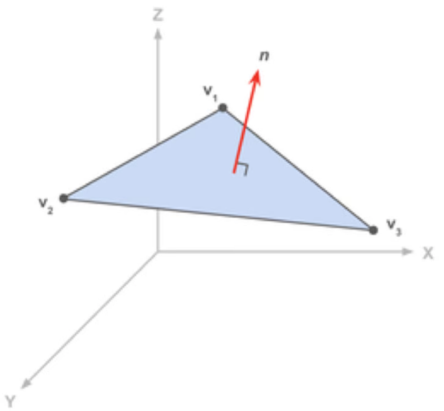
- Mesh data structure must at least describe:
  - **Geometry**: position of vertices specified using coordinates
  - **Connectivity/topology** information for vertices → how are edges and faces formed from the vertices. Options:
    - For each polygon, list all vertices that define it
    - Polygon topology: specify edges of mesh in terms of pairs of vertices (connection between vertices and edges)
    - Mesh topology: for each edge, state which polygons are incident connection between edges and polygons (connection between edges and polygons)
  - Other information not explicitly given can be calculated from given information

Explicit face representation

- **Triangle soup**: each triangle is represented using vertex coordinates (topology is not represented explicitly)
  - Triangle: 3 vertices each having 3 coordinates `(x,y,z)`.
  - n triangles: `n * 3 * 3 = 9 * n` floats → high redundancy!
- Data format used in production:
  - **STL** (STereoLitography)

t	v_i	v_j	v_k
0	(1.0, 1.0, 1.0)	(-1.0, 1.0,-1.0)	(1.0, 1.0, 1.0)
1	(1.0, 1.0, 1.0)	(-1.0,-1.0, 1.0)	(1.0,-1.0,-1.0)
2	(1.0, 1.0, 1.0)	(1.0,-1.0, 1.0)	(-1.0, 1.0,-1.0)
3	(1.0,-1.0,-1.0)	(-1.0,-1.0, 1.0)	(-1.0, 1.0,-1.0)

Example: STL file format




```
facet normal ni nj nk
    outer loop
        vertex v1x v1y v1z
        vertex v2x v2y v2z
        vertex v3x v3y v3z
    endloop
endfacet
```

```
foreach triangle
    REAL32[3] – Normal vector
    REAL32[3] – Vertex 1
    REAL32[3] – Vertex 2
    REAL32[3] – Vertex 3
    UINT16    – Attribute byte count
end
```

STL file format (ASCII and binary type) stores triangles by explicitly listing vertices and normals: https://all3dp.com/1/stl-file-format-3d-printing/

Shared vertex representation

- AKA: indexed face set
  - Reduce redundancy
- Store vertex coordinates in a separate array: **vertex array** (`3 * n` floats)
- Specification of triangles refers to vertex array table by storing indices of triangles: **triangle/face array** (`3 * n` integers)
- Data formats used in production:
  - OBJ
  - OFF

v	*x*	*y*	*z*
0	1.0	1.0	1.0
1	-1.0	1.0	-1.0
2	-1.0	-1.0	1.0
3	1.0	-1.0	1.0

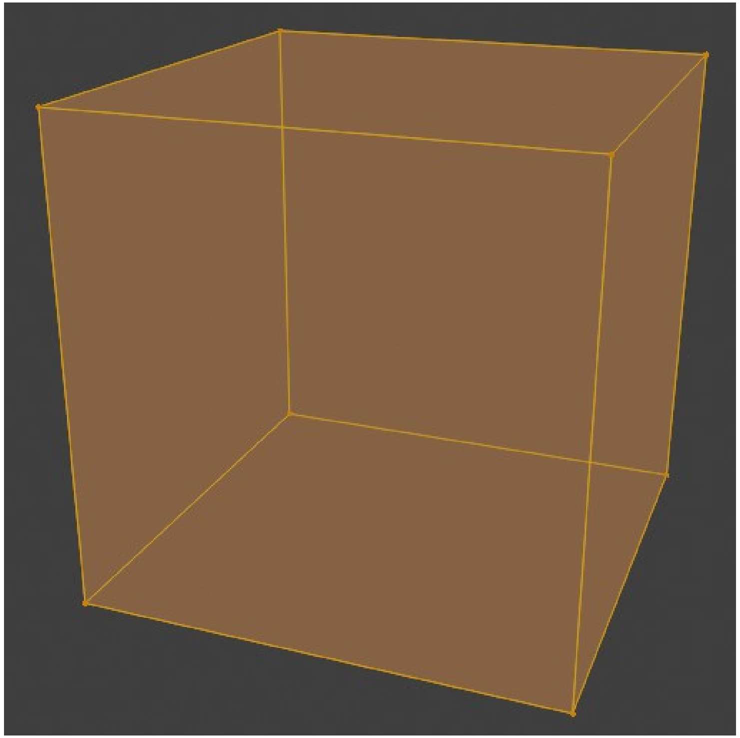
vertex array

t	*i*	*j*	*k*
0	0	1	2
1	0	2	3
2	0	3	1
3	3	2	1

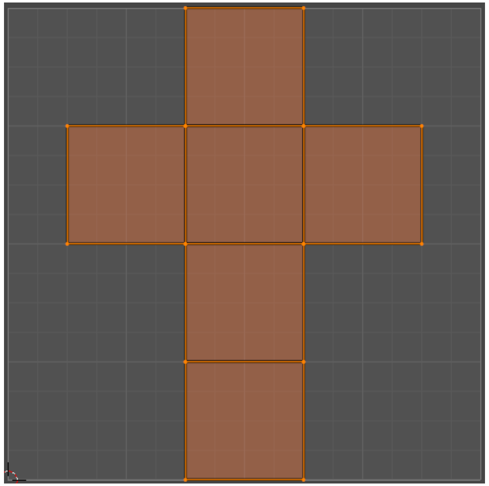
triangle array

Example: representing a cube in a computer

- 8 vertices: **vertex array – geometry information**
  - vert_array = {{-1,1,1},{1,1,1},{1,1,-1},{-1,1,-1},{-1,-1,1},{1,-1,1},{1,-1,-1},{-1,-1,-1}}
- 6 quads for representing faces. Each quad requires 4 vertices: **face index array – topology information**.
  - face_index_array = {4, 4, 4, 4, 4, 4}
  - size(face_index_array) = number of polygons used for representing a shape
- For each face, which indices of vertex array are used: **vertex index array – topology information**.
  - vertex_index_array = {0, 1, 2, 3, 0, 4, 5, 1, 1, 5, 6, 2, 0, 3, 7, 4, 5, 4, 7, 6, 2, 6, 7, 3}
  - size(vertex_index_array) = sum of all values in face index array
  - Note: each face of the cube shares some vertices with other faces



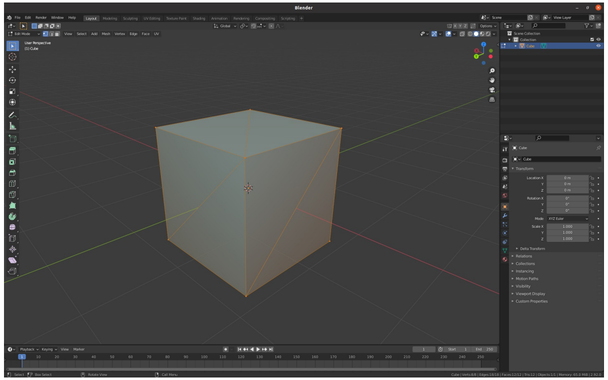
Texture coordinates



- Representing texture coordinates:
  - Texture coordinate per vertex: list of (u,v) coordinates equal to the number of vertices. Example: **texture_coordinates** = {{0,0.5}, {1,0.5}, {0.5,1}, {0,0.5}, {0.5,0}, {1,0.5}}
  - In UV space it is possible that multiple vertices have same texture coordinates thus connectivity information can be used to reduce number of texture coordinates to be written down

Example: OBJ file format

- Less redundancy than explicit representation (e.g., STL)
- Separates geometry (vertices) and topology (connectivity)
- This representation is costly for computation
- Each line starts with letter representing type of data:
  - `v` – vertices
  - `vt` – texture coordinates
  - `vn` – normals
  - `f` – faces (index of vertex, vertex texture coordinate, vertex normal

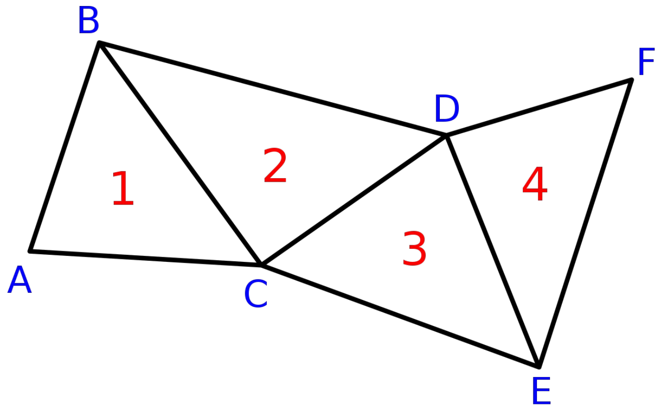


OBJ file format is further extensible:
https://en.wikipedia.org/wiki/Wavefront_.obj_file

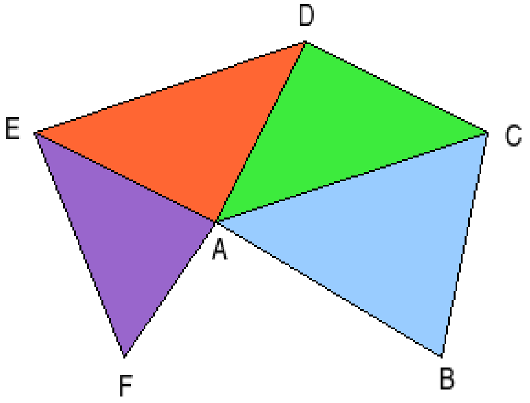
```
# Blender v2.92.0 OBJ File: ''
# www.blender.org
o Cube_Cube.002
v -1.000000 -1.000000 1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
vt 0.625000 0.000000
vt 0.375000 0.250000
vt 0.375000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.500000
vt 0.625000 0.500000
vt 0.375000 0.750000
vt 0.625000 0.750000
vt 0.375000 1.000000
vt 0.125000 0.750000
vt 0.125000 0.500000
vt 0.875000 0.500000
vt 0.625000 1.000000
vt 0.875000 0.750000
vn -1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 1.0000
vn 0.0000 -1.0000 0.0000
vn 0.0000 1.0000 0.0000
s off
f 2/1/1 3/2/1 1/3/1
f 4/4/2 7/5/2 3/2/2
f 8/6/3 5/7/3 7/5/3
f 6/8/4 1/9/4 5/7/4
f 7/5/5 1/10/5 3/11/5
f 4/12/6 6/8/6 8/6/6
f 2/1/1 4/4/1 3/2/1
f 4/4/2 8/6/2 7/5/2
f 8/6/3 6/8/3 5/7/3
f 6/8/4 2/13/4 1/9/4
f 7/5/5 5/7/5 1/10/5
f 4/12/6 2/14/6 6/8/6
```

Example: specific data structures

- Data structures for minimal redundancy and memory efficient representation by implicit topology:
  - Triangle/quad strip
  - Triangle fan
- Complex meshes are represented as set of strips and fans
  - Used for hardware rendering (OpenGL - https://www.khronos.org/opengl/wiki/Primitive)



Triangle strip: implicit representation of strip-like sequence of triangles. https://en.wikipedia.org/wiki/Triangle_strip



Triangle fan: set of triangles sharing one vertex https://www.khronos.org/opengl/wiki/Primitive

Recap: representing and storing mesh

- To describe a mesh we need:
  - Vertex array (vertex positions)
  - Face index array (how many vertices each face is made of)
  - Vertex index array (connectivity)
  - Primitive variables:
    - Vertex color
    - Normals
    - Texture coordinates
    - Other optional application-specific values

Storing polygons: practical note

- Even single polygon mesh in a 3D scene can can be quite large (10^5-10^6 vertices is not unusual)
- Storing vertices and connectivity information must be performed efficiently
  - All vertices must be stored
  - Different techniques exist which try to minimize the amount of data needed for representing connectivity → yielding different standards, formats and API specifications for storing and transferring mesh data, e.g., OBJ or FBX*.

* Those are popular and widely used standards. We will discuss them more when we will be talking about triangle meshes. RenderMan, on the other hand, defines API specification for representing mesh data.

Storing and transferring mesh objects

- Mesh data is often transferred between different modeling, rendering and interactive tools.
- Interface between different tools and specification of how mesh should be stored is defined by standards*: https://renderman.pixar.com/resources/RenderMan_20/ribBinding.html
- Different implementations of mesh storage formats exists, which are:
  - Are more or less compact
  - Are more or less human-readable
  - Can contain additional object data which is described with the mesh (textures, materials, etc.)
  - Can contain various metadata (e.g., physical behavior of object described with mesh)
  - Store only mesh information
  - Store whole scene and mesh is only one of elements
- Popular formats:
  - https://all3dp.com/2/most-common-3d-file-formats-model/
  - https://www.sidefx.com/docs/houdini/io/formats/geometry_formats.html
- 3D scene is not necessary created, rendered and used in same software. Usually, whole pipeline of software is used, at least:
  - DCC → game engines
- Formats: OBJ, GLTF, USD
- Interesting: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh/polygon-mesh-file-formats

* note that tendency is towards standardization of whole scene description. Which, besides mesh polygons, include materials, lights, cameras, different shape representations, etc.

Practical note: using different formats

- Professional file formats and representations are commonly used and are very efficient
- Using modeling/interactive/rendering tools, user is often provided with "importer/exporter" – feature which enables importing and exporting different file formats
  - Example: https://www.sidefx.com/docs/houdini/io/formats/index.html
- The problem comes if one writes its own rendering program and parsing file formats for import can be not that easy. Luckily, different libraries can be used for this purposes:
  - Example: https://github.com/assimp/assimp

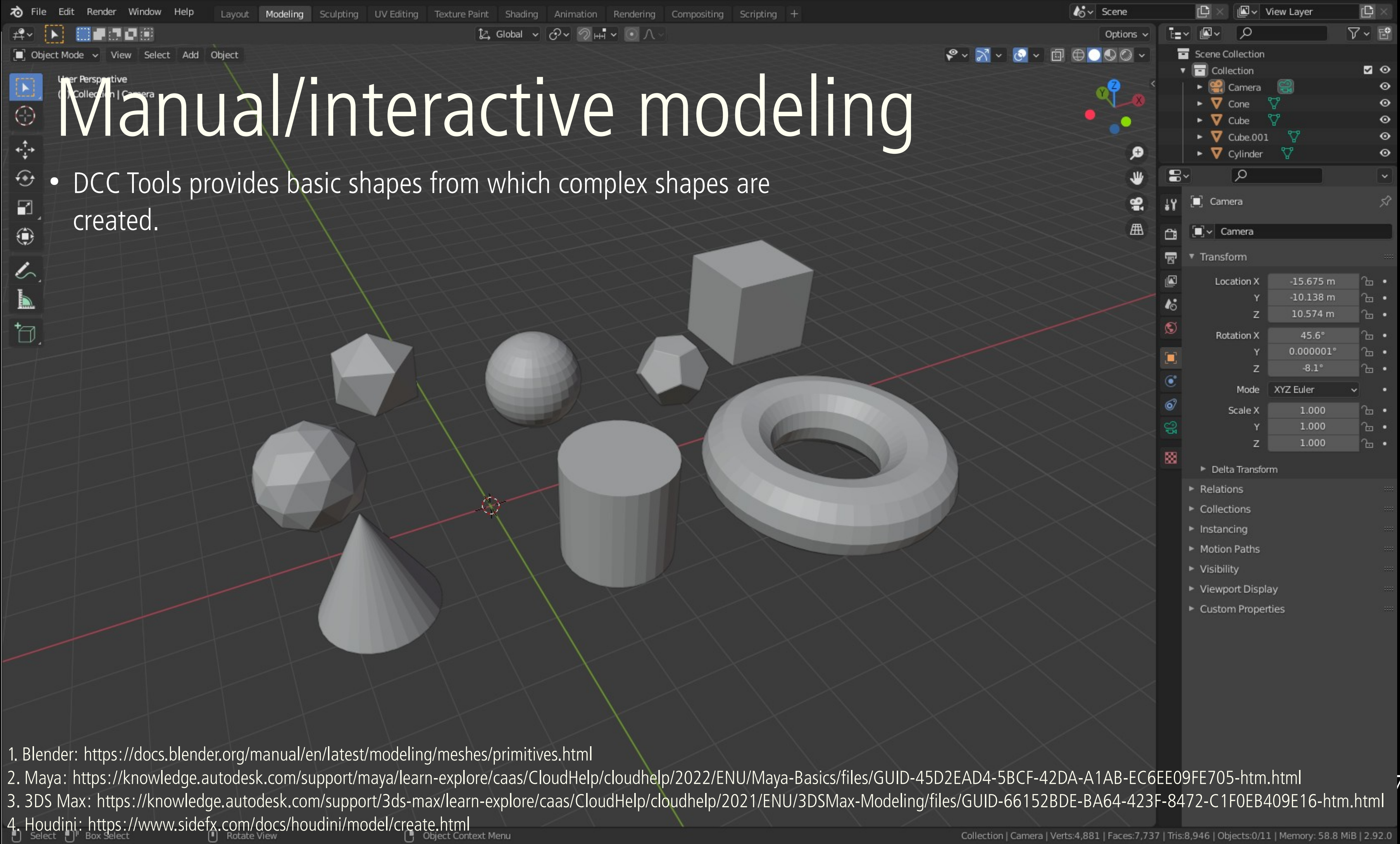
Practical note: units

- Note that until now we haven't discussed units in which shape is stored.
- Units are important for preserving consistent scale among different modeling/rendering tools in which object might be transferred.
- Let's discuss creation of two meshes in Blender. Mesh, that is, vertices are defined in a coordinate system. In this coordinate system we can have two same objects where one is larger and another is smaller. By relation/proportion we can distinguish the scale and we might not care about units.
  - Since positions of vertices are relative to coordinate system, the axis of coordinate system must specify units.
- But what happens if one of this objects in exported into another tool, for example Unity? Unity might use different coordinate system unit scale and object might be too big or too small in this coordinate system!
  - Example: different scales with same coordinates.
- Unit is defined by user who models the object and it must be taken care of during transfer.

Mesh polygons: modeling and acquisition

Manual/interactive modeling

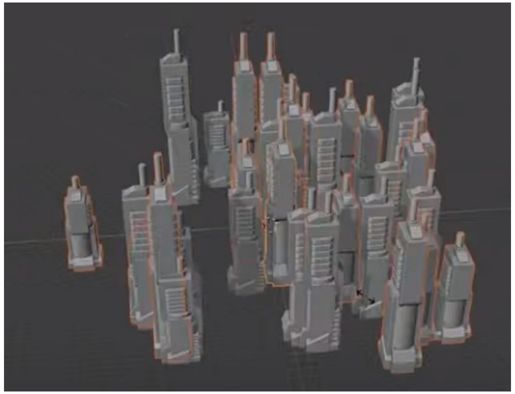
- DCC Tools provides basic shapes from which complex shapes are created.

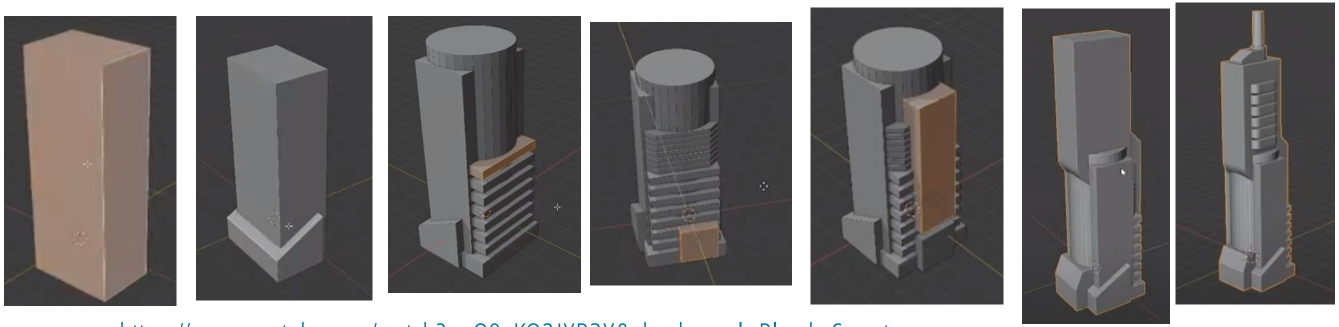


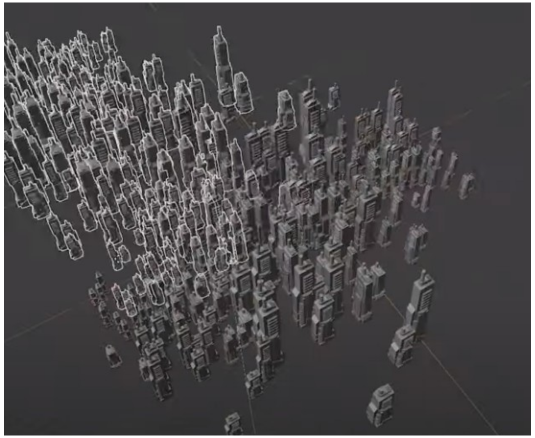
1. Blender: https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html
2. Maya: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2022/ENU/Maya-Basics/files/GUID-45D2EAD4-5BCF-42DA-A1AB-EC6EE09FE705-htm.html
3. 3DS Max: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2021/ENU/3DSMax-Modeling/files/GUID-66152BDE-BA64-423F-8472-C1F0EB409E16-htm.html
4. Houdini: https://www.sidefx.com/docs/houdini/model/create.html

Practical tip: complex shapes from base shapes

- Anything can be decomposed in simple forms[1,2]: box, sphere, cylinder, torus, cones, etc.







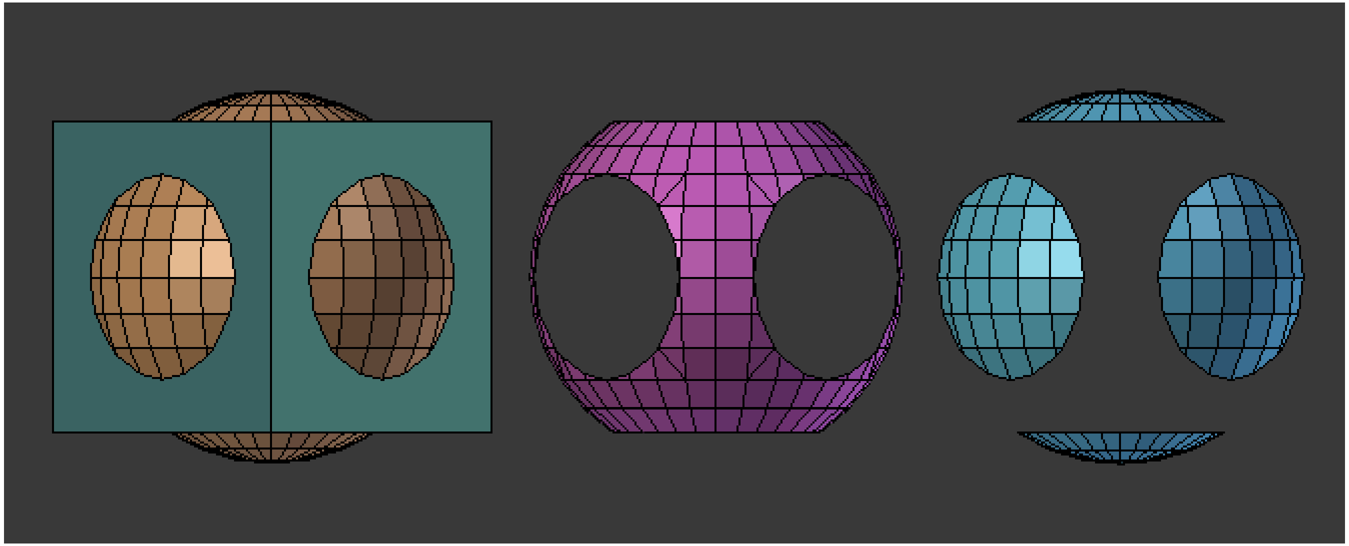
https://www.youtube.com/watch?v=Q0qKO2JYR3Y&ab_channel=BlenderSecrets

1. http://www.thedrawingwebsite.com/2015/02/18/practicing-your-draw-fu-forms-forms-are-like-sentences/

2. https://www.youtube.com/watch?v=6T_-DiAzYBc&list=RDCMUClM2LuQ1q5WEc23462tQzBg&start_radio=1&rv=6T_-DiAzYBc&t=1343&ab_channel=Proko

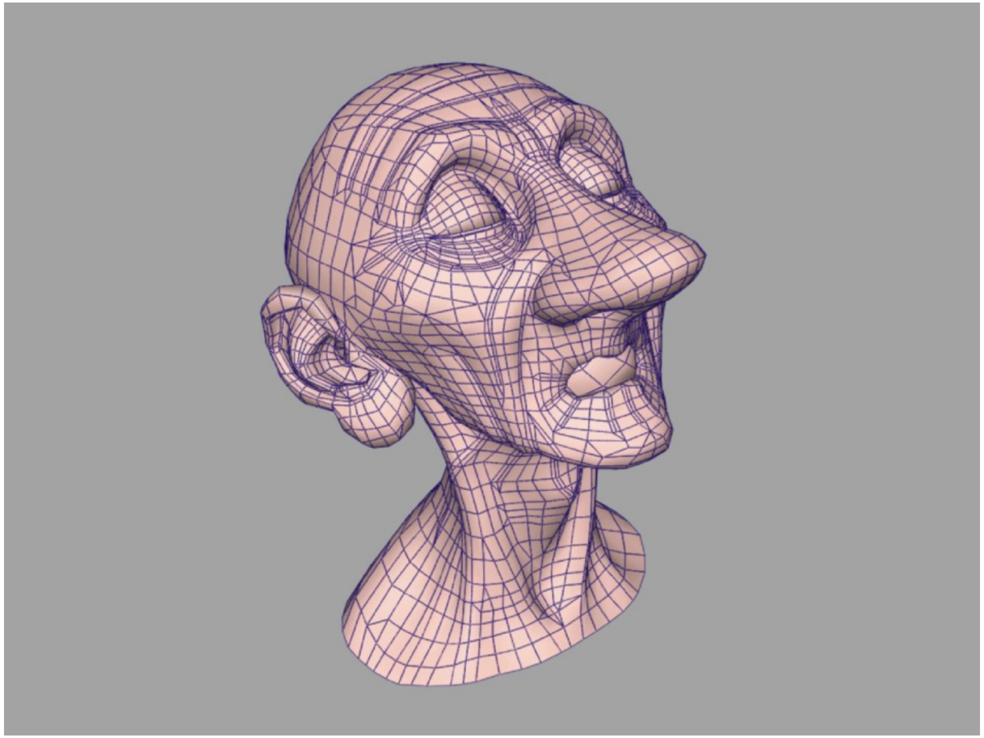
Constructive solid geometry

- Boolean operations applied on meshes:
  - Union
  - Intersection
  - Difference

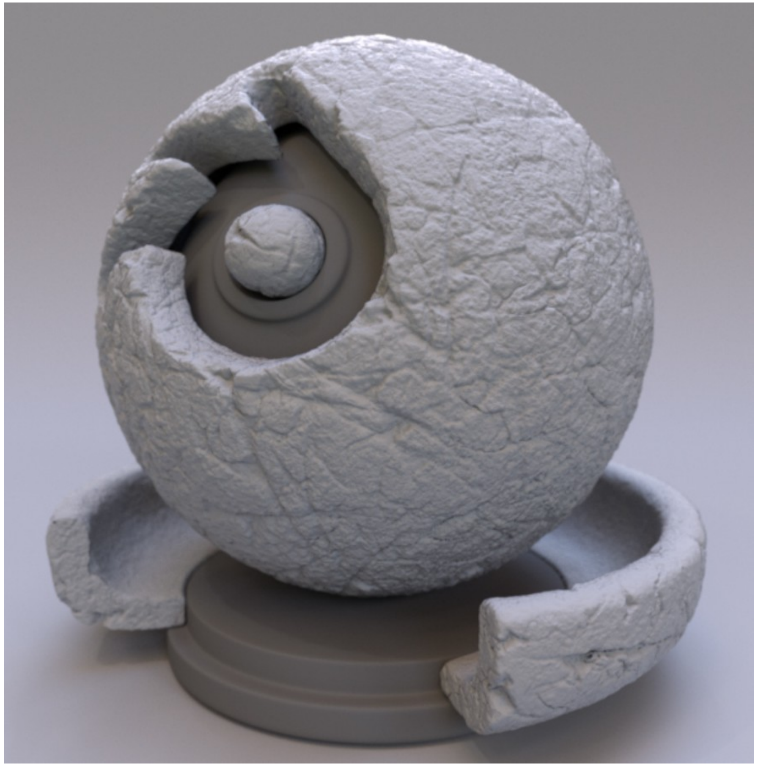


Union, intersection and difference Boolean operators applied on meshes: https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/booleans.html

Subdivision surfaces and displacement



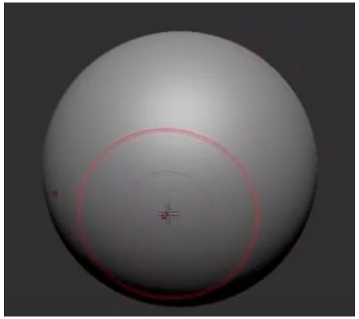
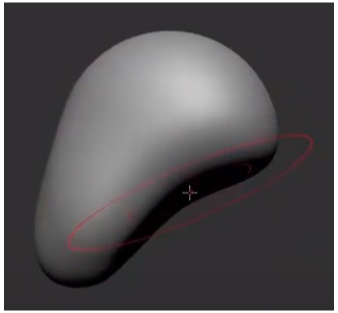
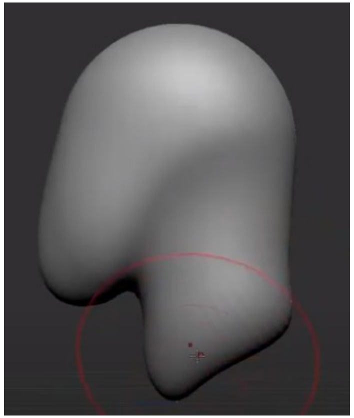
Subdivision surfaces for smooth surfaces in Pixar:
https://graphics.pixar.com/library/Geri/paper.pdf
https://www.fxguide.com/fxfeatured/pixars-opensubdiv-v2-a-detailed-look/

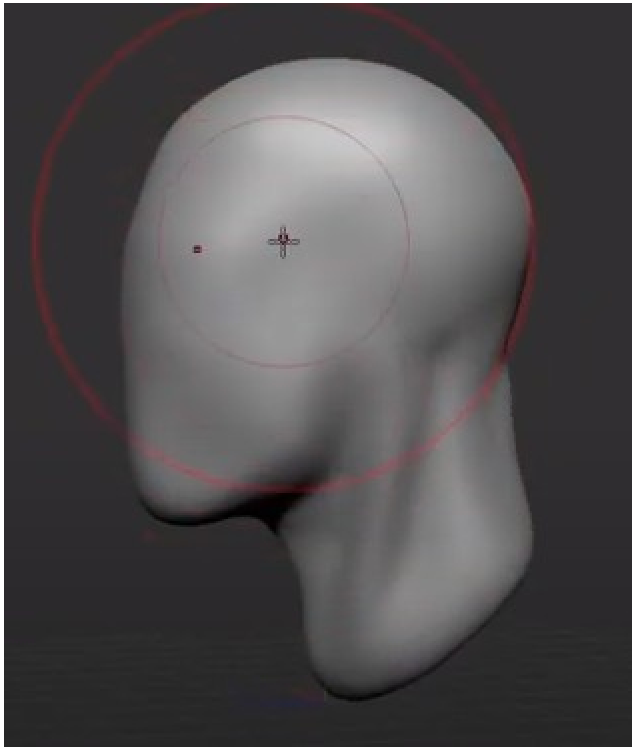


Blender displacement of subdivided surface:
https://docs.blender.org/manual/en/latest/render/materials/components/displacement.html

Practical tip: complex shapes from base shapes

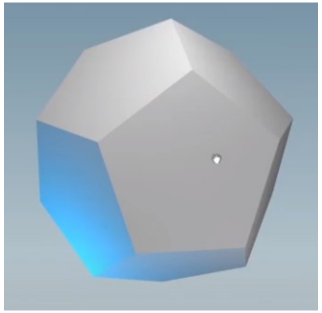
- Sculpting base shapes.

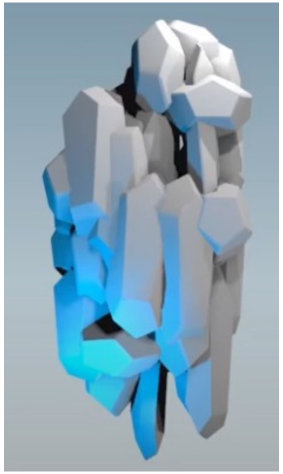




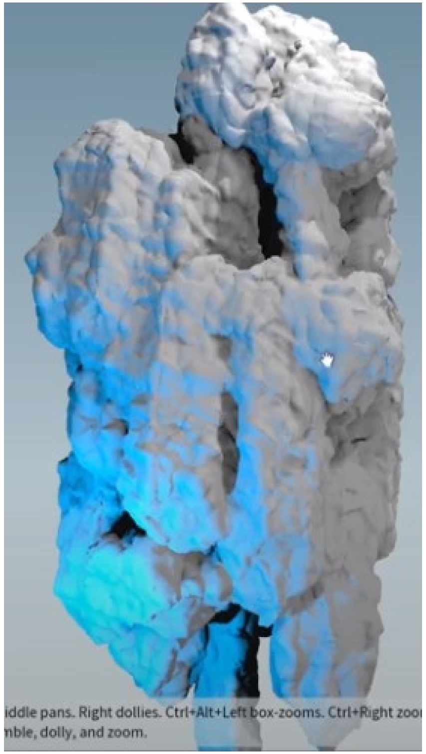


https://www.youtube.com/watch?v=p4Y1hVCMMXw&ab_channel=RealEnay

Practical tip: complex shapes from base shapes





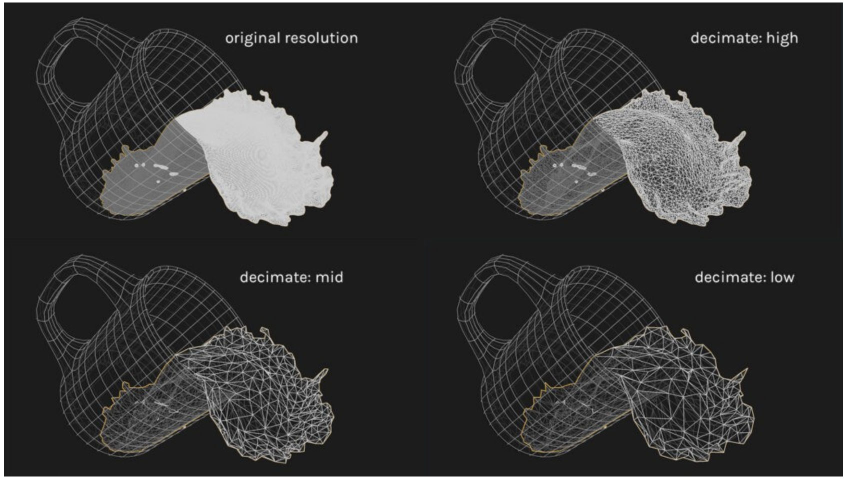


- Procedural modeling

https://www.youtube.com/watch?v=fxOxygaEOFk&ab_channel=SimonHoudini

Simulations

- Polygonal model can be generated using simulations (e.g., fluid simulation using voxel representation) which is then tessellated into polygons for rendering.



Fluid simulation in Blender:
https://andi-siess.de/designing-a-book-cover-as-3d-vector/

Houdini fluid simulation: https://www.sidefx.com/tutorials/crashing-wave/

Procedural modeling

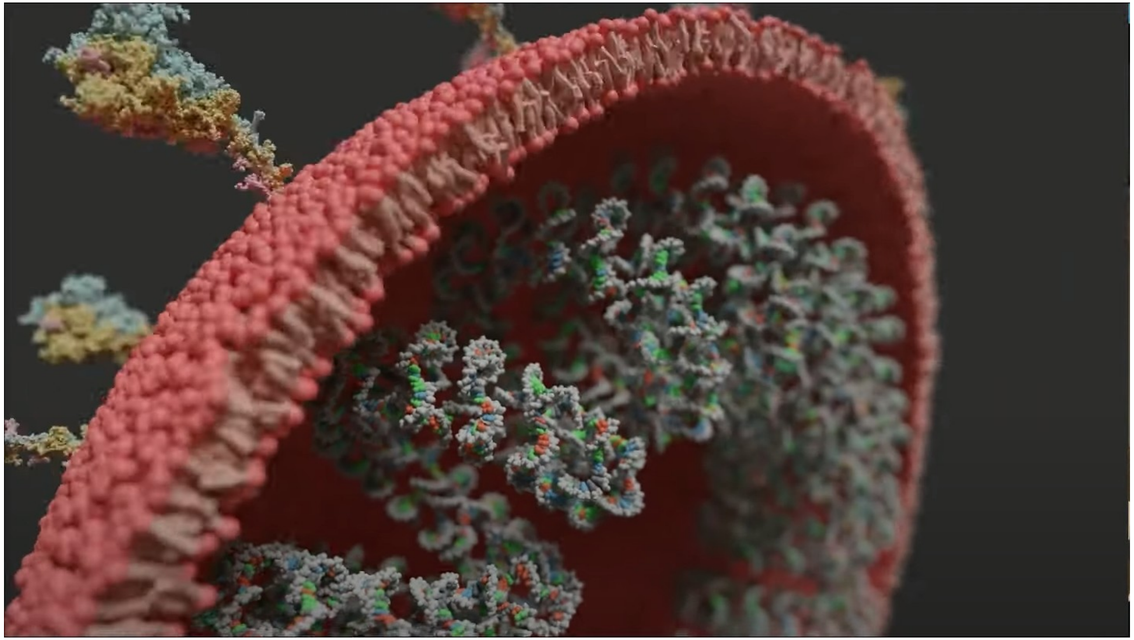
- Similarly to simulations, polygonal mesh can be created by writing programs that create such data – procedural modeling

Procedural modeling in Houdini:
https://entagma.com/category/tutorials/

Blender procedural modeling using geometry nodes:
https://blenderartists.org/t/procedural-abandoned-house-with-geometry-nodes/1363024

Transforming data into visualizations

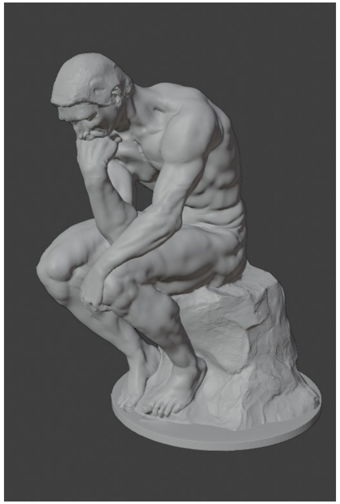
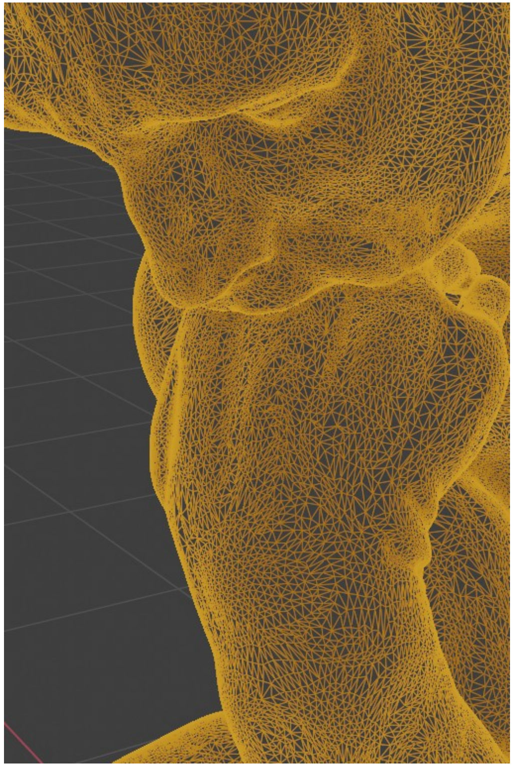
- Data found in different forms can be converted to polygon data. Example: visualizing protein data using spheres.



Visualizing viruses in Blender:
https://www.youtube.com/watch?v=adhTmwYwOiA&ab_channel=Blender

Scanning and photogrammetry

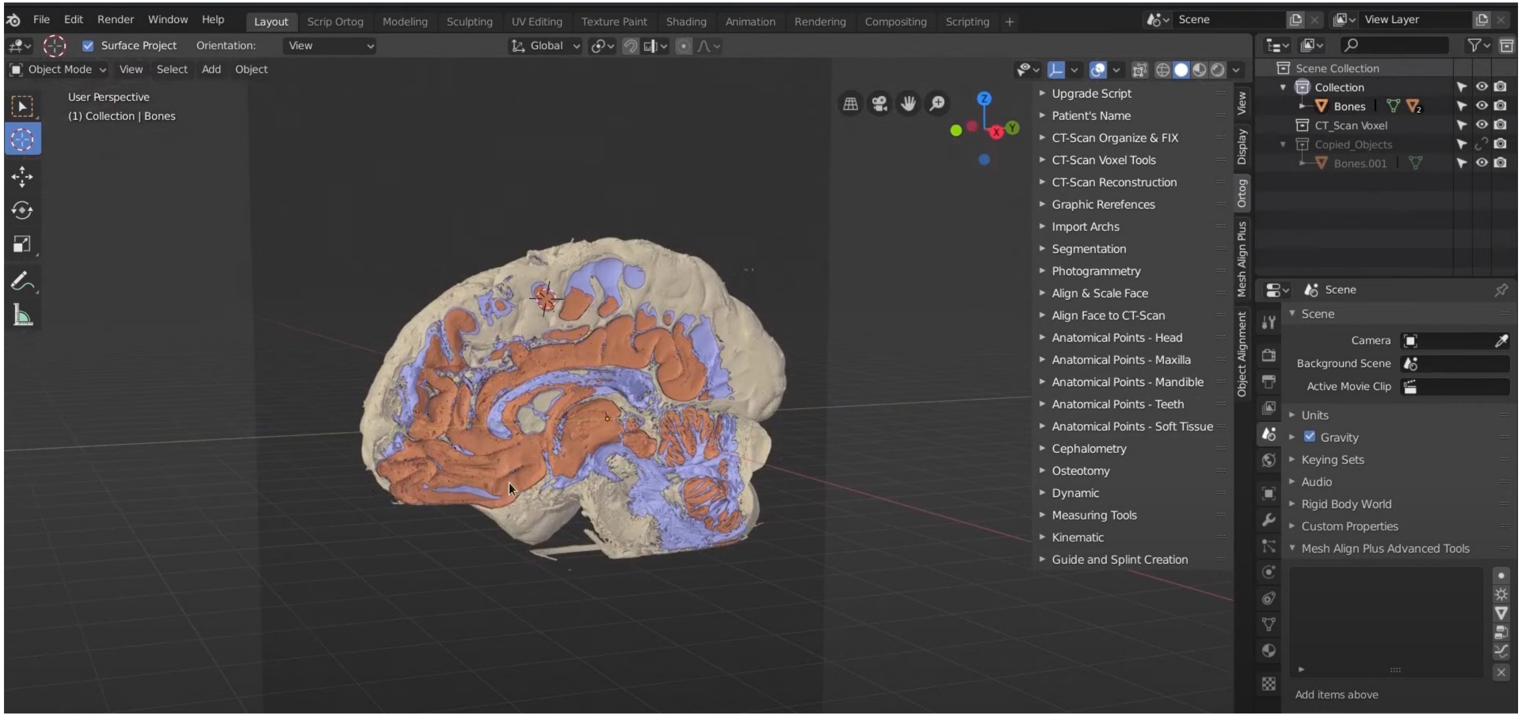
- Polygonal model can be generated using scanning and photogrammetry which results in point clouds and then is transformed into polygons for rendering.

Scan the world: https://www.myminifactory.com/scantheworld/
Sans Factory: https://www.scansfactory.com/
Mega Scans: https://quixel.com/megascans/home/
Art Station 3D scanning and photogrammetry:https://www.artstation.com/channels/photogrammetry_3d_scanning?sort_by=popular

Data from volumetric scans

- Generating surface from volume data scanned with CT or MRI.

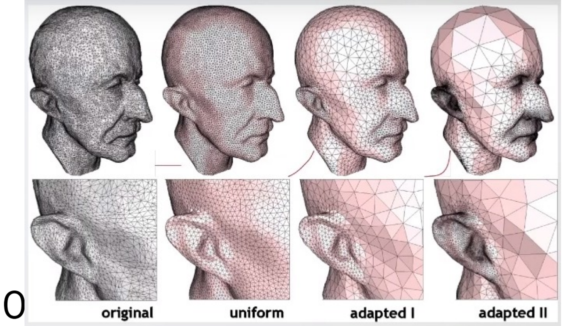


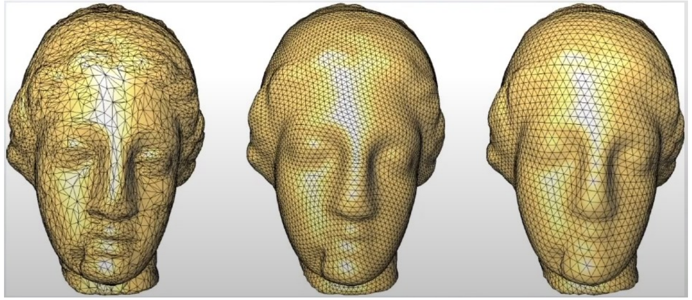
MRI to mesh in Blender: https://www.blendernation.com/2019/07/15/converting-a-video-into-a-dicom-and-a-3d-mesh/

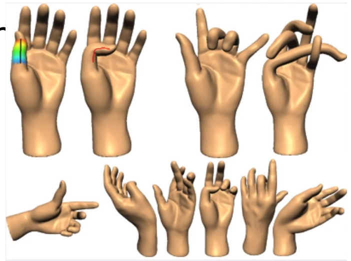
Mesh polygons: outlook

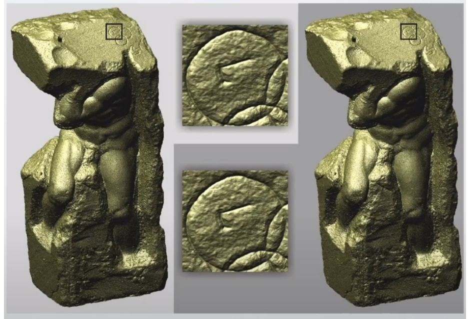
Deeper into topic

- Polygonal mesh is highly used and researched method in computer graphics. We have covered foundations. There are many other topics. Some of those will be discussed later, some are out of scope for this course:
  - Remeshing
  - Simplification, level of detail
  - Mesh compression
  - Consolidation, mesh reparatio
  - Animation, skinning and deform









Exploring meshes

- **Meshlab** is open-source tool for mesh manipulation and processing
  - Mesh smoothing and sharpening
  - Re-meshing: subdivide, re-sample, simplify,
  - Topological operations: fill holes, fix self-intersections
  - Boolean operations
- **libigl and CGAL** libraries for mesh processing
- **Blender**
  - Modeling with meshes
  - Procedural meshes with Python
- **Sources of mesh data**:
  - https://casual-effects.com/data/index.html
  - https://polyhaven.com/models
  - https://sketchfab.com/
  - http://graphics.stanford.edu/data/3Dscanrep/
- **More informations:**
  - https://www.realtimerendering.com/#polytech

Mesh shape representation: verdict

- Pros:
  - Most common surface representation
  - Simple for representing and intuitive for modeling and acquisition
  - A lot of effort has been made to represent various shapes with meshes
  - Lot of research has been done to convert other shape representations to mesh representation
  - Graphics hardware is adapted and optimized to work with (triangle) meshes → fast rendering
- Cons:
  - Not Guaranteeing smoothness
  - Triangle meshes are not efficient or intuitive for manual/interactive modeling
  - Not every object is well suited to mesh representation:
    - Shapes that have geometrical detail at every level (e.g., fractured marble)
    - Some objects have structure which is unsuitable for mesh representation, e.g., hair which has more compact representations

Reading material

- https://www.youtube.com/watch?v=V3Npa0uZYlE&list=PL4TptkuzgxxUVZ-_DiO33kp4_rkoAy1BC&index=6&ab_channel=ChristophGarth
- https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh/introduction.html
- https://www.pbr-book.org/3ed-2018/Shapes
- Real-time rendering book
  - Chapters: 16 and 17
- Computer graphics practices and principles book
  - Chapters 8, 9, 23 and 25