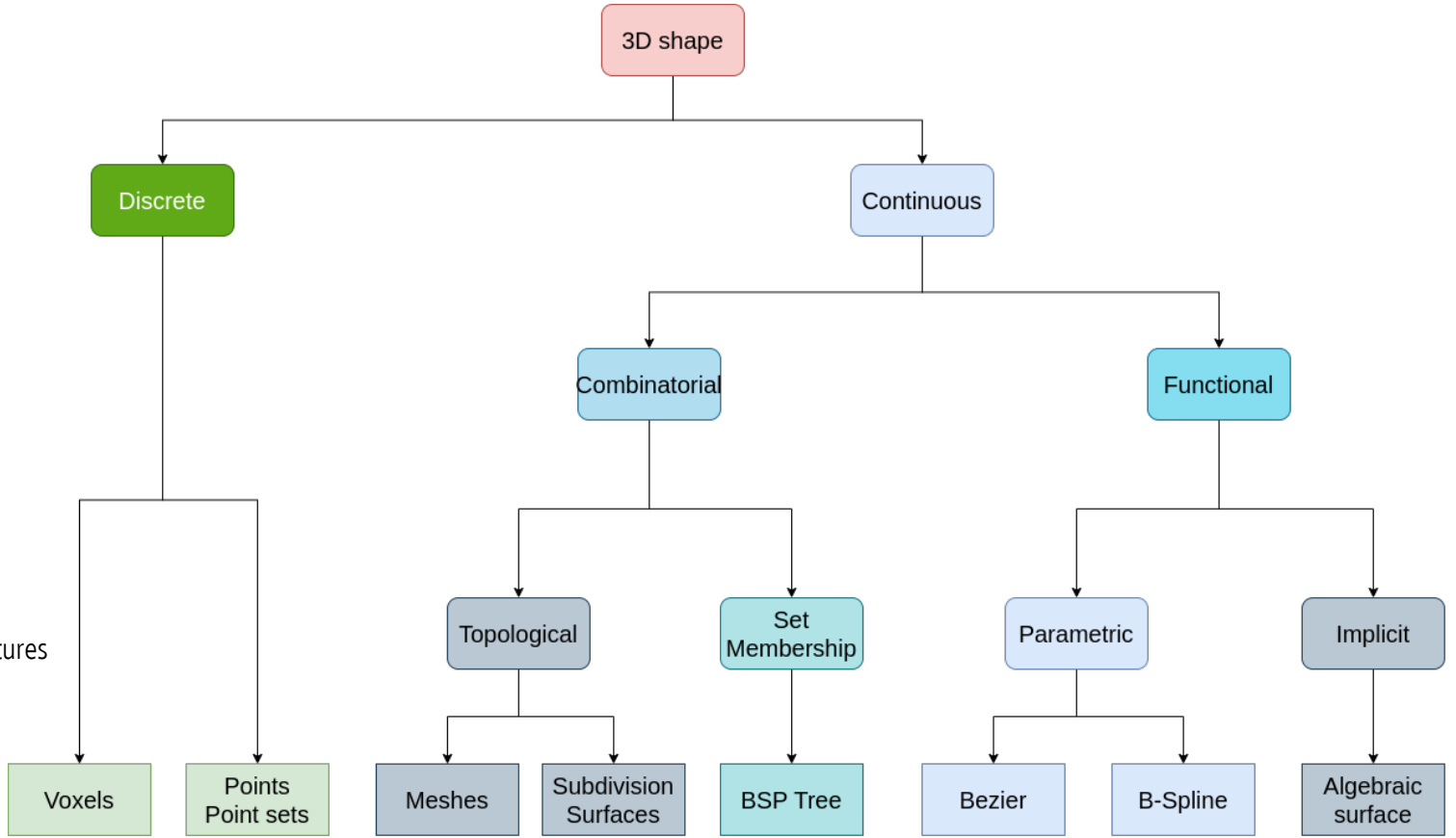
3D Models

Shape representations

Recap: shape representations

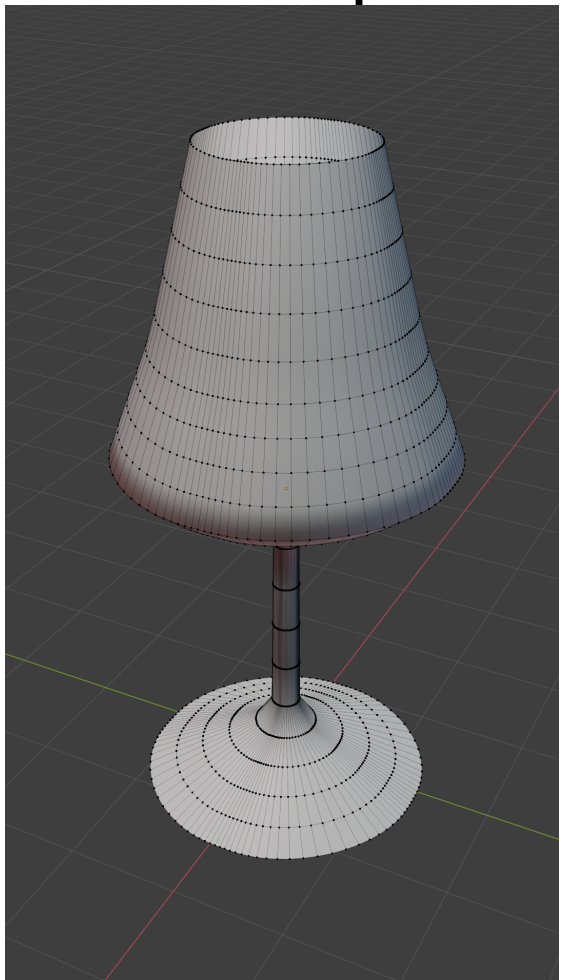
- Points
  - Point clouds
  - Particles and Particle systems
- **Surfaces:**
  - **Polygonal mesh**
  - Subdivision surfaces
  - **Parametric surfaces**
  - Implicit surfaces
- Volumetric objects/solids
  - Voxels
  - Space partitioning data-structures
- High-level structures
  - Scene graph

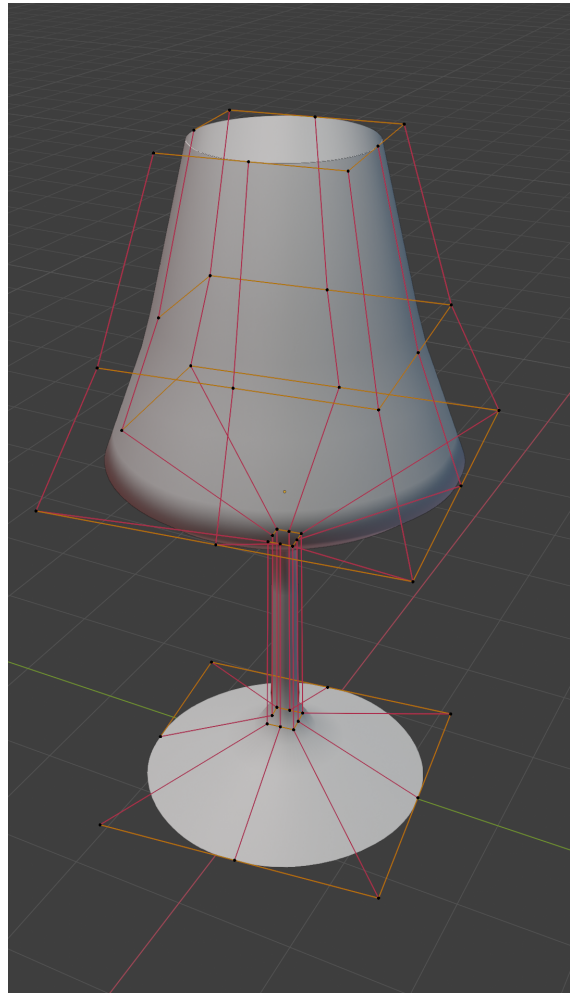


Foundations of 3D surface representation

Foundational **surface shape representations** found in geometrical modeling are*:

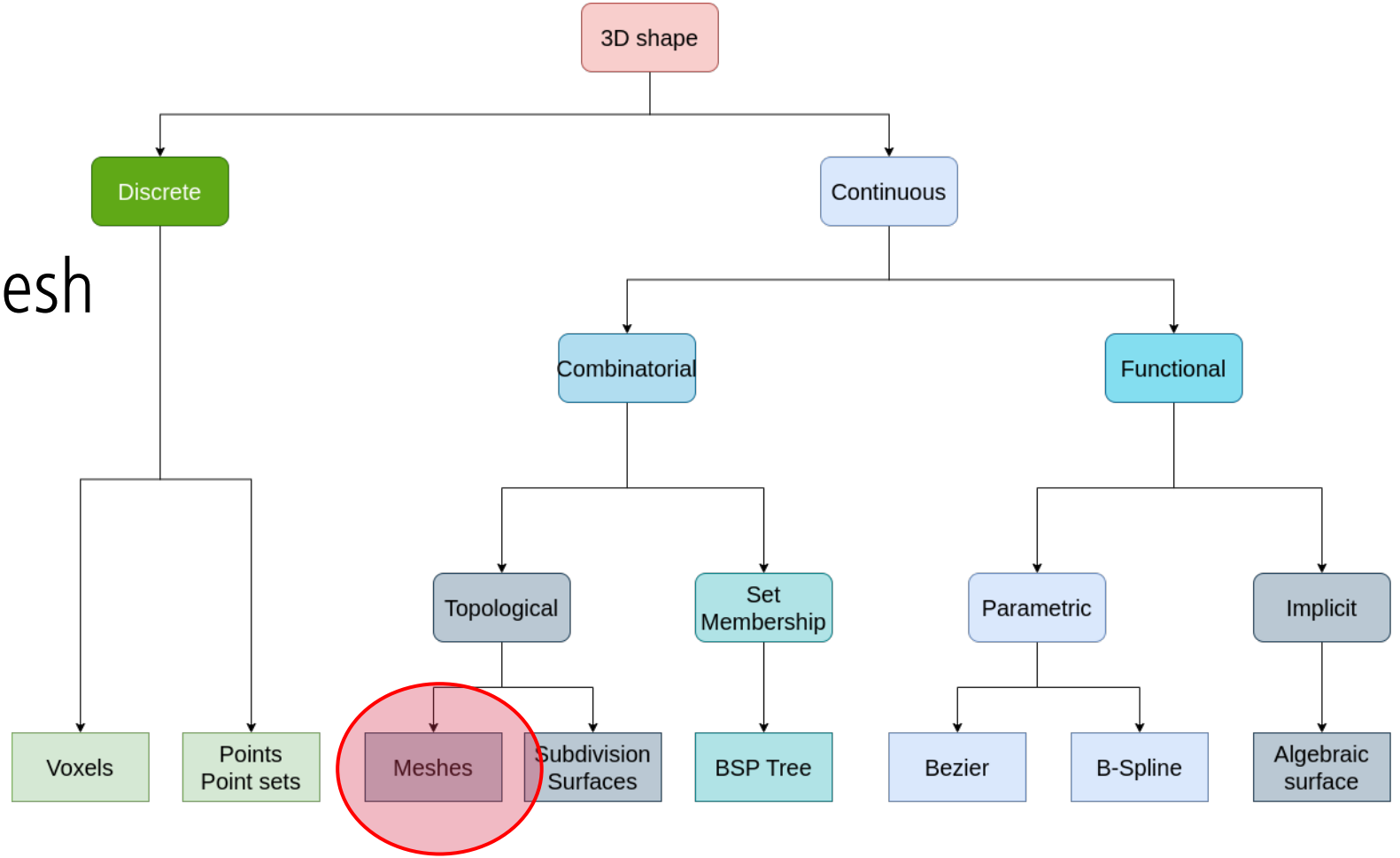
- **Polygon meshes**
- **Parametric surfaces**





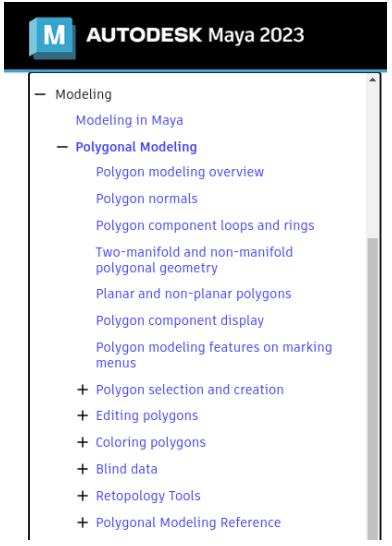
*Note that these representations are used to describe surface of the shape (a manifold – 2D surface in 3D world). Later, we will discuss how to describe interior of object (its volume). Interior of object can be described purely with spatially varying material enclosed in described surface. Also, advanced shape representations (e.g., voxels) can be used to efficiently describe the mesh. Since the topic of volumetric representation requires more knowledge about material and/or advanced shape representations, it will be covered later.

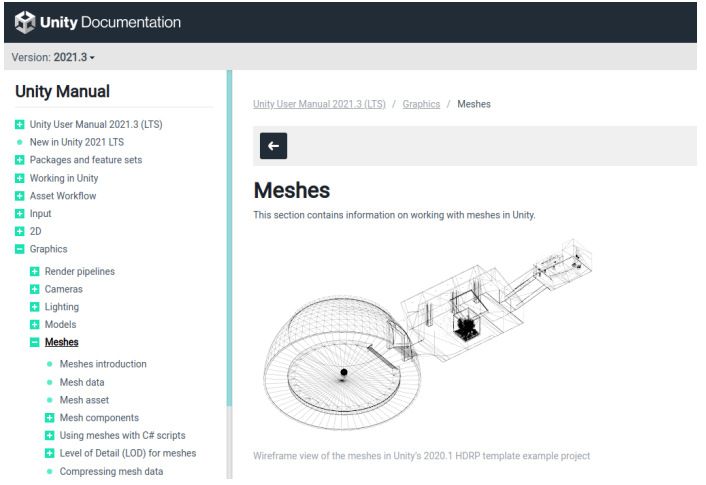
Polygonal mesh

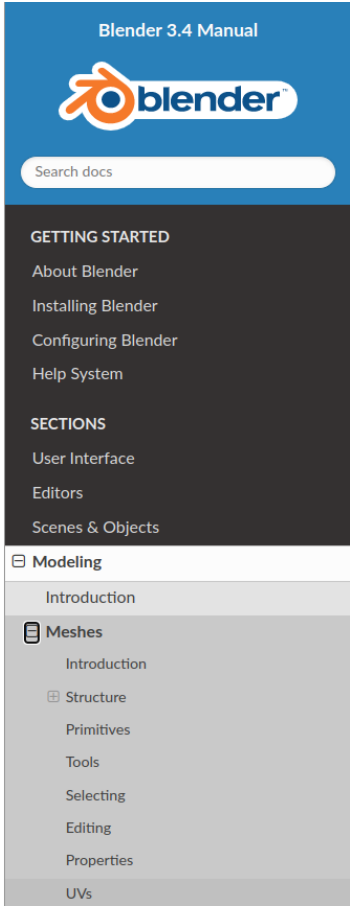


Polygon meshes

- **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







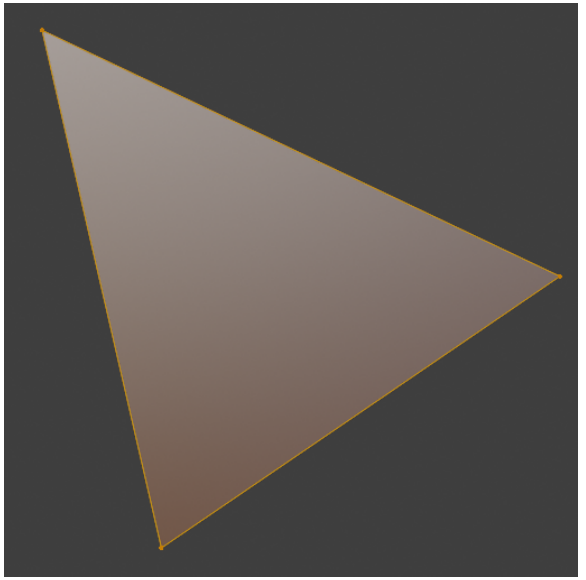
Blender: https://docs.blender.org/manual/en/latest/modeling/meshes/
Maya: https://help.autodesk.com/view/MAYAUL/2023/ENU/?guid=GI
Houdini: https://www.sidefx.com/docs/houdini/nodes/lop/mesh.html
Unity: https://docs.unity3d.com/Manual/class-Mesh.html
Unreal: https://docs.unrealengine.com/4.26/en-US/WorkingWithContent/Types/StaticMeshes/

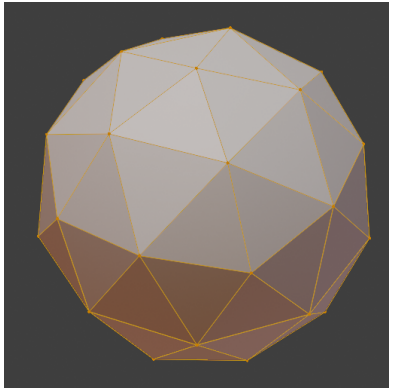
* Very often, mesh is commonly used for transporting models and scenes from DCC tools to game engines. DCC tools enable modeling using different shape representations, but in a lot of cases, all shapes are transformed to mesh representation and exported to other programs.

Mesh polygons: representation

Mesh building block: polygon

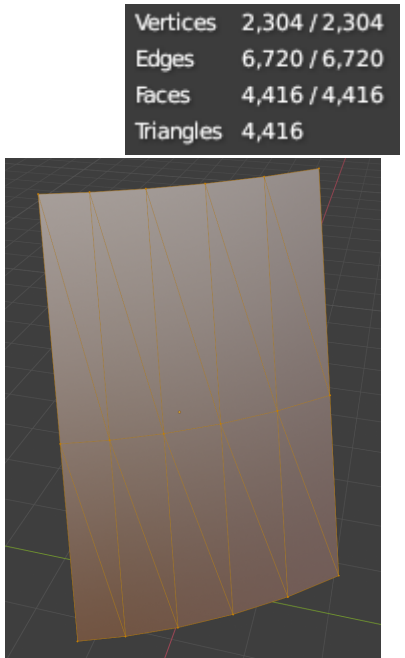
- Polygon is planar shape which is defined by connecting array of points.
- Individual points are called **vertices** (vertex, singular)
  - In 2D they are defined using two coordinates,e.g., (x,y)
  - In 3D they are defined using three coordinates, e.g., (x,y,z)
- Lines connecting two vertices are called **edges**.
- Once edges are presented and connect vertices we can define a **face**
  - Order of connecting vertices matter and it can be clockwise or counterclockwise – **winding direction**
  - Face orientation is defined by **normal** and normal depends on winding direction

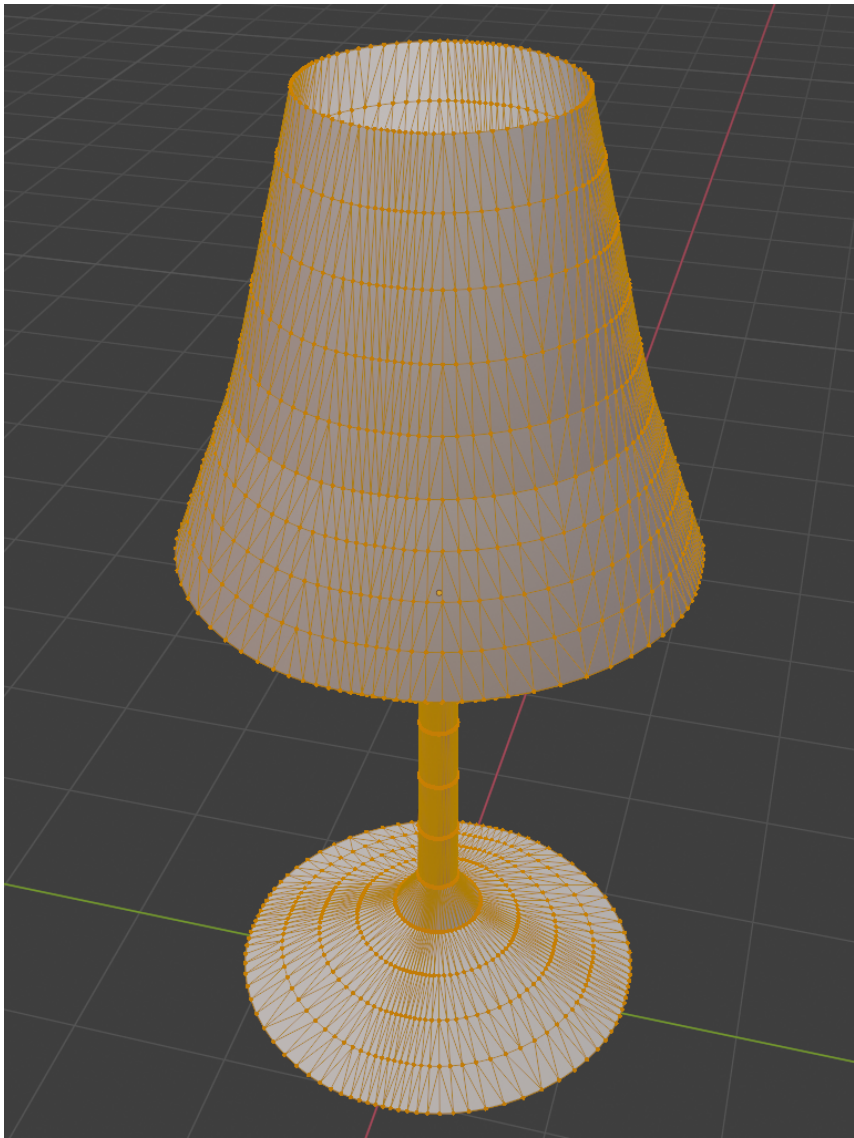




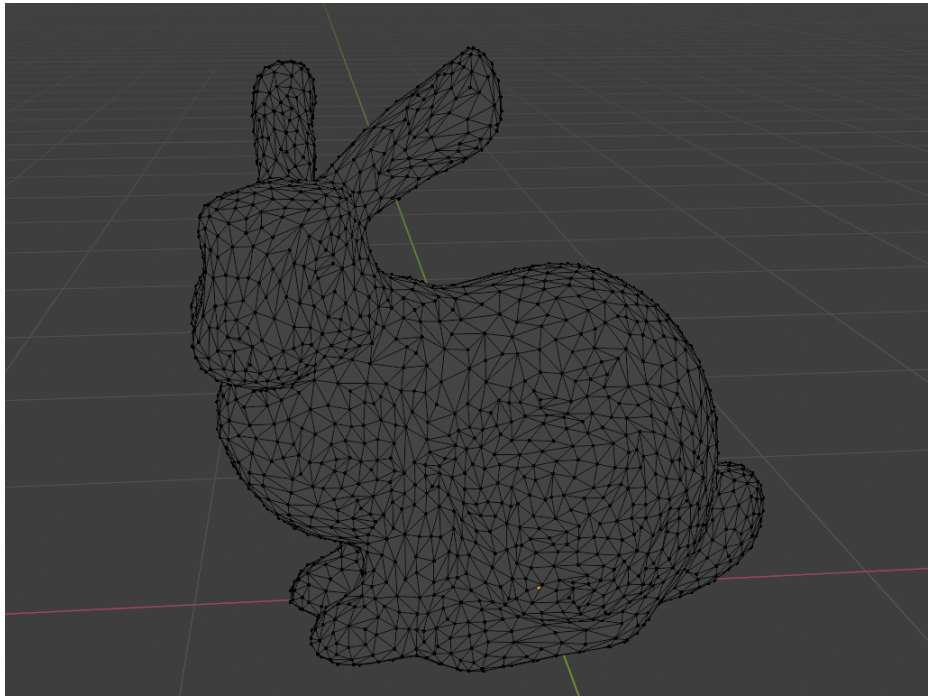
Types of polygons: triangle

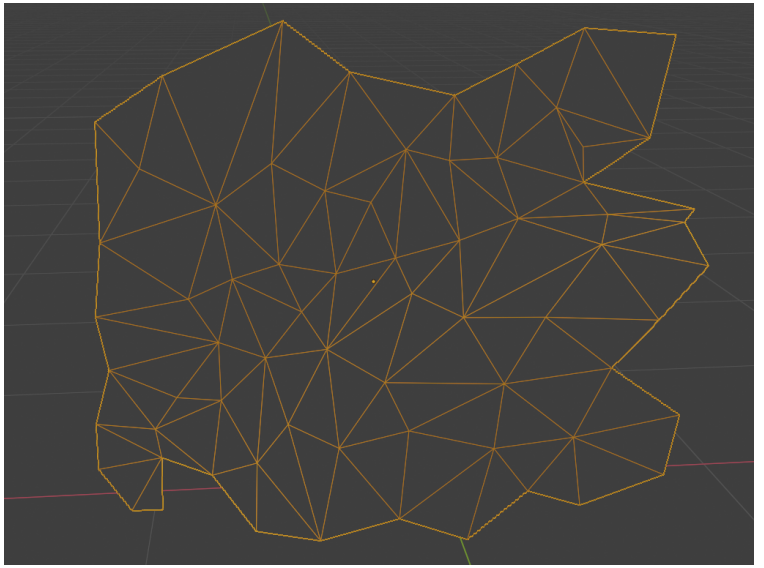
- Polygon with three vertices → **triangle polygon**.
  - Very important type of polygon in computer graphics!





Types of polygons: triangle





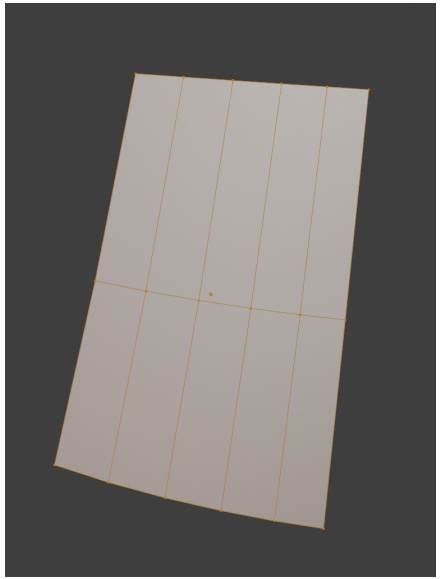
Triangle mesh

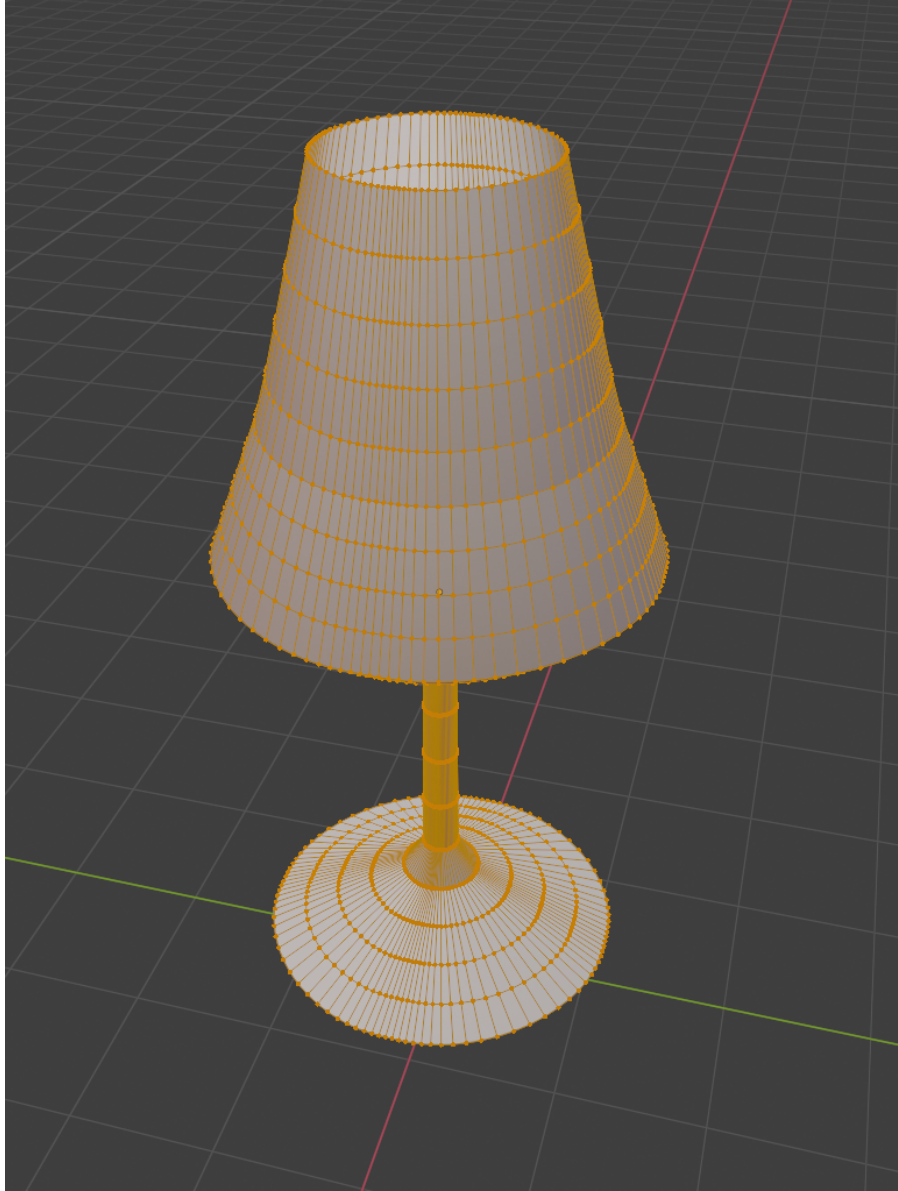
- Triangle mesh is foundational and most widely used data-structure for representation of a shape in graphics
- Triangle mesh consists of many triangles joined along their edges to form a surface
- Triangle is fundamental and simple primitive:
  - All vertices lie in the same plane – **always coplanar**
  - 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

Types of polygons: quads

- In case of four vertices, the polygon is called **quad polygon** (shortly quad).

Vertices	2,304 / 2,304
Edges	4,512 / 4,512
Faces	2,208 / 2,208





Quad mesh

- Often used as a modeling primitive
- Complexity:
  - Easy to create a quad where not all vertices lie on a plane
- In graphics pipeline it is always transformed to triangle.
  - Optionally, In ray-tracing-based rendering plane-ray intersection may be defined and then triangle representation is not needed*.

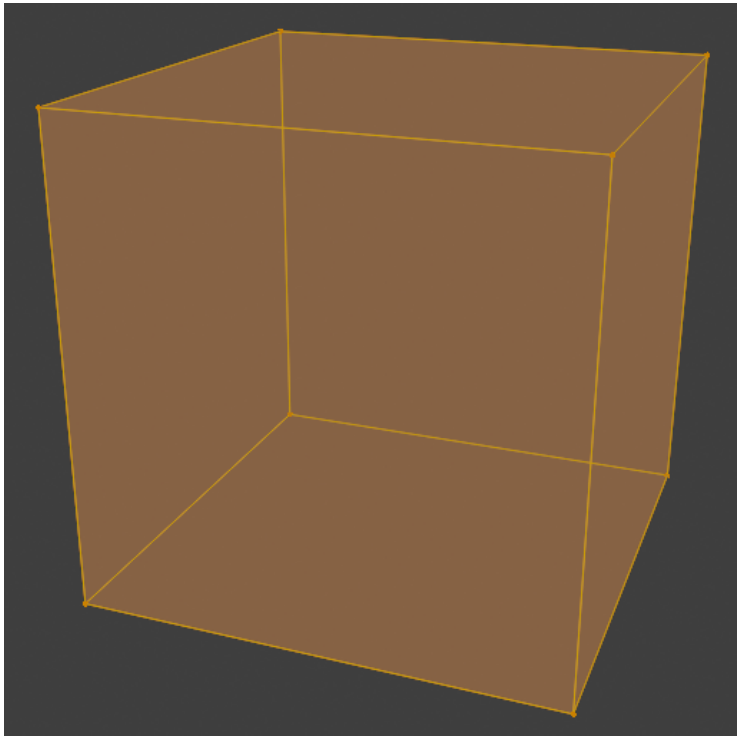
* As we will see, there is always a trade-off between which representation is good for modeling and which representation is good for rendering. Ray-tracing-based rendering can get very flexible with rendering wide representations of shapes but then there is a question if this is feasible to implement and maintain. Often, mapping between different shape representation is researched and used so that on higher level users are provided with intuitive authoring tools and on low level, rendering engine is given efficient representation for rendering process.

Types of polygons: general polygon

- Polygon with more than four vertices is called **general polygon.**
  - Polygons can be convex or concave, and more complex, they may also have holes.
- **Atomic element** – face - of mesh can be any polygon. Common types are: triangle and quad.
- It is a good practice to 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.

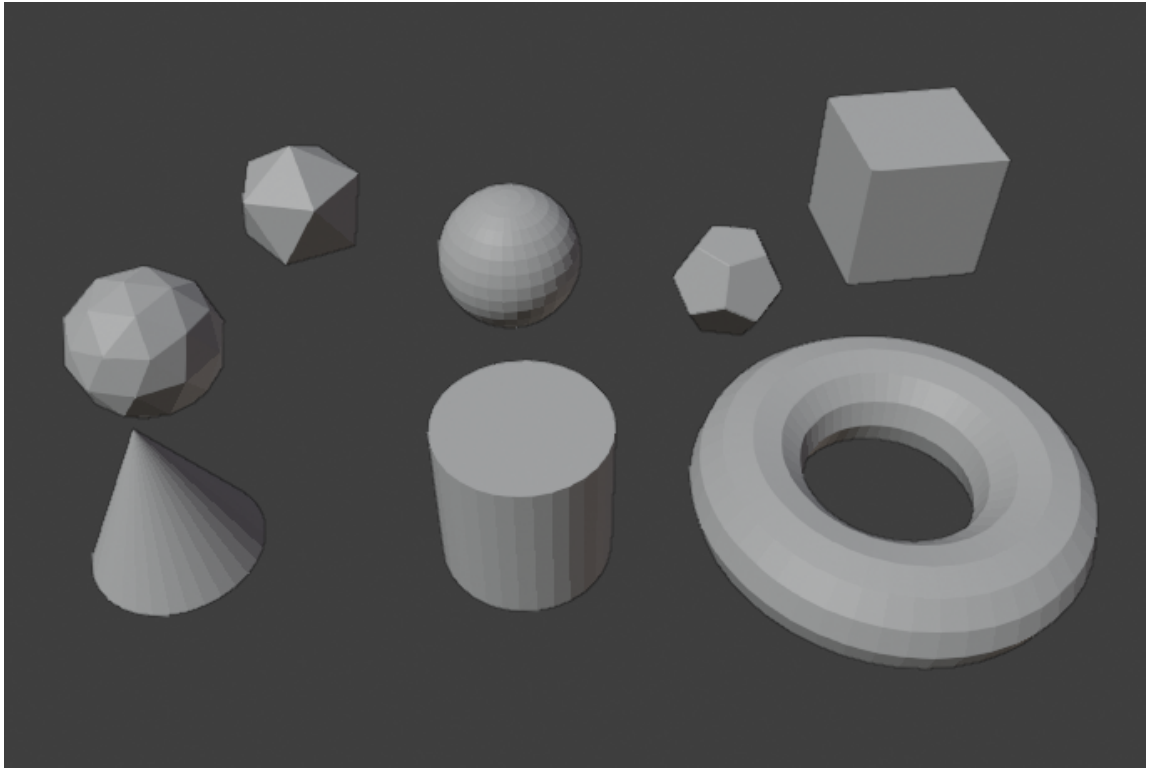
Connectivity

- Complex polygons depend on vertex connectivity information.
- Example: cube
  - Define 8 vertices
  - Define how are those vertices connected to form faces



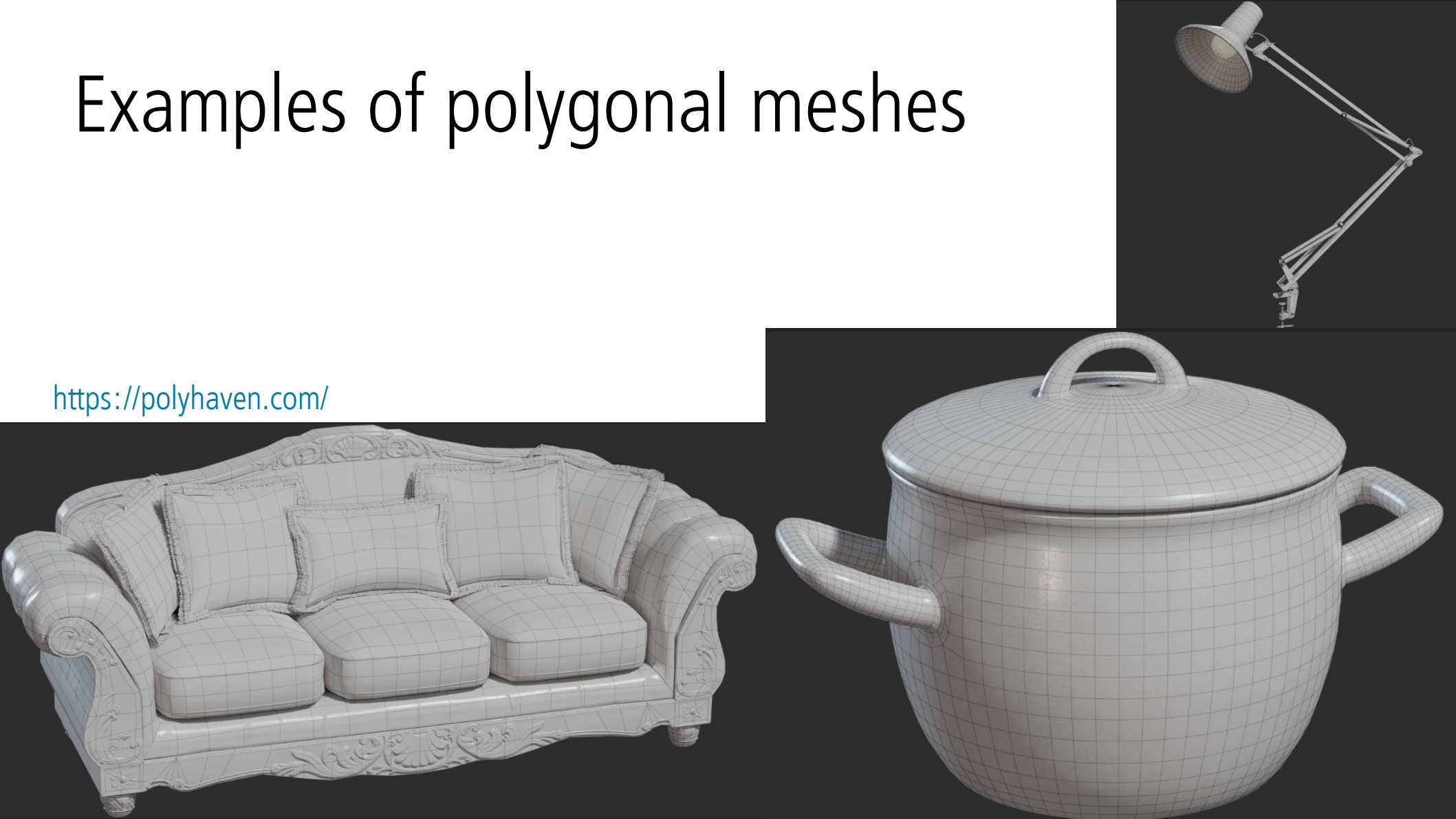
Polygon mesh representation

- Basic information needed for representing and storing polygonal mesh:
  - **Vertex positions** → Geometry
  - **Vertex connectivity** → topology
- 3D polygon mesh: 2D surface embedded in 3D space



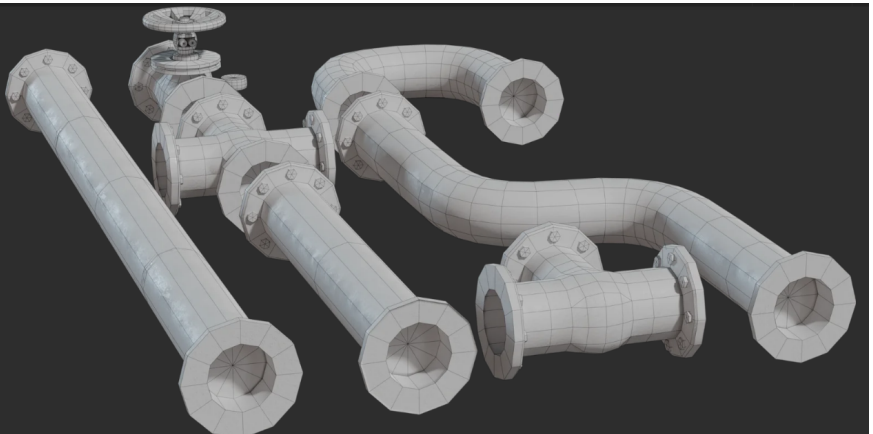
Examples of polygonal meshes

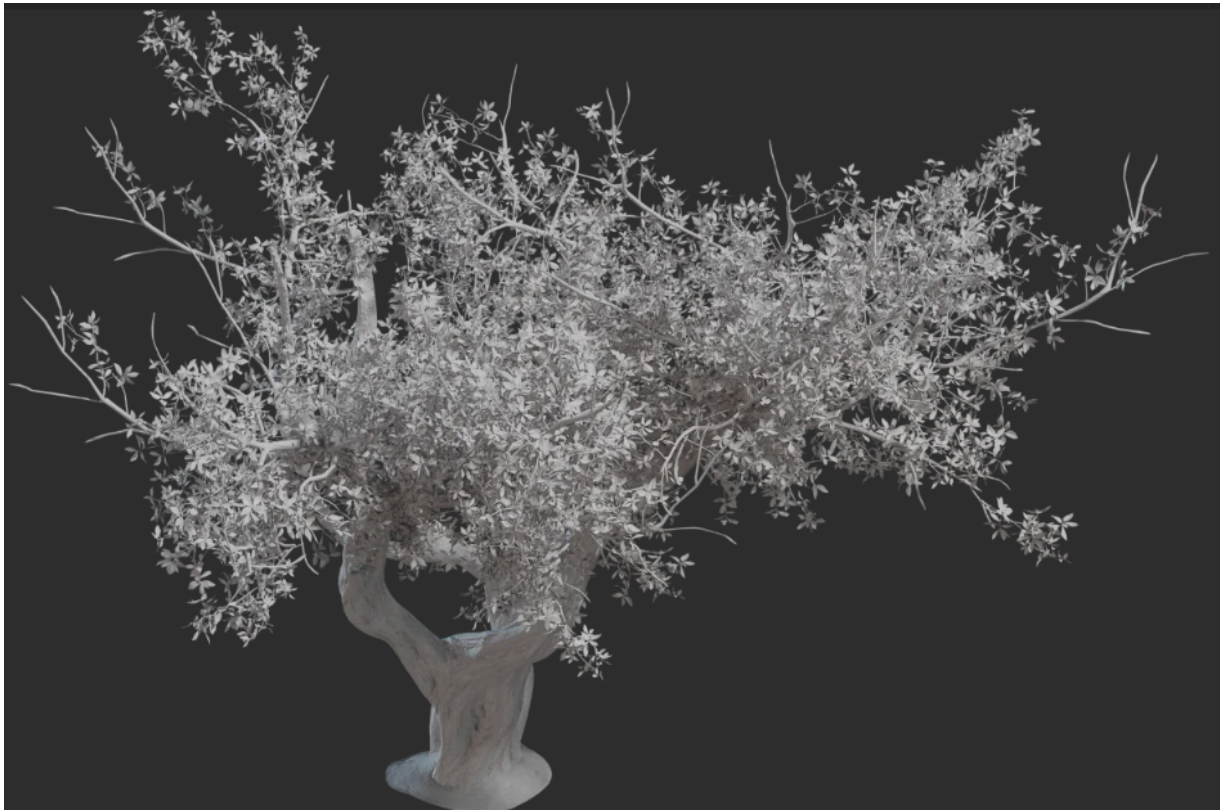
https://polyhaven.com/



Examples of polygonal meshes

https://polyhaven.com/





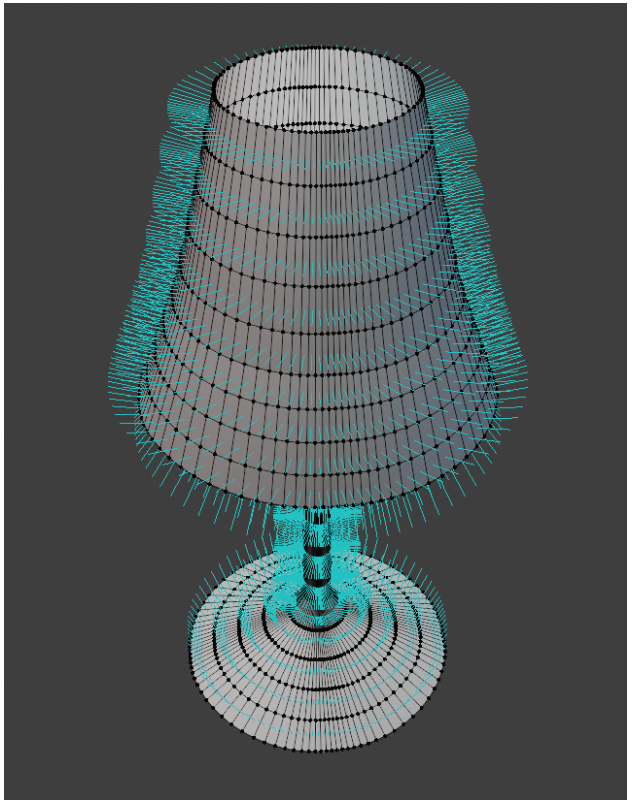
Additional mesh information

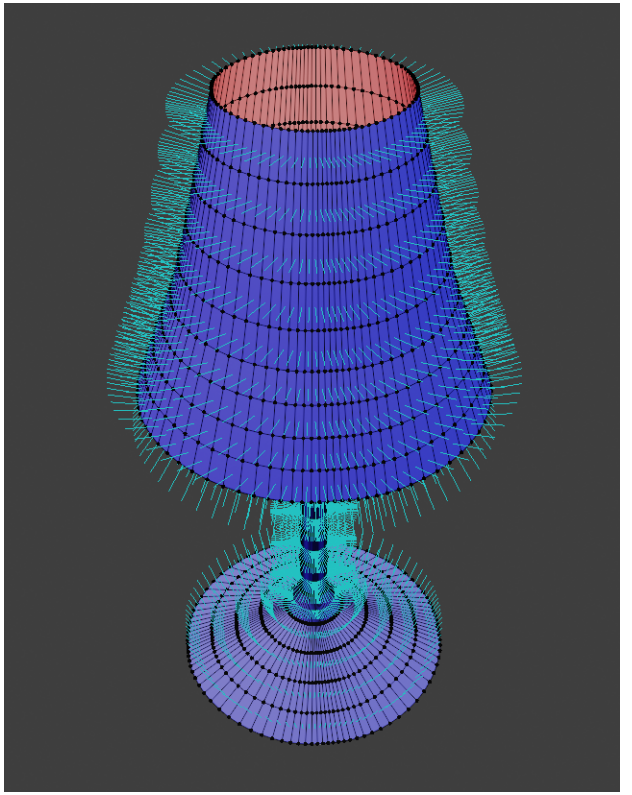
- Besides vertex positions and connectivity information, additional information can be used.
- Those are values that user creates during modeling and which are used for rendering* :
  - Normal*
  - Texture coordinate*
  - Color*
  - Any kind of information that can be encoded and used for rendering: weights, temperature, etc.
- Representation and storage of models is highly developed topic. For now we represented ideas on which any professional solutions build on.

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

Additional mesh information: Normals

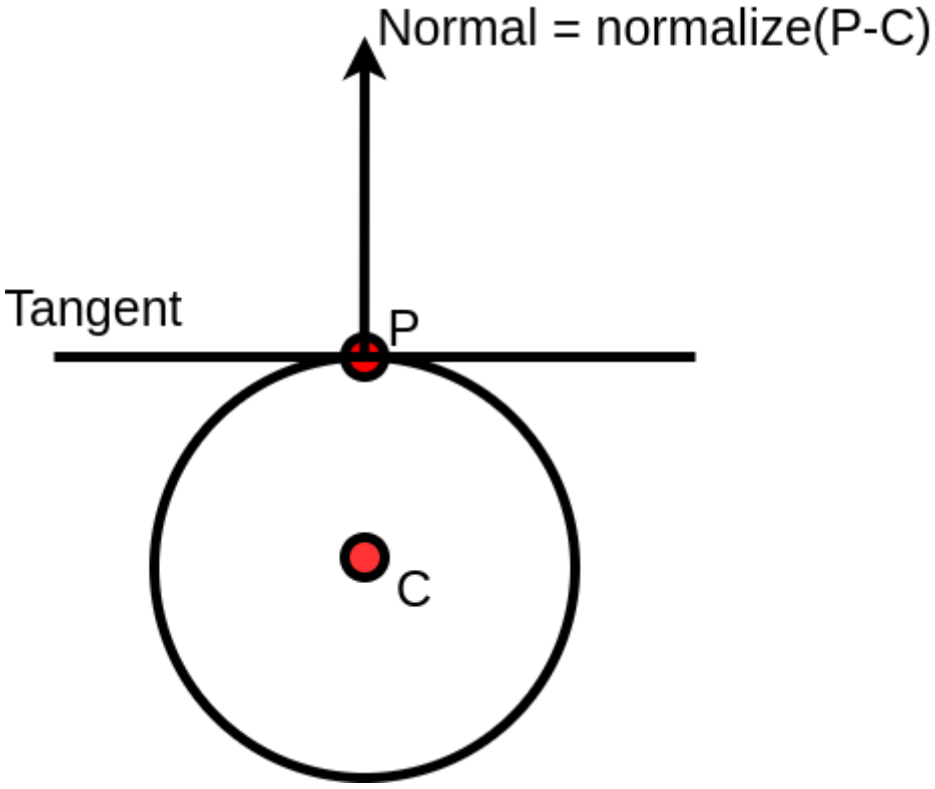
- Orientation of surface in each point is determined by normal vector.
- Normal vector is **core information for rendering (shading) and modeling**.
- Mesh normal can be defined per:
  - face
  - per vertex





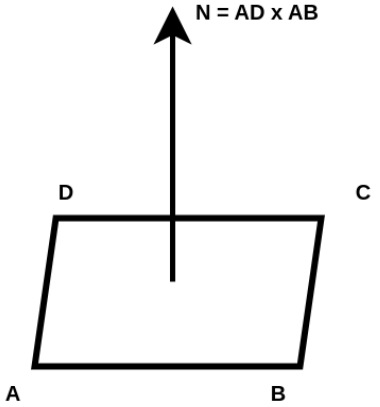
Computing normals

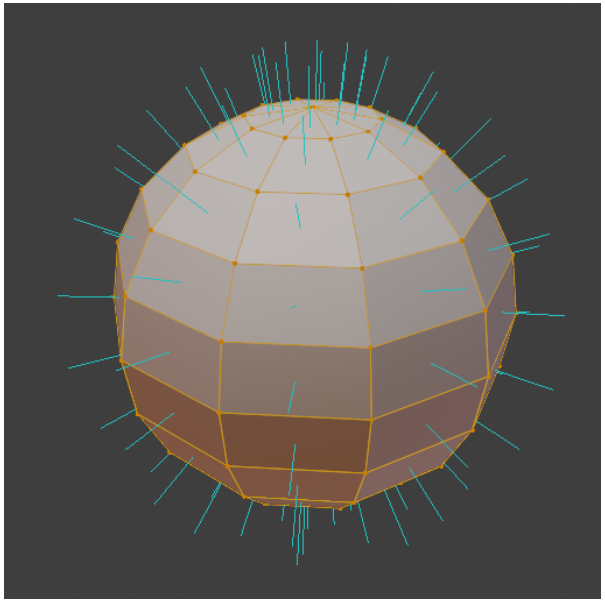
- Normal in surface point is **vector perpendicular to tangent** in that surface point
- Computation of normals depends on shape representation
  - Sphere normal
  - Mesh face normals can be calculated using triangle or quad polygon edges.

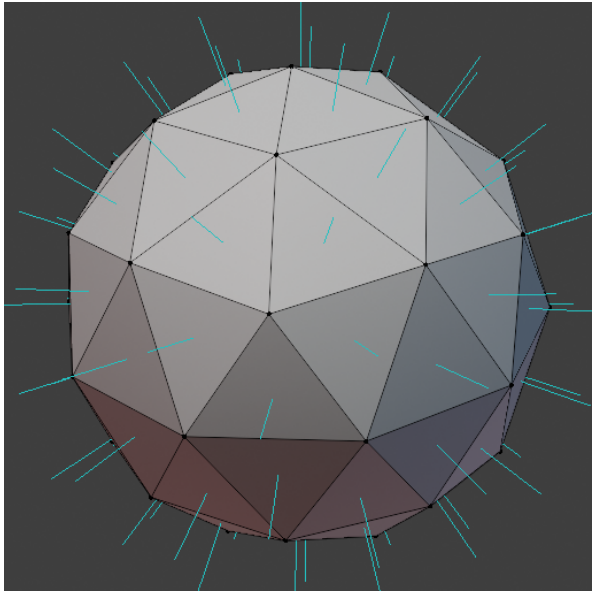


Normals: per face

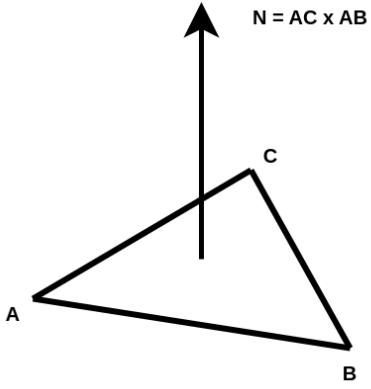
- Cross product between triangle or quad edges
- Winding order of vertices defines orientation of normal





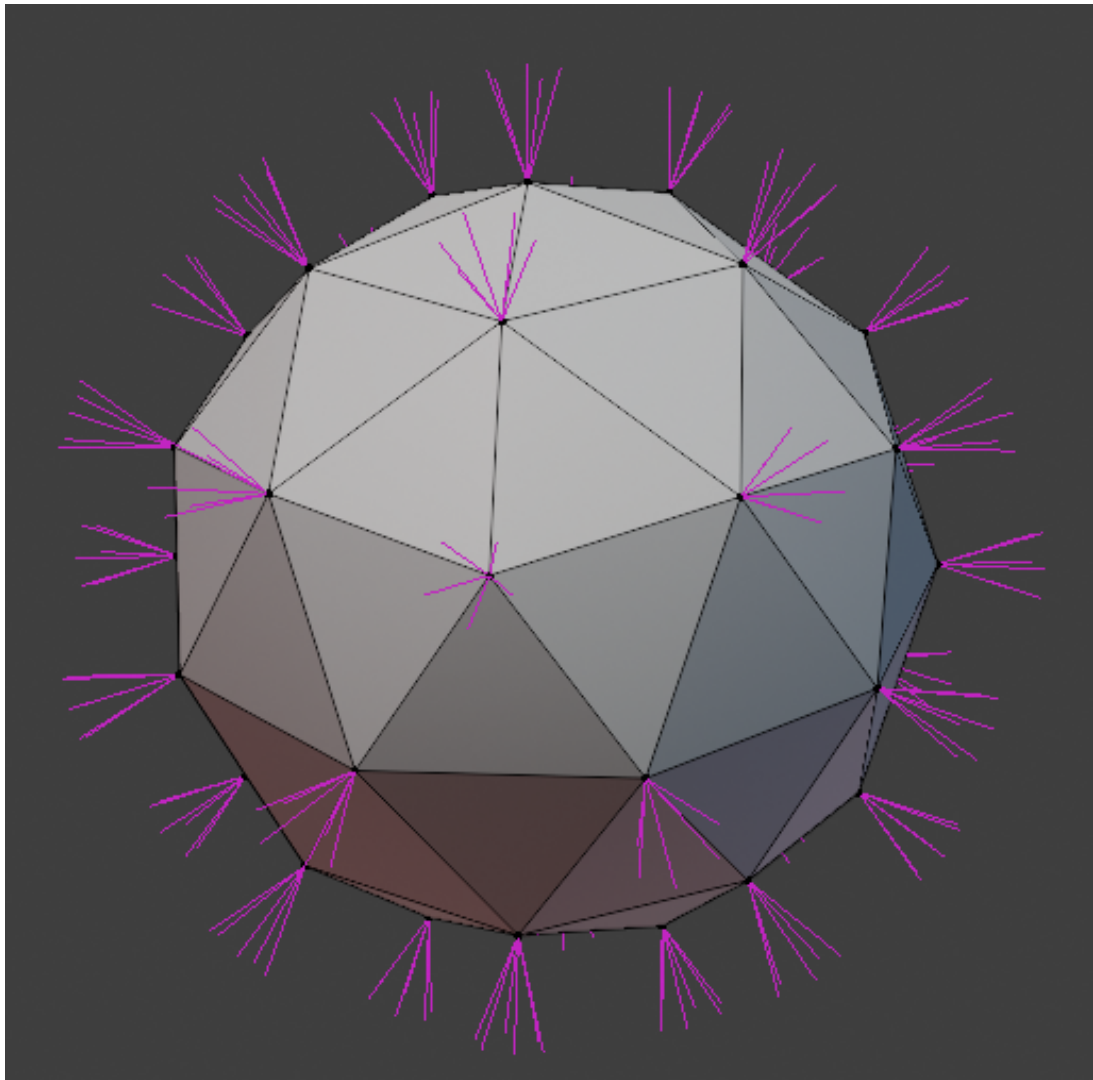


```
N = (B-A).crossProduct(C-A);
```



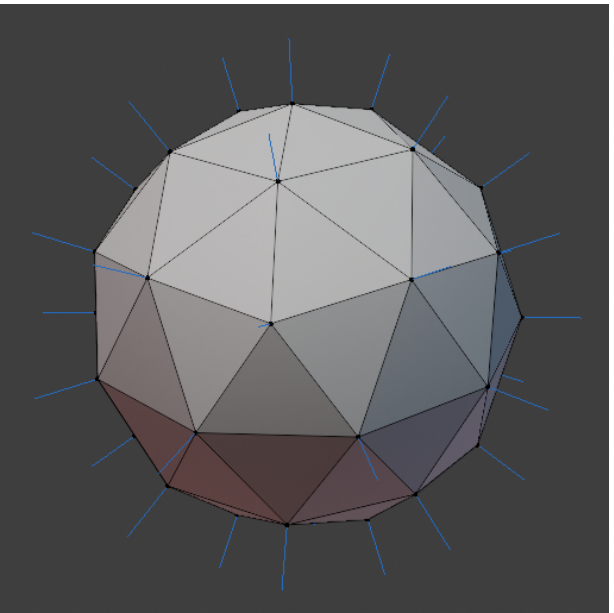
Normals: per vertex

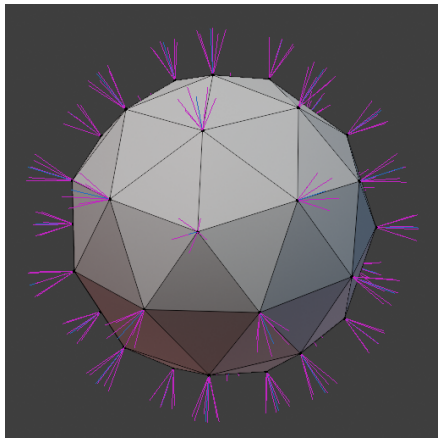
- If normals are defined per vertex, then multiple normals, one for each face, can be stored per vertex.

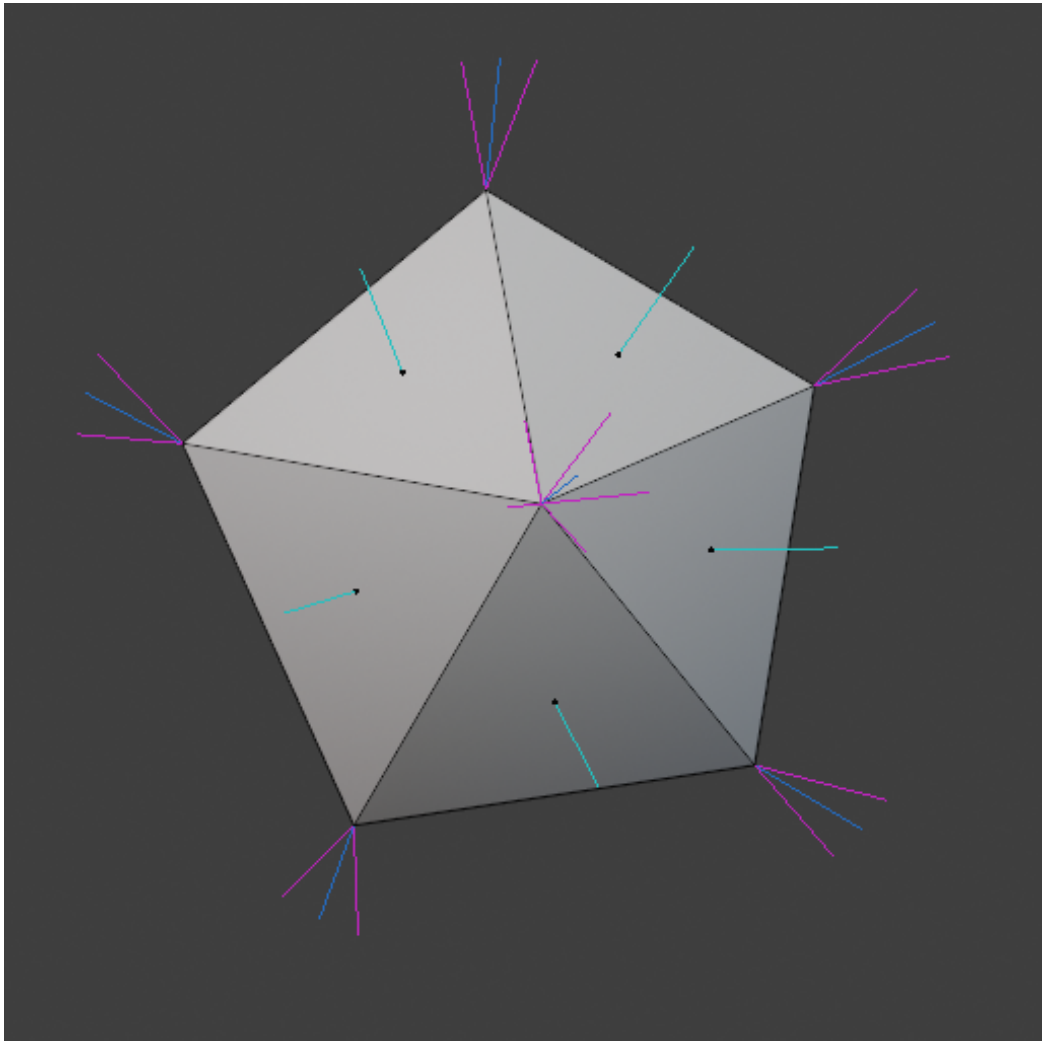


Normals: per vertex

- One normal per vertex is calculated as:
  - For each face:
    - Calculate face normal
    - Add normal to each connected vertex normal
  - For each vertex normal:
    - normalize

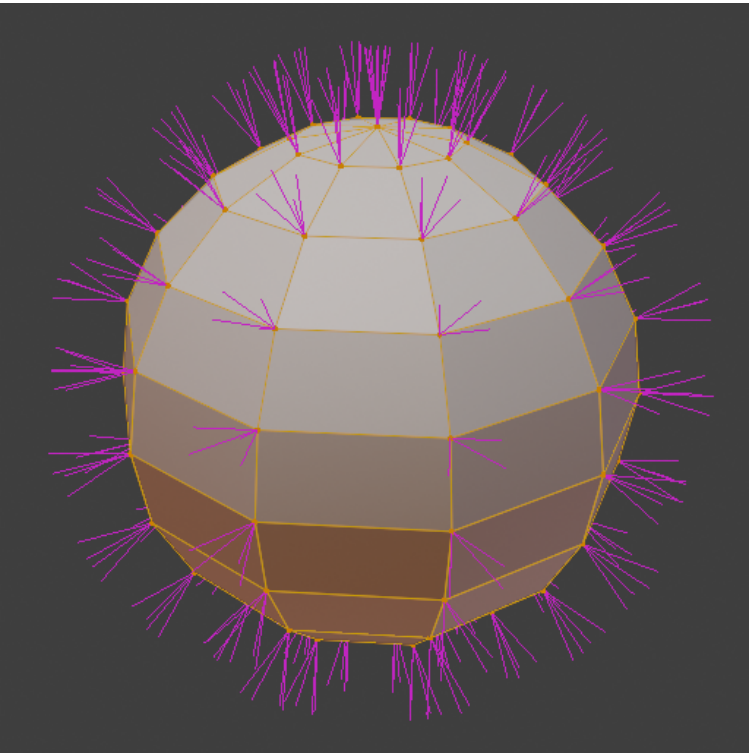


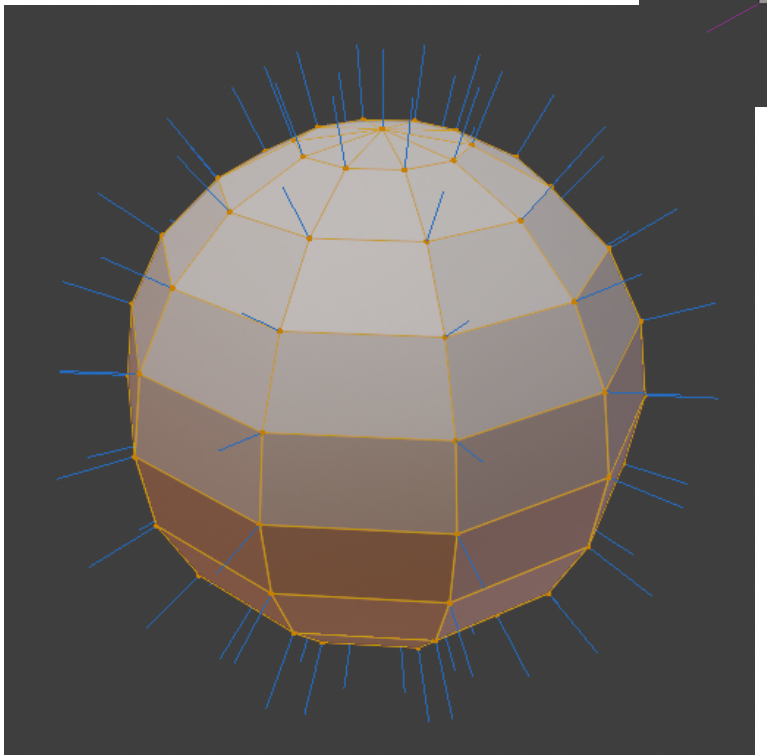


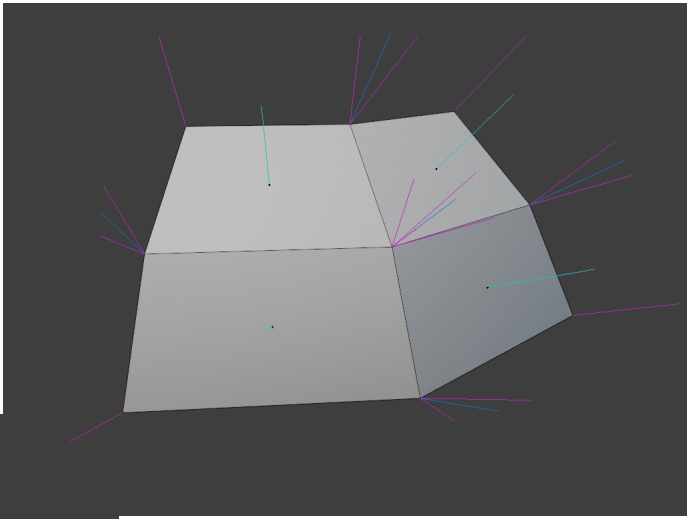


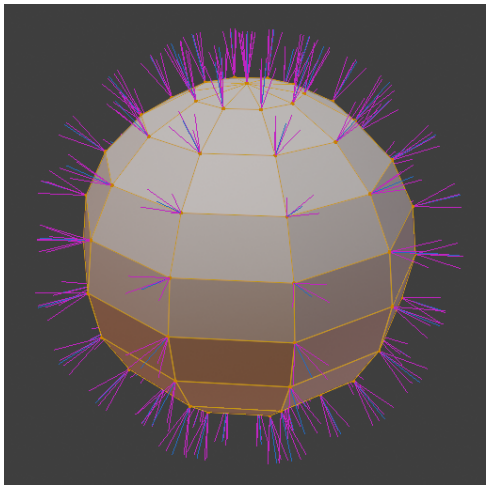
Normals: per vertex

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



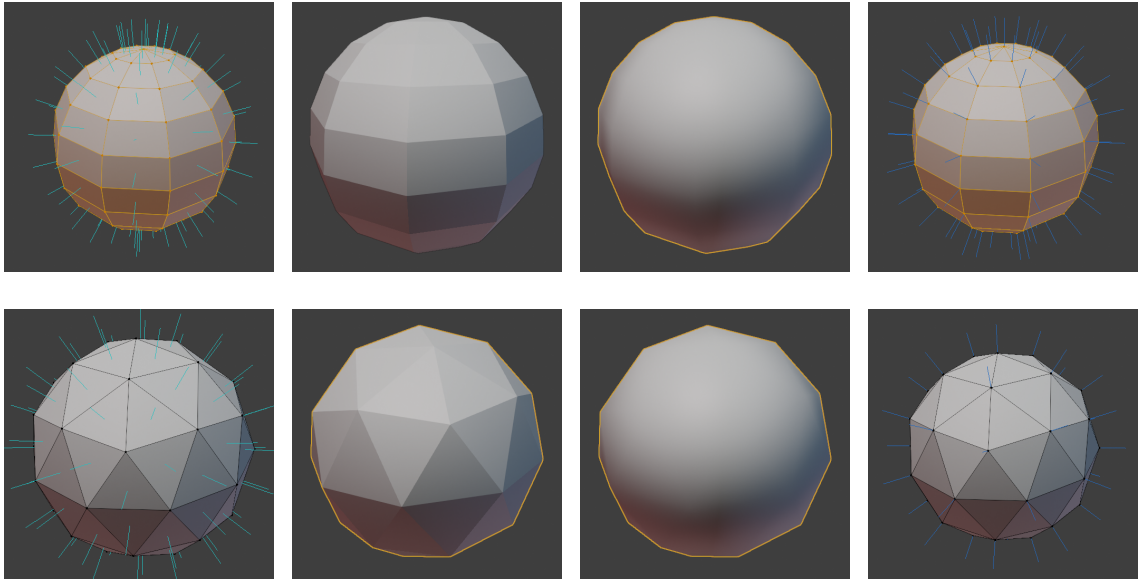






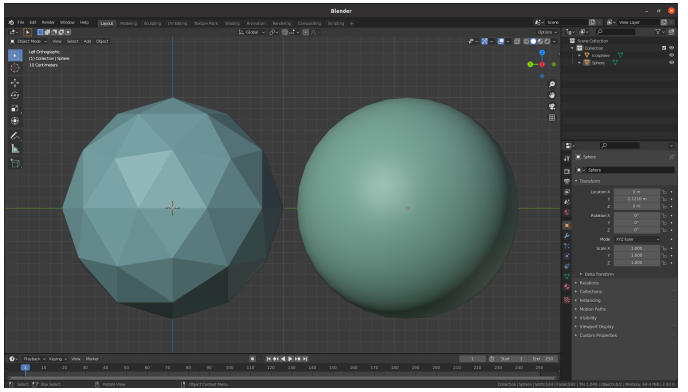
Using normals: shading

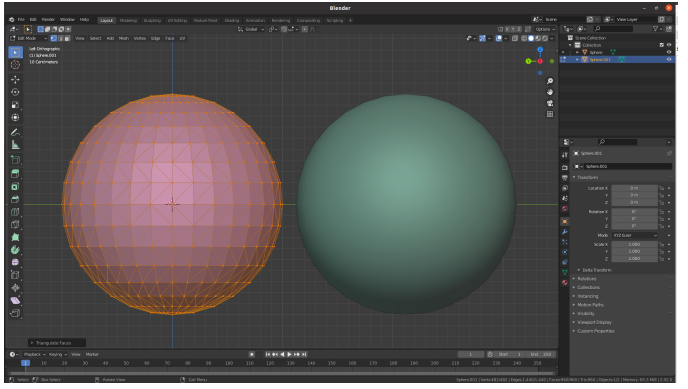
- Appearance of object depends both on shape and material.
  - Normal is main shape information used in shading; appearance calculation
- **Gouraud shading**: normals defined per vertex are linearly interpolated over triangles (using **barycentric coordinates**).
  - Geometry is not changed: note faceted silhouette
  - Illusion of smooth surface is created during shading



Note: approximating curved surfaces with polygons

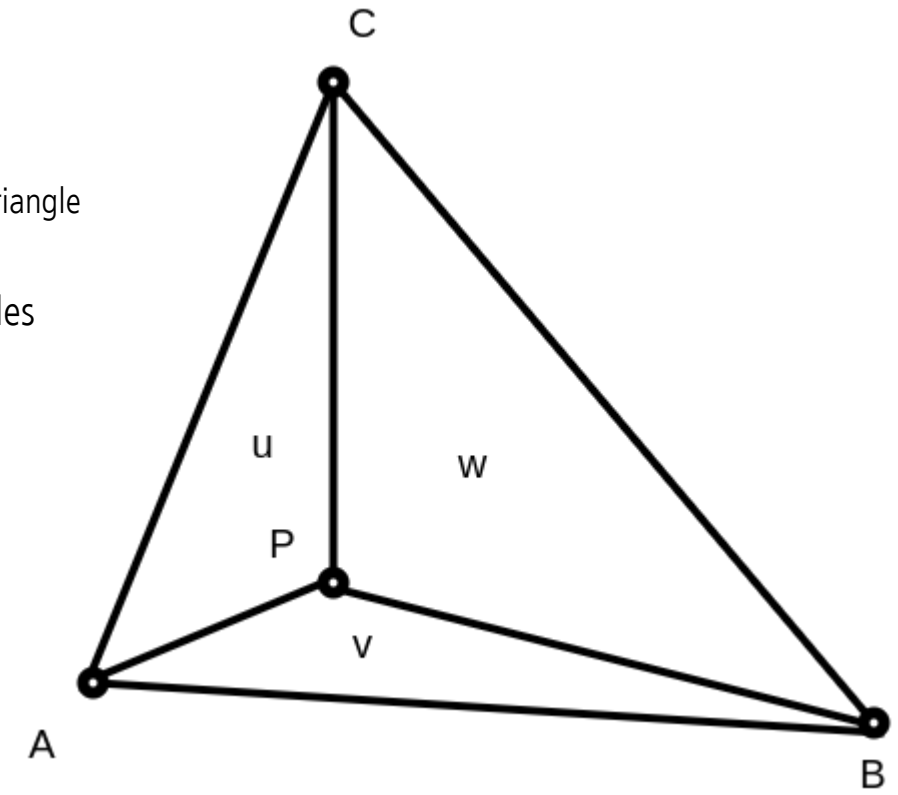
- Example: sphere vs icosahedron
  - Each point on icosahedron is close to point of sphere
  - Each normal vector of icosahedron is close to vector normal of the sphere in the same point. But, function that assigns normals to the sphere is continuous while for icosahedron is piecewise constant → this influences reflection of light!





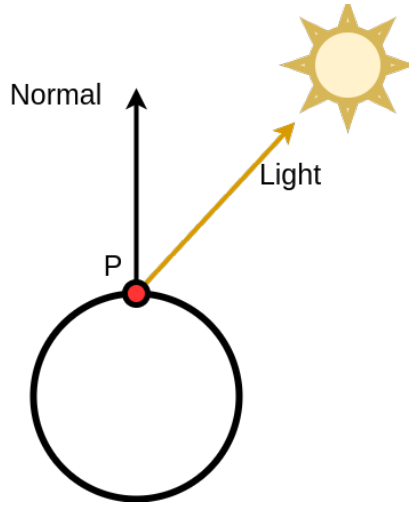
Triangle mesh: barycentric coordinates

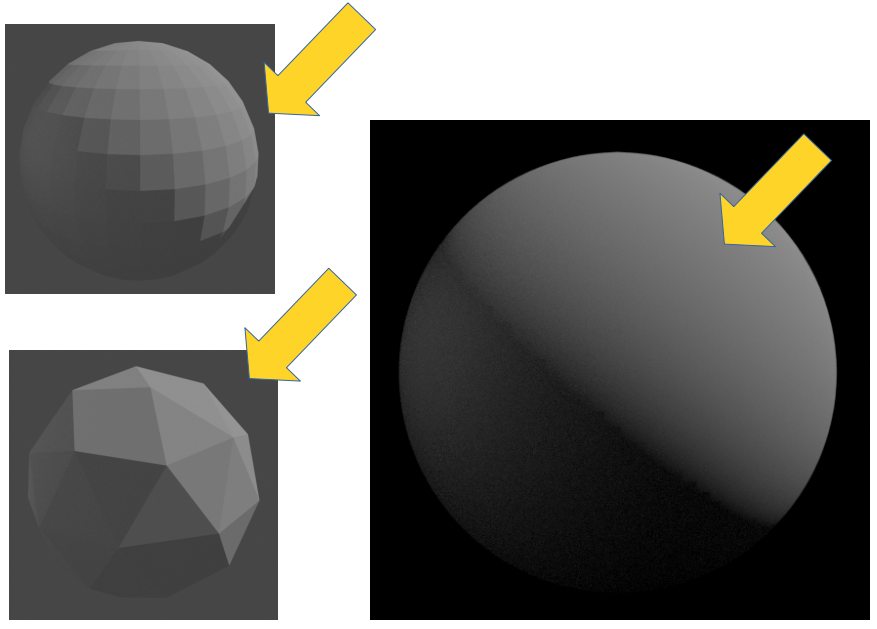
- Express position of any point located on triangle with three scalars.
  - P = uA + vB + wC, where A,B,C are triangle vertices
  - u,v,w, are barycentric coordinates, where u + v + w = 1 and
  - 0 <= u,v,w <=1, otherwise P is outside of triangle. If equal then P is on triangle edge.
- aka areal coordinates: 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



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(θ) = dot(N, L)



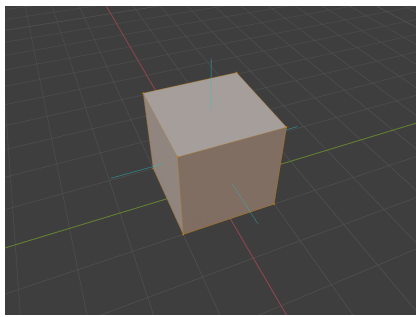
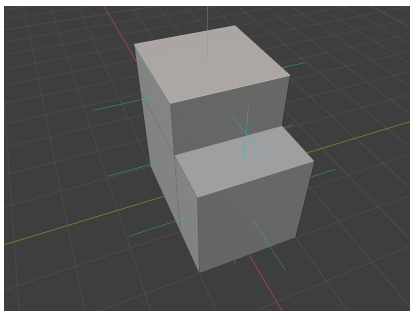
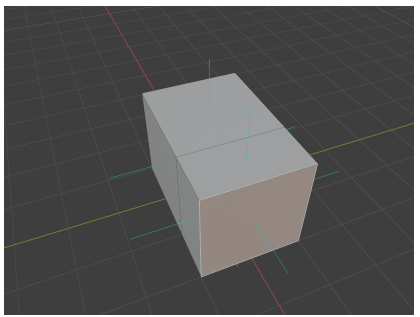
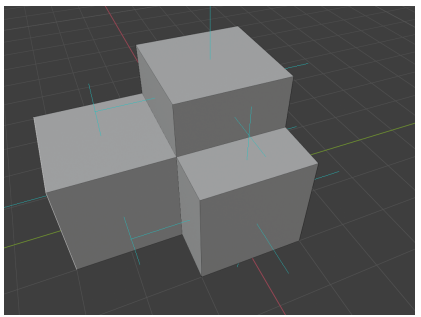
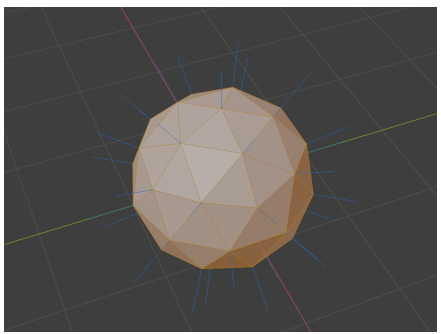
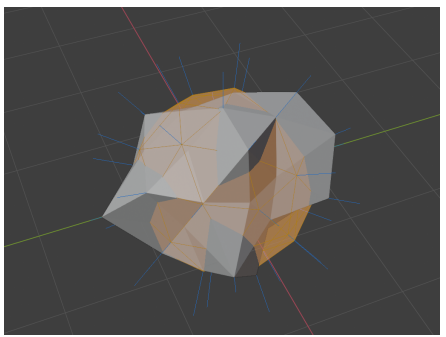
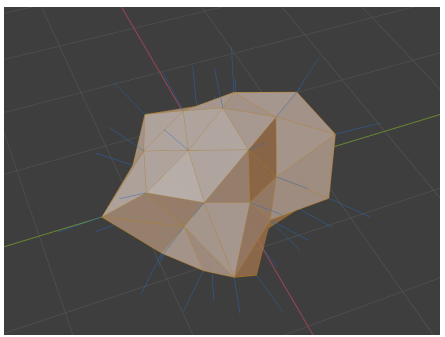


Using normals: shading

- Generally, normal vector is perpendicular to surface, but it can be perturbed so it is not perpendicular.
  - This is trick which is used for shading TODO

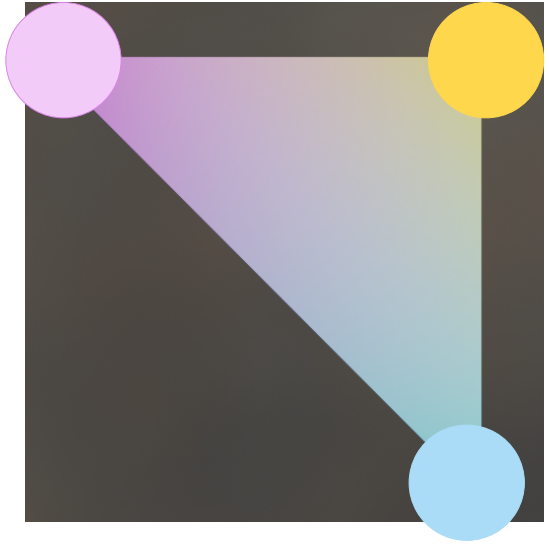
Using normals: geometric manipulation

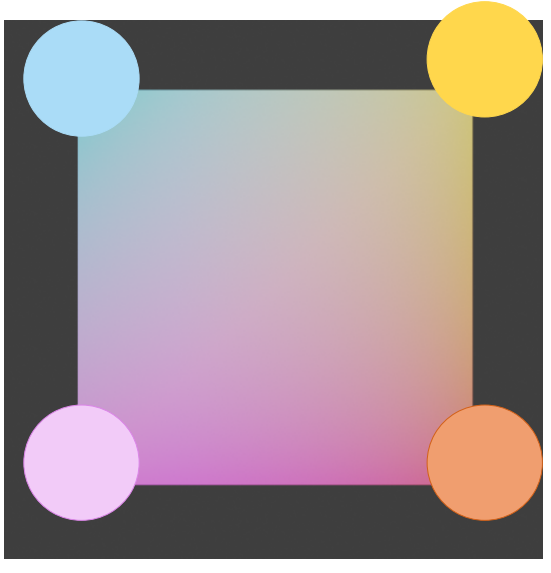
- One of the basic modeling operations is **extrusion**
  - This operation uses normal information for moving the face
- Proceudural modeling utilizes normal direction for **displacing mesh vertices**
- These are only some examples of normal vector usages. The point is to highlight its importance.

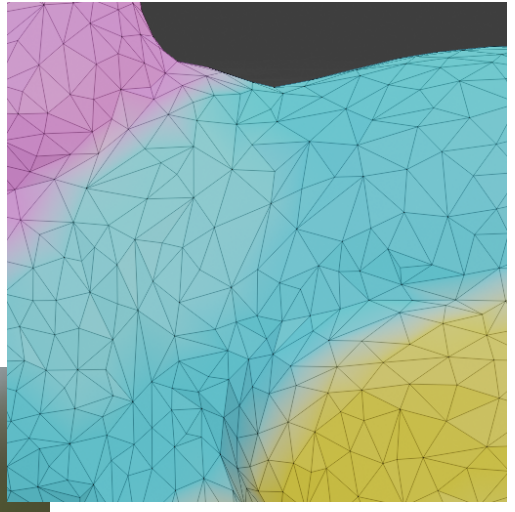








Additional mesh information: vertex colors

- 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

- TODO

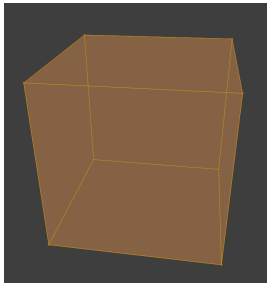
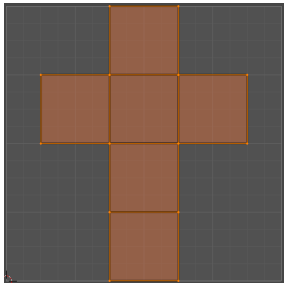
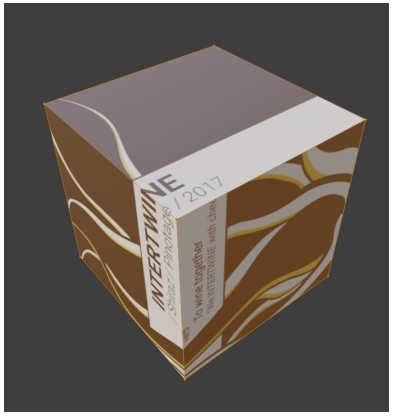
Additional mesh information: texture coordinates

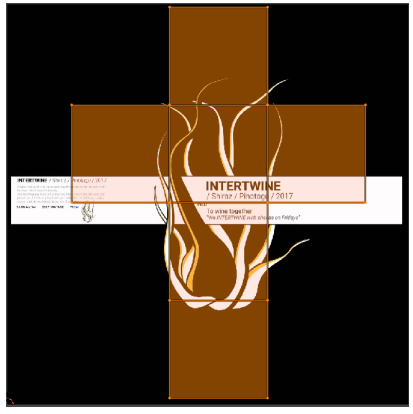
- To add more details on object, often images and procedural patterns are used → **textures**.
- The problem of applying texture image on 3D object is often quite complex than on flat plane.
  - A a way of mapping a 2D image/pattern to 3D shape can be done by "unwrapping" mesh onto 2D plane.

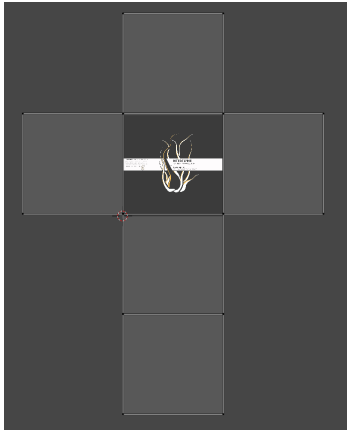
https://polyhaven.com/a/wine_bottles_01

Mesh unwrapping: intuition

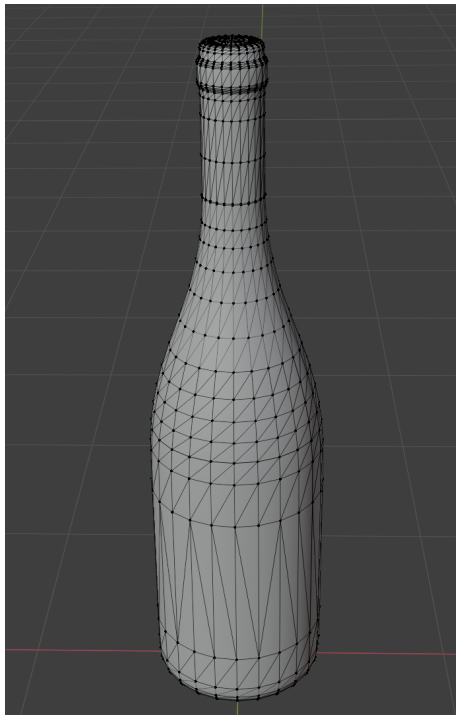
- Mesh is unwrapped into 2D plane we can imagine it lies in 2D coordinate system.
- Vertices of unwrapped mesh now have coordinates in this 2D coordinate system (u,v) → **texture coordinates***
  - (u,v) are in [0,1] range
  - Unwrapped faces can be manipulated, but then texture might get deformed
 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

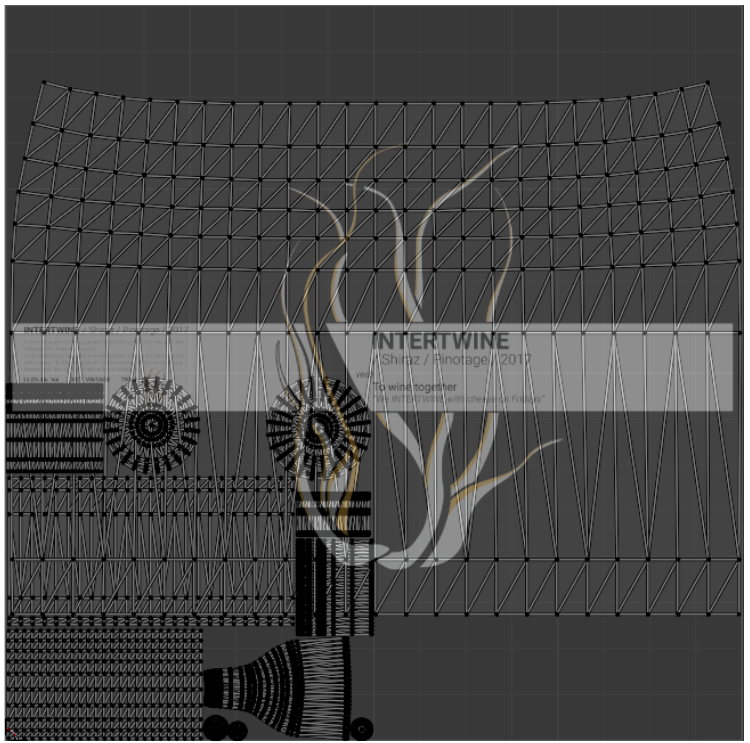








Texture coordinates: mesh unwrap





* 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.

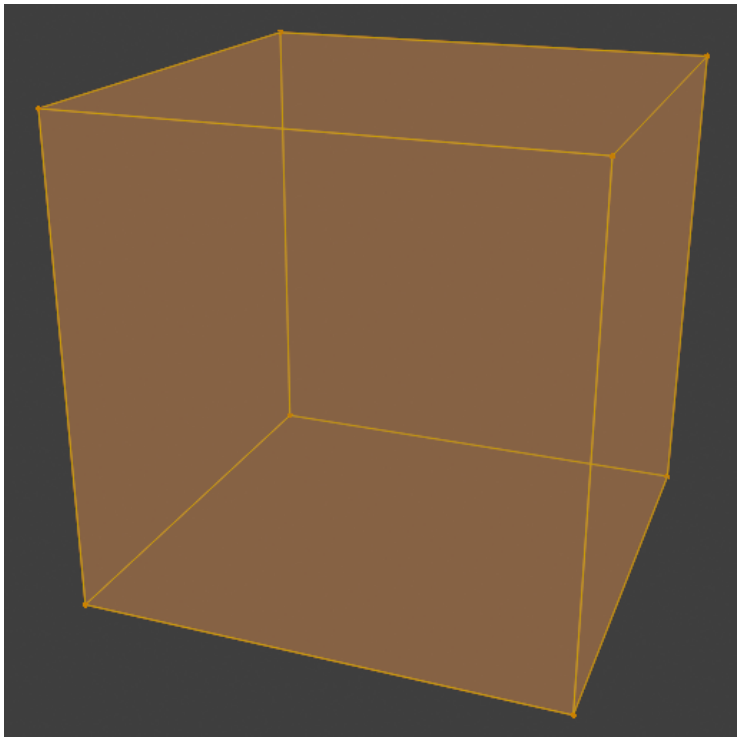
Mesh polygons: storing

Important properties of mesh representation

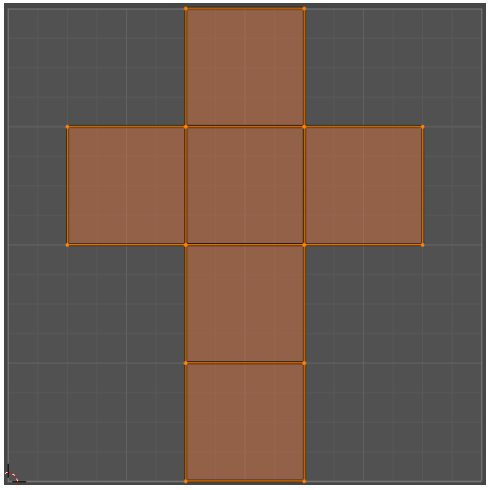
- Efficient topology traversal
- Efficient use of memory
- Efficient updates

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

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

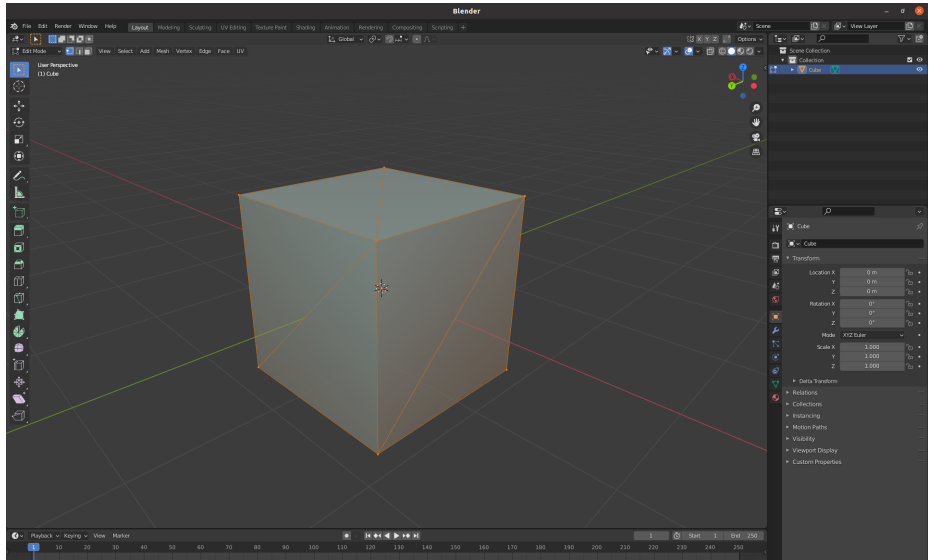
- Mesh data is often transferred between different modeling, rendering and interactive tools.
- Interface between different tools and specification of how mesh should be stored is defined by standards*: https://renderman.pixar.com/resources/RenderMan_20/ribBinding.html
- Different implementations of 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.

Example: OBJ file format

OBJ file format. 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




```
# 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
```

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

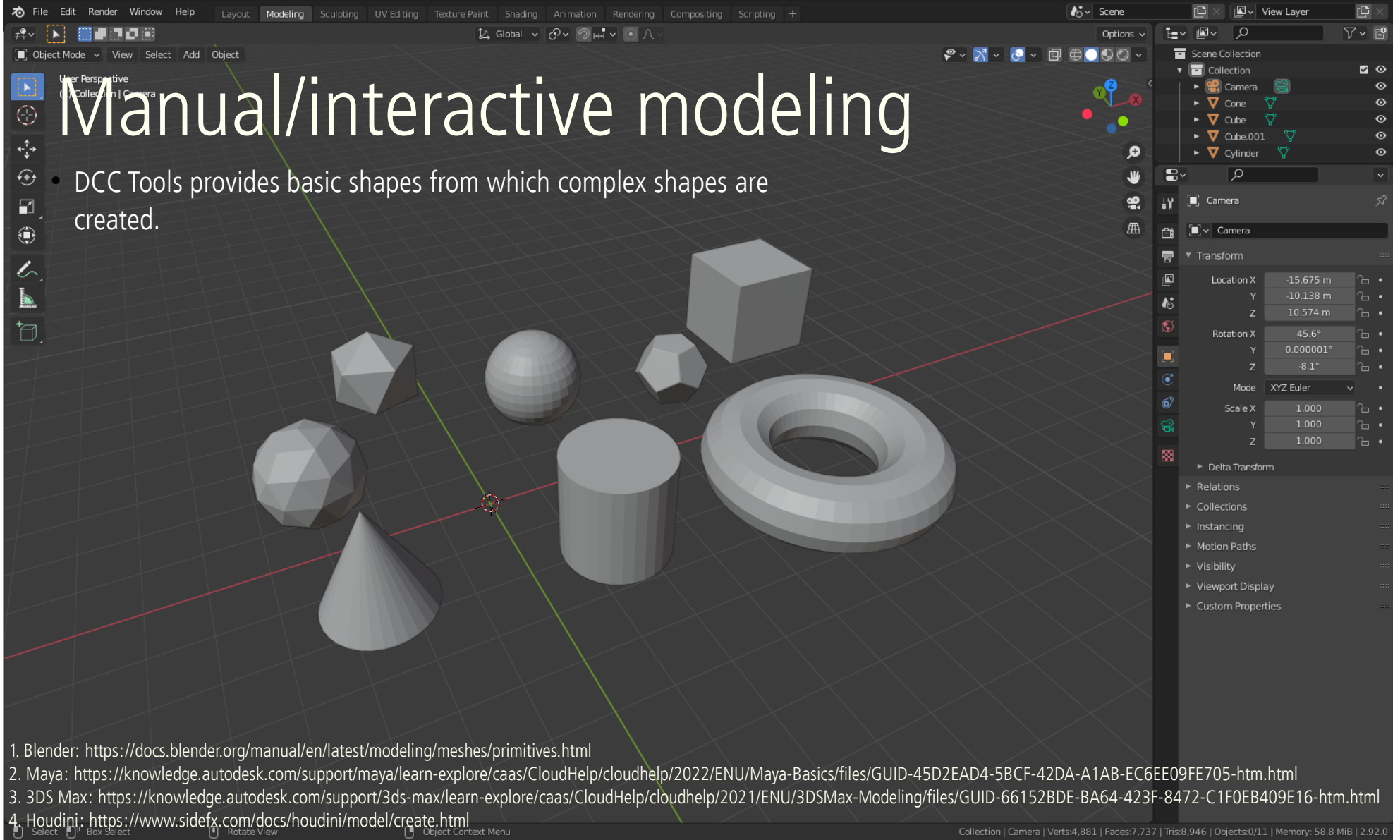
Practical note: units

- 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: acquisition and modeling

Manual/interactive modeling

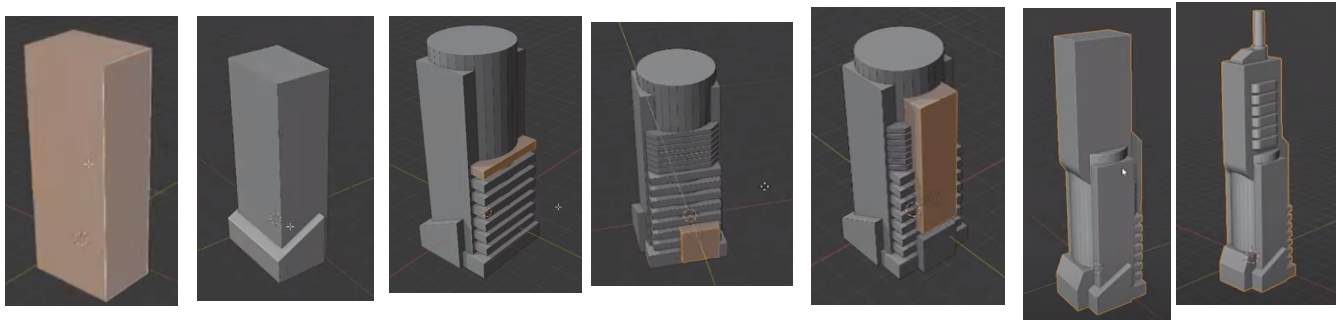
- DCC Tools provides basic shapes from which complex shapes are created.



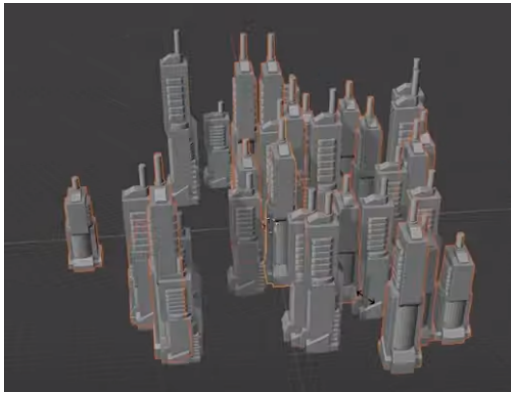
1. Blender: https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html
2. Maya: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2022/ENU/Maya-Basics/files/GUID-45D2EAD4-5BCF-42DA-A1AB-EC6EE09FE705-htm.html
3. 3DS Max: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2021/ENU/3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

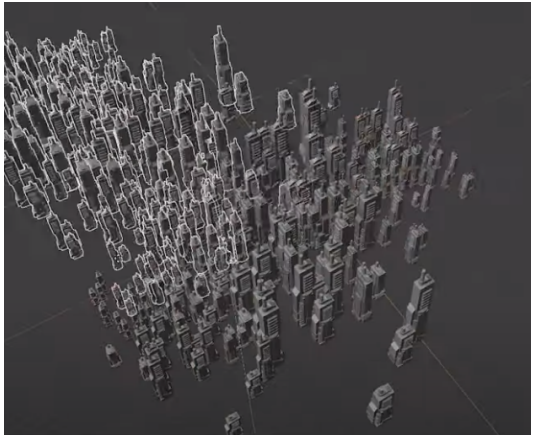
Practical tip: complex shapes from base shapes

- 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



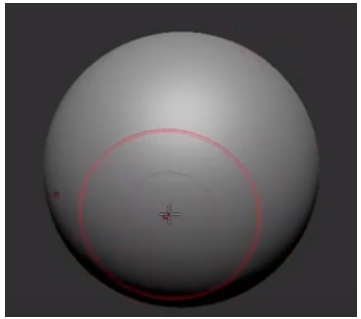
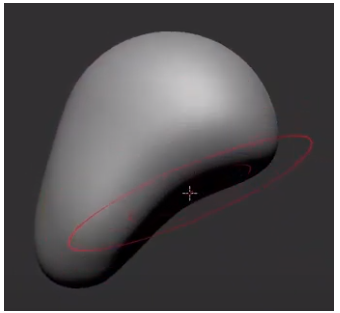
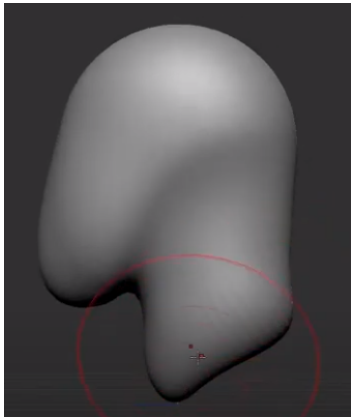
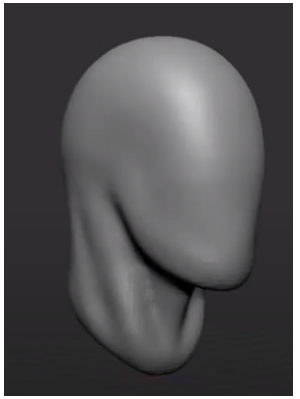
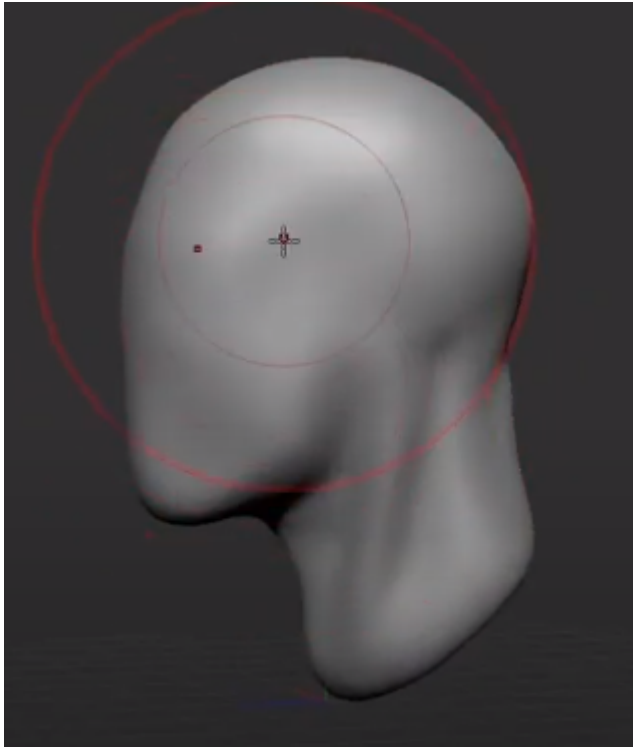


1. http://www.thedrawingwebsite.com/2015/02/18/practicing-your-draw-fu-forms-forms-are-like-sentences/

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

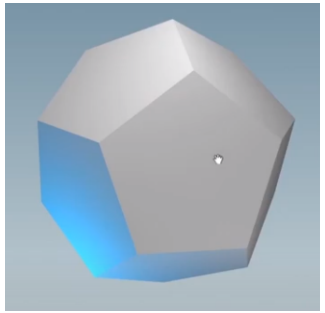
Practical tip: complex shapes from base shapes

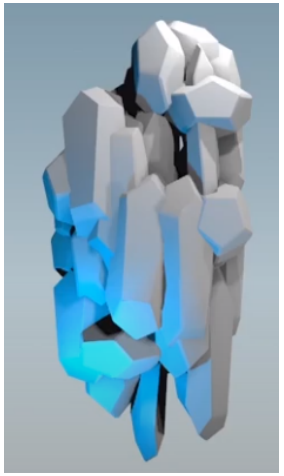
- Sculpting base shapes.

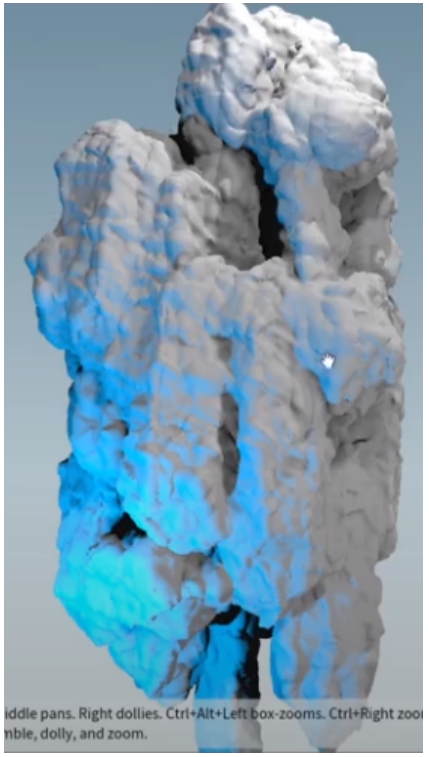






https://www.youtube.com/watch?v=p4Y1hVCMMXw&ab_channel=RealEnay

Practical tip: complex shapes from base shapes



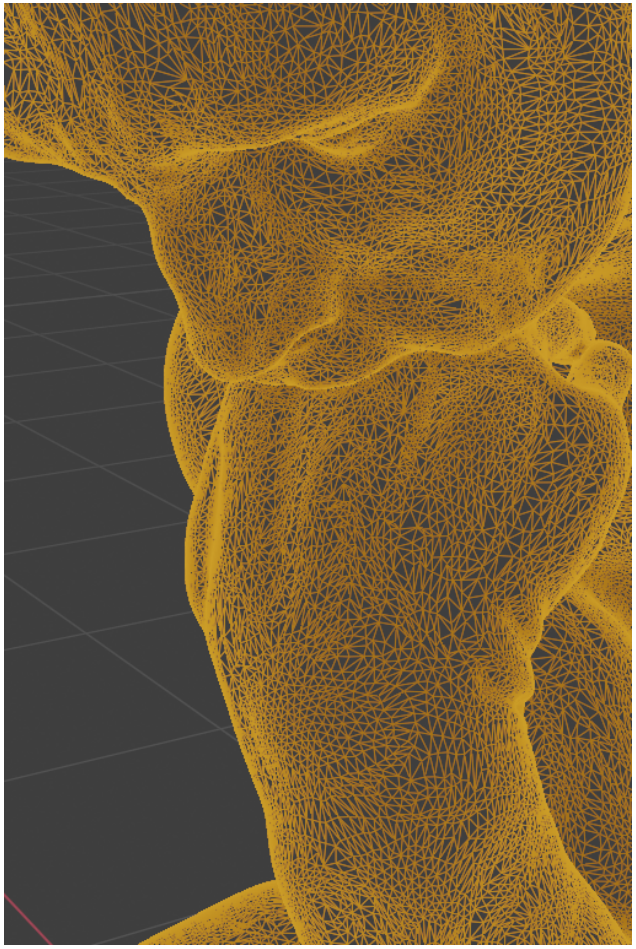


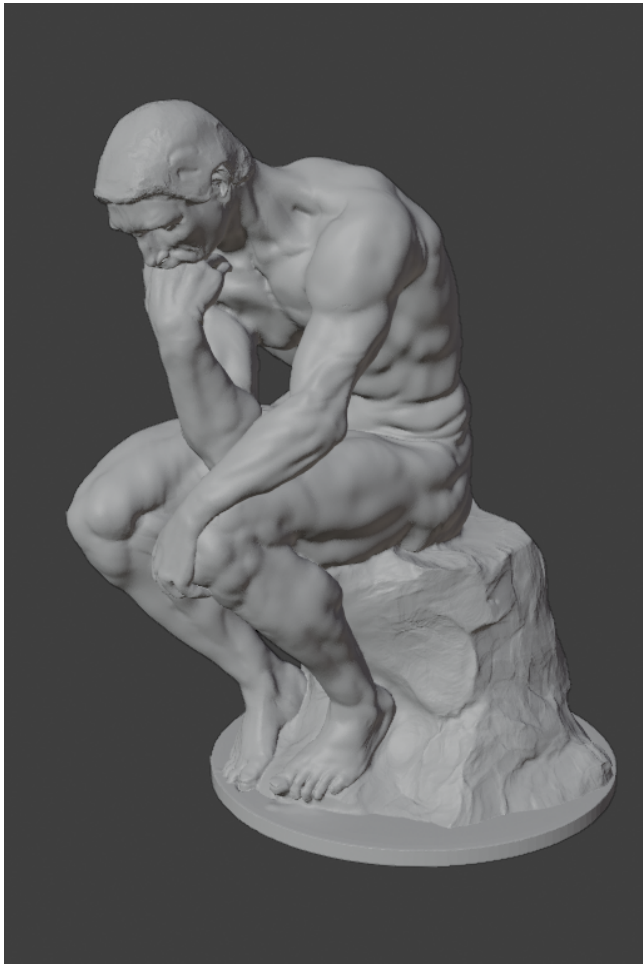


- Procedural modeling

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

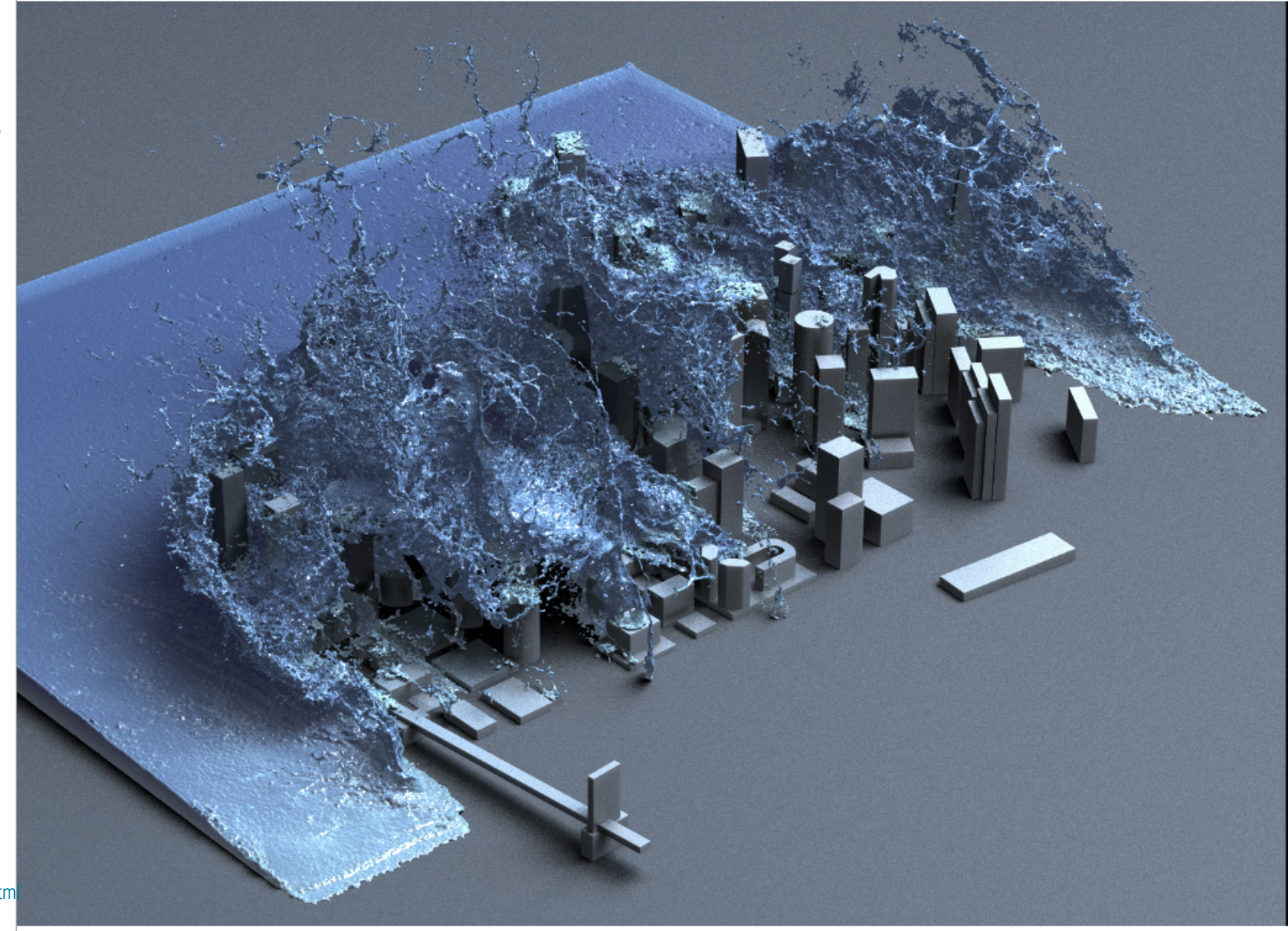
Scanning and photogrammetry





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

Simulations



https://people.csail.mit.edu/kuiwu/gvdb_sim.html

Mesh polygons: concluding

Important properties of meshes

- Goal: not to go deep into definitions but rather to verify properties using simpler methods
- **Mesh boundary**: formal sum of vertices
- **Closed mesh**: mesh boundary is zero. Required for defining what is "inside" and "outside" by winding number rule
- **Manifold mesh**: each vertex has arriving and leaving edge
  - Manifolds are desired since it is easy to work with them (both manually and algorithmically)
  - Smooth vs not smooth manifolds (e.g., cube)
  - Self-intersecting meshes are not manifolds
  - In graphics we generally use polyhedral manifolds
- **Oriented vs unoriented meshes**
  - We use oriented meshes so that boundary can be defined

<IMAGES: DEPICT IMPORTANT PROPERTIES!>

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** library for mesh processing
  - Mesh curvature
- **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

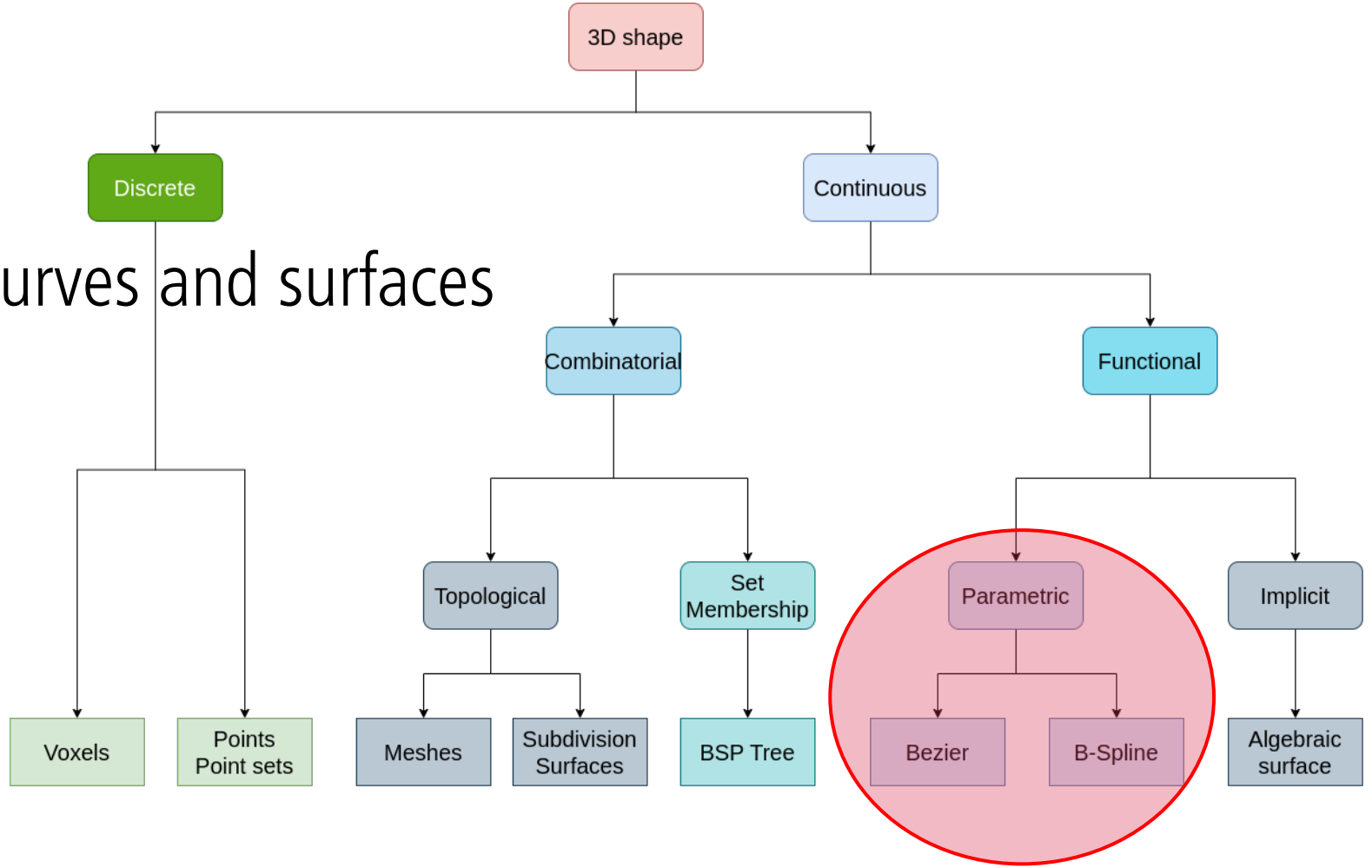
Deeper into topic

- **CGAL** is advanced geometry processing library
- 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:
  - Tessellation and triangulation
  - Consolidation, mesh reparation
  - Representation
  - Simplification, level of detail
  - Compression

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

Parametric curves and surfaces



Another shape representation

- Representing surface using mesh is the most used and widespread option for both authoring and transfer.
  - Triangle mesh and thus triangle is basic atomic rendering primitive for GPU graphics pipelines and most ray-tracers.
- However, objects made in modeling systems can have many underlying geometric descriptions.
  - Different geometric descriptions enable easier and efficient modeling and representation of shapes on the user side.
  - On the rendering side, all higher-level geometrical descriptions are evaluated as set of triangles and then used.

<IMAGE: show THE CONCEPT OF USER AND RENDERING SIDE AND HOW OBJECTS CAN BE REPRESENTED>

Applications

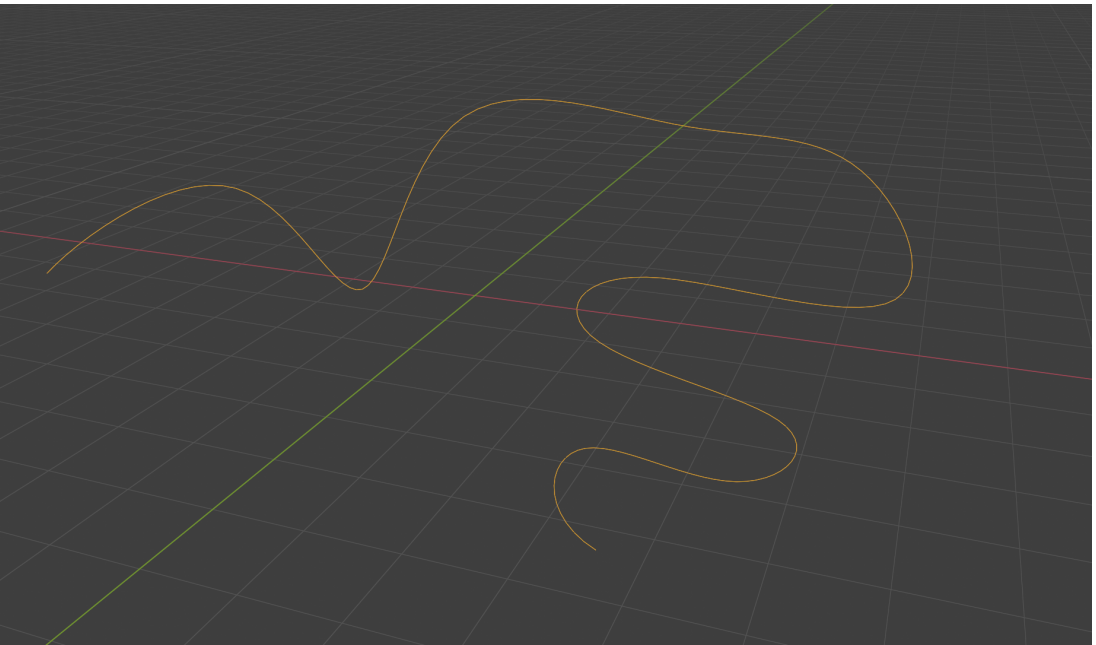
- Design and modeling of smooth surfaces: cars, ships, airplanes, etc.
- architecture,
- Nature,
- Camera paths,
- Vector graphics,

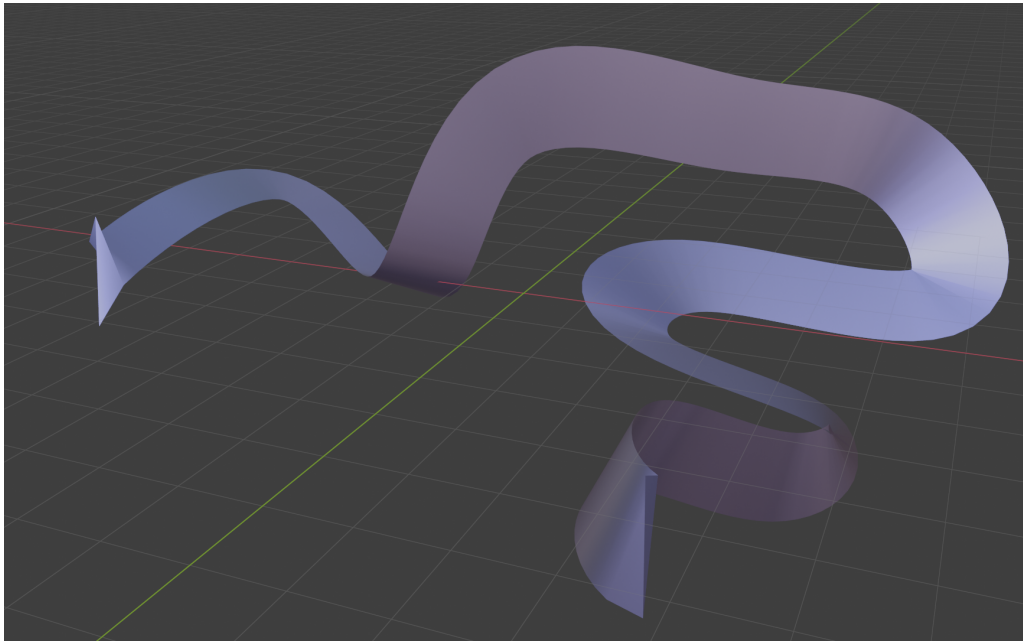
Parametric surfaces vs Mesh

- Parametric surfaces are one of alternative geometric representations that have certain advantages over meshes in certain scenarios.
- Some advantages of curves and curved (subdivision) surfaces are:
  - They are represented by equations and thus have more compact representation than meshes (less memory for storing and transfer) and less transformation operations are needed
  - Since they are represented by equations they provide salable geometric primitives – geometry can be generated on the fly by evaluating the equations (Analogy: vector and raster images)
  - They can represent smoother and more continuous primitives than lines and triangles, thus more convenient for representing object like hair, organic and curved objects
  - Other scene modeling tasks can be performed more simpler and faster, e.g., animation and collision
- <IMAGES: APPLICATION OF CURVES AND CURVED SURFACES>

Parametric curves and surfaces

- To understand parametric surfaces, we will start with parametric curves

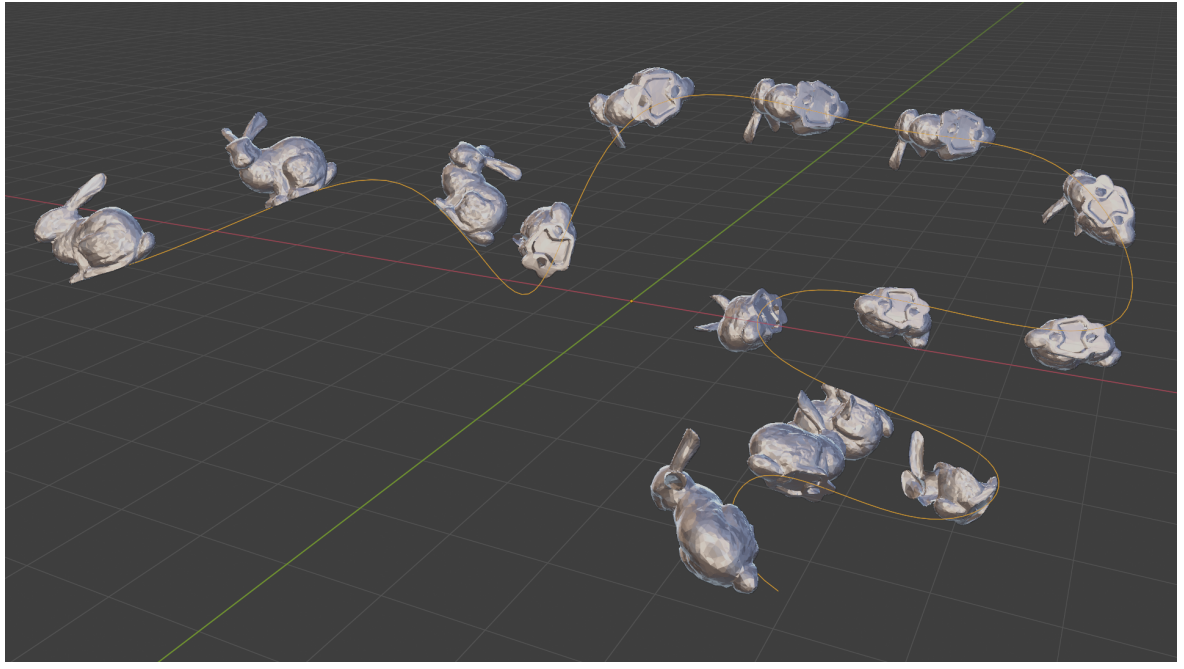




Parametric curves

Parametric curves

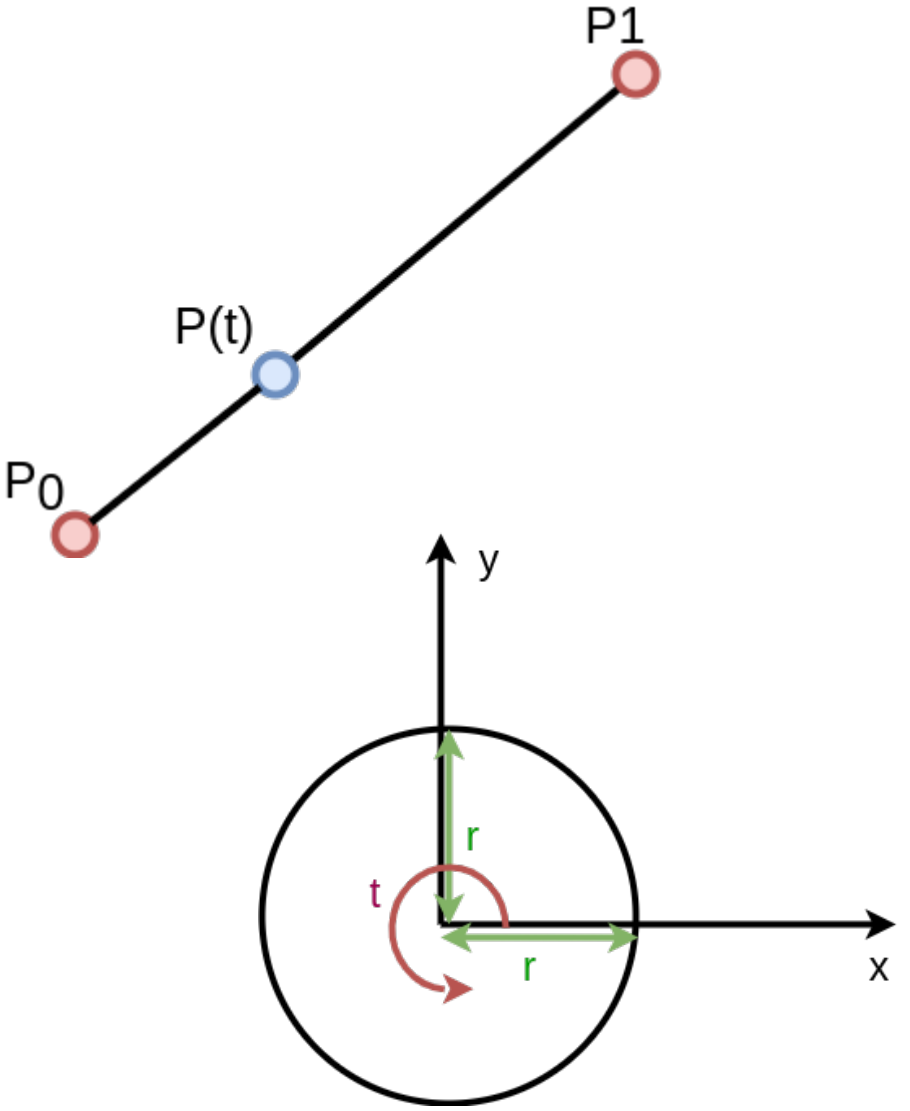
- Wide context of usages:
  - Animating object/camera/light over path: position and orientation
  - Rendering hair



https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-23-hair-animation-and-rendering-nalu-demo

Parametric curves

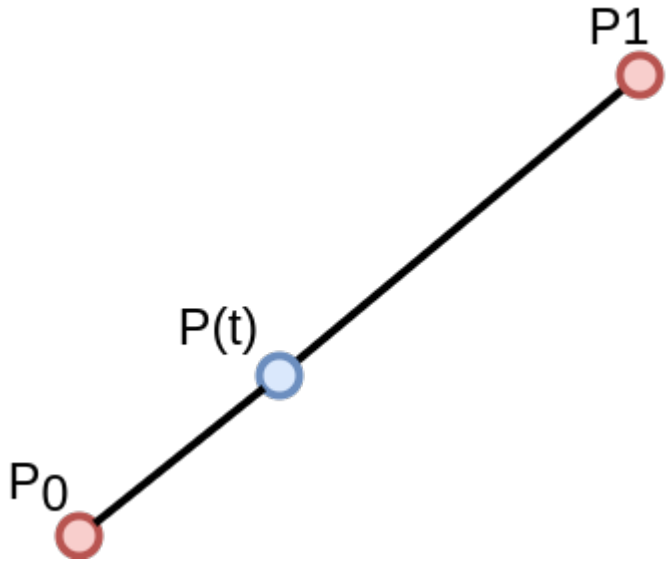
- Described with a formula as a function of parameter t: p(*t*), *t* in [a,b]
  - Generated points are continuous
- Example: **line segment**
  - $p(t) = (1-t)\, p_0 + t\, p_1$
- Example: **circle**
  - x(t) = r cos(2 * PI * t), t in [0,1]
  - y(t) = r sin(2 * PI * t), t in [0,1]



- Arbitrary curves? Various implementations:
  - Bezier curve
  - Hermite curve
  - Catmull-Rom spline
  - B-Splines

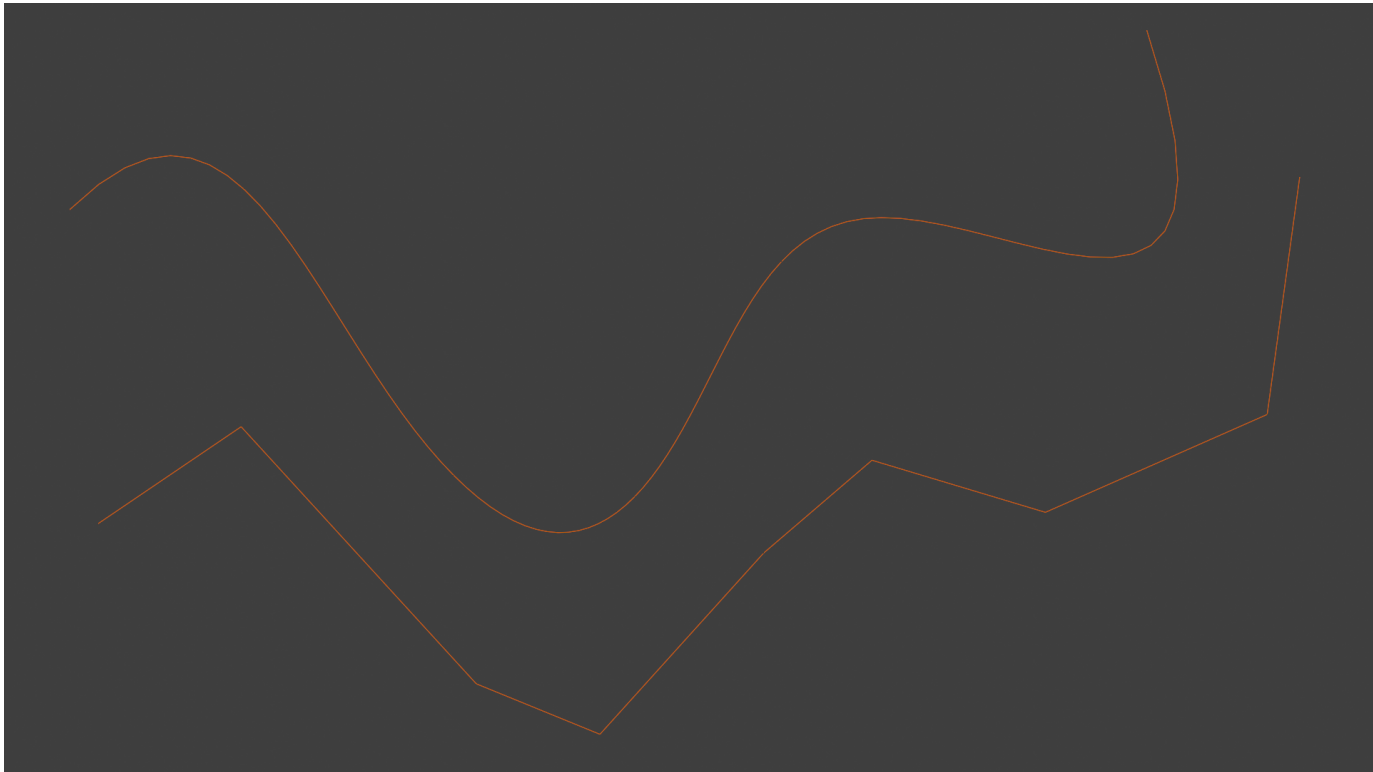
Bezier curves: linear interpolation

- **Linear interpolation** between two points – **control points** - p_0 and p_1 traces out straight line.
  - $p(t) = (1-t)\,p_0 + t\,p_1$, short: `lerp(p0, p1, t)`
  - For $0 < t < 1$, generated points are on straight line between p_0 and p_1. Otherwise outside.
  - $p(0) = p_0$ and $p(1) = p_1$



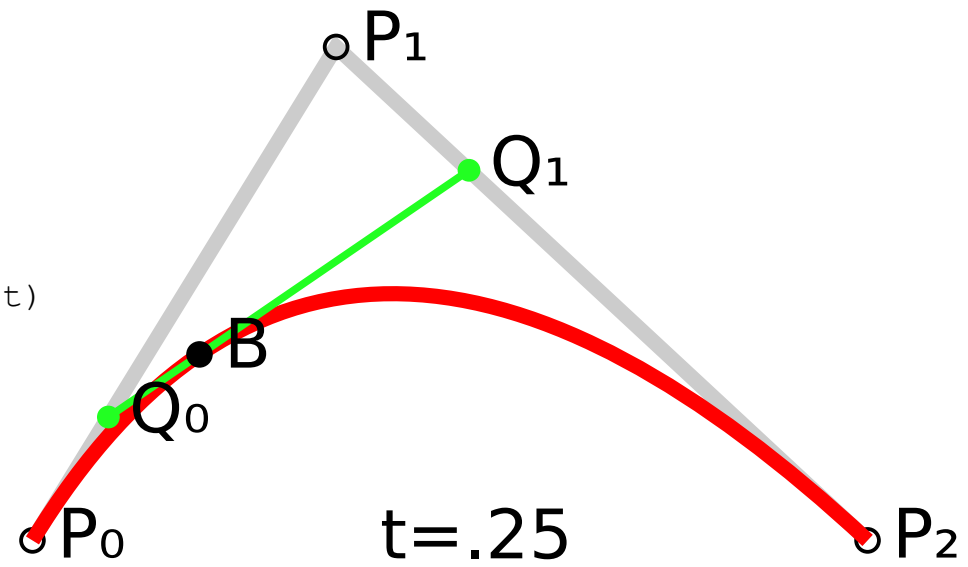
Bezier curves: linear interpolation

- Linear interpolation is fine for two points. But interpolating between multiple points gives us straight segments with sudden (discontinuous) changes at joints between.



Bezier curves: repeated interpolation

- This problem can be solved by taking linear interpolation one step further and **linearly interpolate repeatedly → Bezier curves***
  - To repeat interpolation, **control points** are added: P_2
  - Example: **3 control points**: P_0 ,P_1, P_2
  - Linearly interpolate P_0 and P_1 to obtain Q_0
  - Linearly interpolate P_1 and P_2 to obtain Q_1
  - Linearly interpolate Q_0 and Q_1 to obtain curve point B
  - `B(t) = lerp(lerp(P0,P1,t),lerp(P1,P2,t),t)`

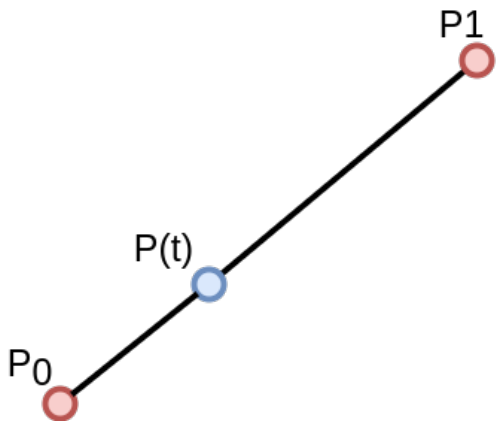


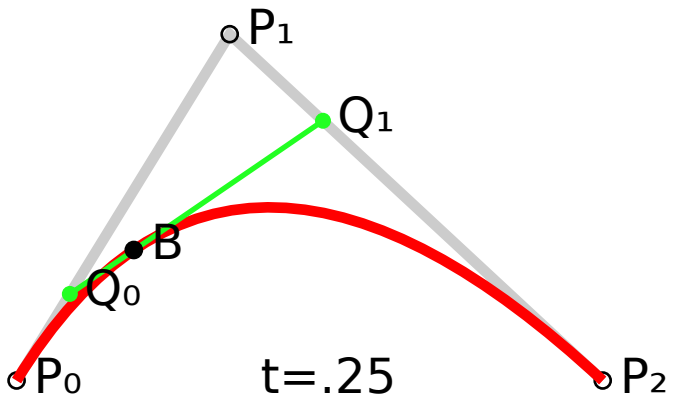
https://en.wikipedia.org/wiki/B%C3%A9zier_curve

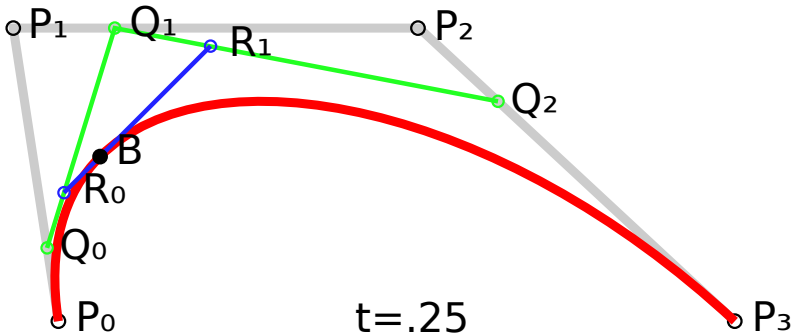
* Independently discovered by Paul de Casteljau and Pierre Bezier for use in French car industry.

Bezier curves: degrees

- **Degree of curve** is n, if n +1 control points are used.
  - More control points → more degrees of freedom
  - n = 1 → 1st degree, **linear curve**
  - n = 2 → 2nd degree, **quadratic curve**
  - n = 3 → 3rd degree, **cubic curve**







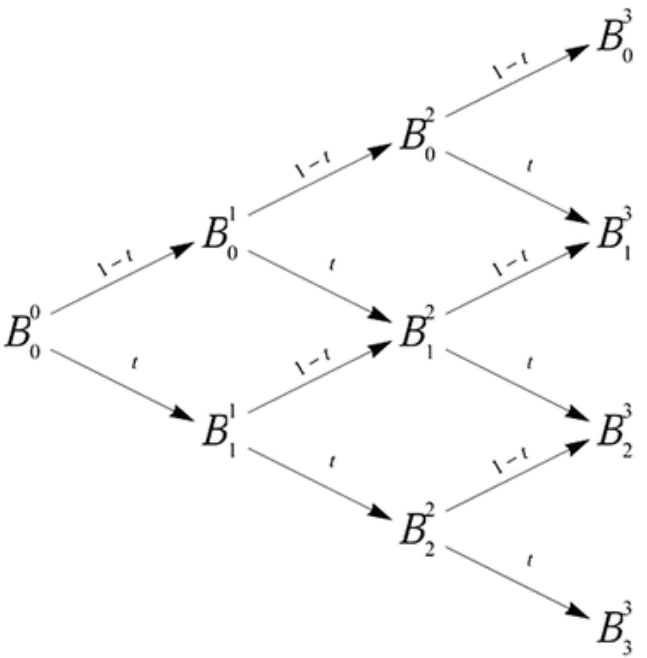
Bezier curves: repeated interpolation

- Repeated or recursive linear interpolation is often referred as **de Casteljau algorithm**.
- Linear Bezier: $\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1, \; 0 \le t \le 1$
- Quadratic Bezier: $\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2], \; 0 \le t \le 1,$
- Cubic Bezier: …
- Higher degree: recursive formula:

$$\mathbf{B}_{\mathbf{P}_0}(t) = \mathbf{P}_0, \text{ and}$$

$$\mathbf{B}(t) = \mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_{n-1}}(t) + t\mathbf{B}_{\mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n}(t)$$

- De Casteljau algorithm gives pyramid of coefficients



Bezier curve: another representation

- Quadratic Bezier: $\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2],\ 0 \le t \le 1,$
- Quadratic Bezier re-arranged → algebraic description

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2,\ 0 \le t \le 1$$

- Every Bezier curve can be described with algebraic fromula. Therefore, **repeated interpolation is not needed.**
- Generalized algebraic description: **Bernstein form**:

$$\begin{aligned}\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i \\ &= (1-t)^n\mathbf{P}_0 + \binom{n}{1}(1-t)^{n-1}t\mathbf{P}_1 + \cdots + \binom{n}{n-1}(1-t)t^{n-1}\mathbf{P}_{n-1} + t^n\mathbf{P}_n, \qquad 0 \leqslant t \leqslant 1\end{aligned}$$

Bezier curve: Bernstein form

- Bernstein form:

$$\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}(t)\mathbf{P}_i, \quad 0 \le t \le 1$$

- **Bernstein polynomials** aka **Bezier basis functions**:

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n$$

- If n = 2 (quadratic)

$$B(t) = \sum_{i=0}^{2} \binom{2}{i} t^i (1-t)^{(2-i)}$$

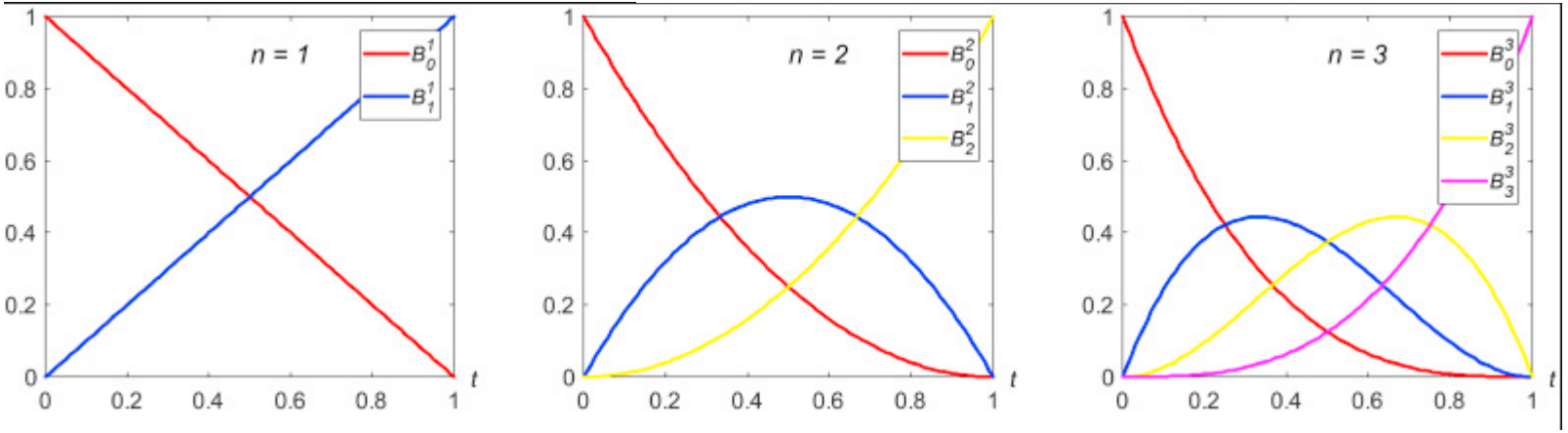
$$B(t) = \binom{2}{0} t^0 (1-t)^2 P_0 + \binom{2}{1} t^1 (1-t)^1 P_1 + \binom{2}{2} t^2 (1-t)^0 P_2, 0 \le t \le 1$$

$$\mathbf{B}(t) = (1-t)^2 \mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2 \mathbf{P}_2, \ 0 \le t \le 1$$

Bernstein polynomials

- When t increases, blending weight form P0 decreases and blending weight for p1 increases, and so on…

$$\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}(t)\mathbf{P}_i, \quad 0 \le t \le 1 \qquad b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n$$



Bernstein polynomials aka blending functions

https://www.sciencedirect.com/science/article/abs/pii/S0997753817306381

Bernstein polynomials

- Bernstein function contains **Bezier basis function (Bernstein polynomials)** that defines properties of the curve:

$$b_{i,n}(t) \in [0,1] \text{ when } t \in [0,1]$$

- Polynomials are in [0,1] when t is given in [0,1]

$$\sum_{i=0}^{n} b_{i,n}(t) = 1$$

- Curve will stay close to the control points P_i.
  - Furthermore, whole Bezier curve will be located in **convex hull** of control points – useful for computing bounding area or volume of curve.

Bezier curve: matrix representation

- In practice, it is useful (faster calculation) to represent Bezier curve in matrix form.
- In practice, **cubic Bezier curves are often used**.

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3,\ 0 \le t \le 1$$

$$B(t) = (1\ t\ t^2\ t^3\ ) \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ 1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

https://blog.demofox.org/2016/03/05/matrix-form-of-bezier-curves/

Bezier curves: good properties

- Intuitive theory, good for understanding curves: repeated interpolation
- Compact form: Bernstein form and matrix representation, power form*
- Derivative of curve is straightforward: derivation of polynomial → tangent vector of curve point
- Useful: Arbitrary number of points can be generated on curve
  - If rotation of those points is needed, then curve (few control points) are rotated and then points are generated.
- More on topic: https://www.youtube.com/watch?v=aVwxzDHniEw&ab_channel=FreyaHolm%C3%A9r

* Real-time rendering book, ch 17.1.

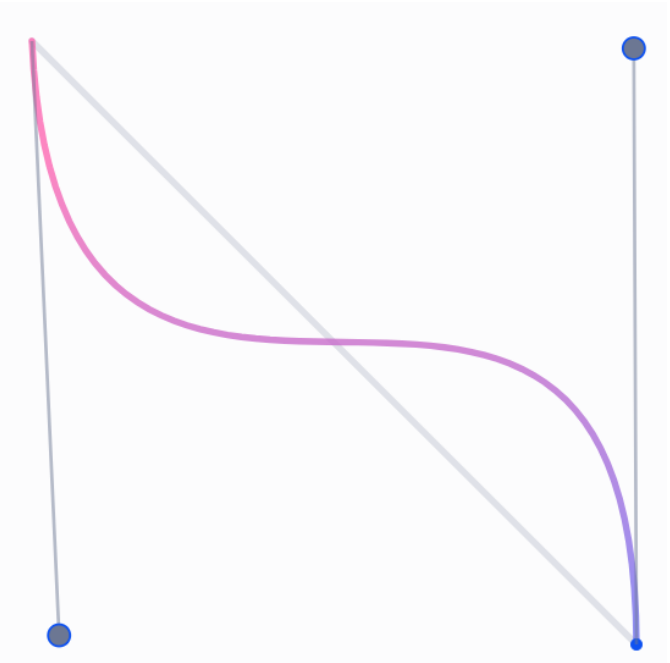
Bezier curves: problems

- Bezier curves do not pass through all control points (except endpoints)
- Not many degrees of freedom:
  - Only control points can be chosen freely
  - Not every curve can be described with Bezier curve (e.g., simple circle can not be described with one or collection of Bezier curves)
  - Alternative is rational Bezier curve*
- Degree increases with number of control points
  - Hard to control if higher degree curve is used, complex computation
  - Bernstein polynomials do not interpolate well
  - For this reason, lower degree curves (often cubic curves) are concatenated to form larger spline

* Hoschek, J. and Lasser, D. (1993) Fundamentals of Computer Aided Geometric Design. A K Peters.

Joining Bezier curves

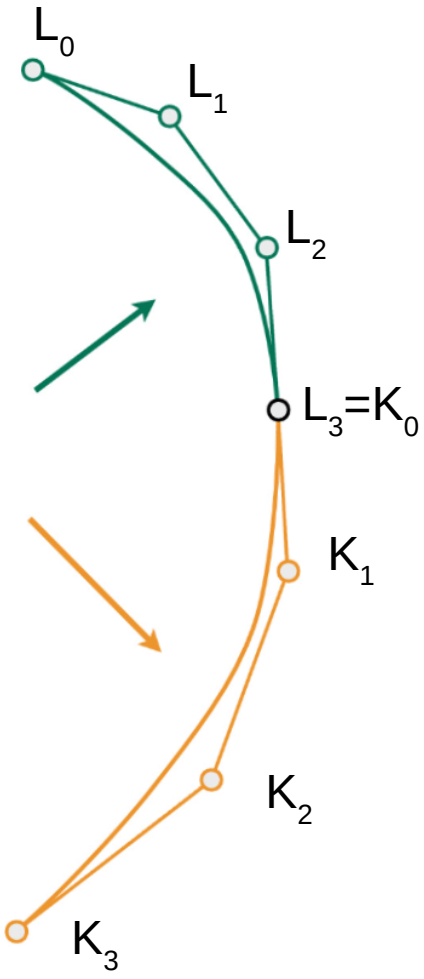
- Often, **multiple Bezier curves** of lower degree are joined together to form **splines**
  - Lower degree curves lower the complexity of computation
  - Resulting curves will go through set of of points
- For this purpose, **cubic degree** is often used
  - Cubic curves are lowest degree curves that can describe S-shaped curve called **inflection**



Cubic Bezier with inflection: https://blog.maximeheckel.com/posts/cubic-bezier-from-math-to-motion/

Joining cubic Bezier curves

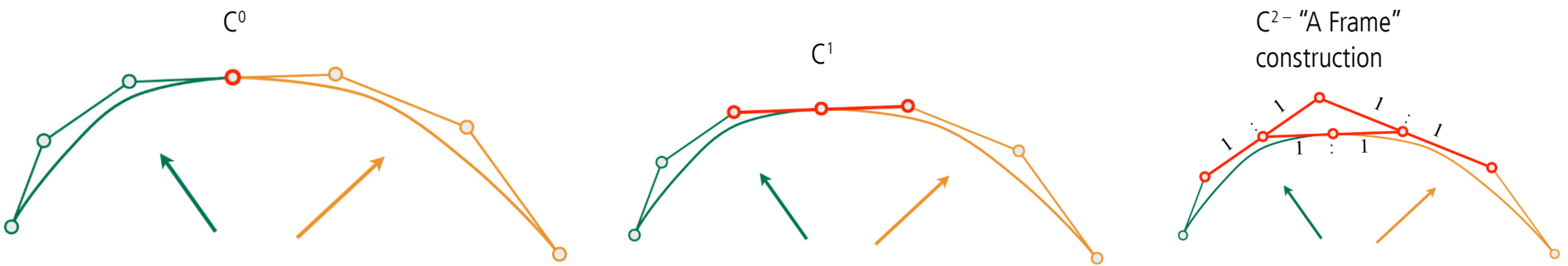
- Example: two cubic Bezier curves with 4 control points:
  - L_i, i = 0, 1, 2, 3
  - K_i, i = 0, 1, 2, 3
- To join the curves we can set $L_3 = K_0$ → **joint** - point where curves are joined
- Composite curve formed from several curve pieces is called **piecewise Bezier curve, p(t)**
  - Now t is in [t1, t2]
  - First curve is p(t'), where t' in [0,1].
  - Second curve is p(t'), where $t' = (t - t1) / (t2 - t1)$
- Two curves connected just using $L_3 = K_0$ will not be smooth at joint.
- Improved smoothness is achieved using tangent constraint: $(K_1 - K_0) = c\,(L_3 - L_2)$, $c > 0$



* Curves and Surfaces in Computer Aided Geometric Design, Farin, Gerald

Joined curves continuity

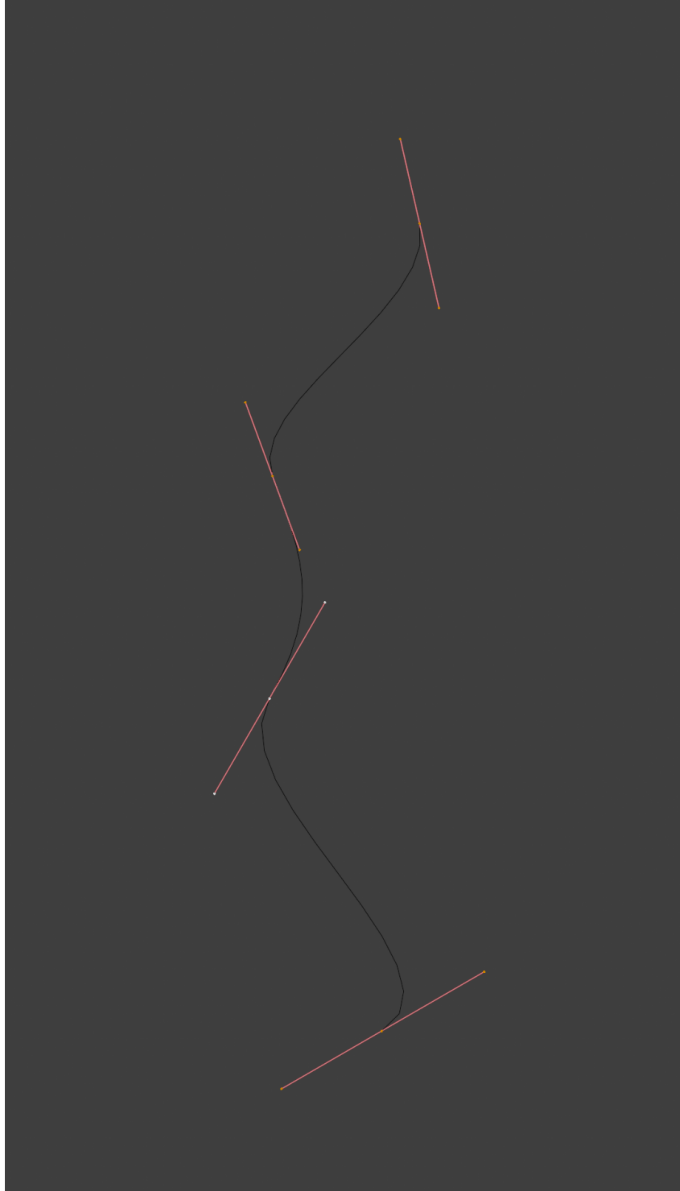
- This way, many cubic Bezier curves are chained into piecewise cubic Bezier line
- Continuity of joined curves:
  - C^0 – **positional continuity** - segments should joint at the same point
  - C^1 – **velocity continuity** - derivation of any point (including joints) must be continuous
  - C^2 - **acceleration continuity** - first and second derivatives are continuous functions



https://cs184.eecs.berkeley.edu/sp22/lecture/7/intro-to-geometry-splines-and-be

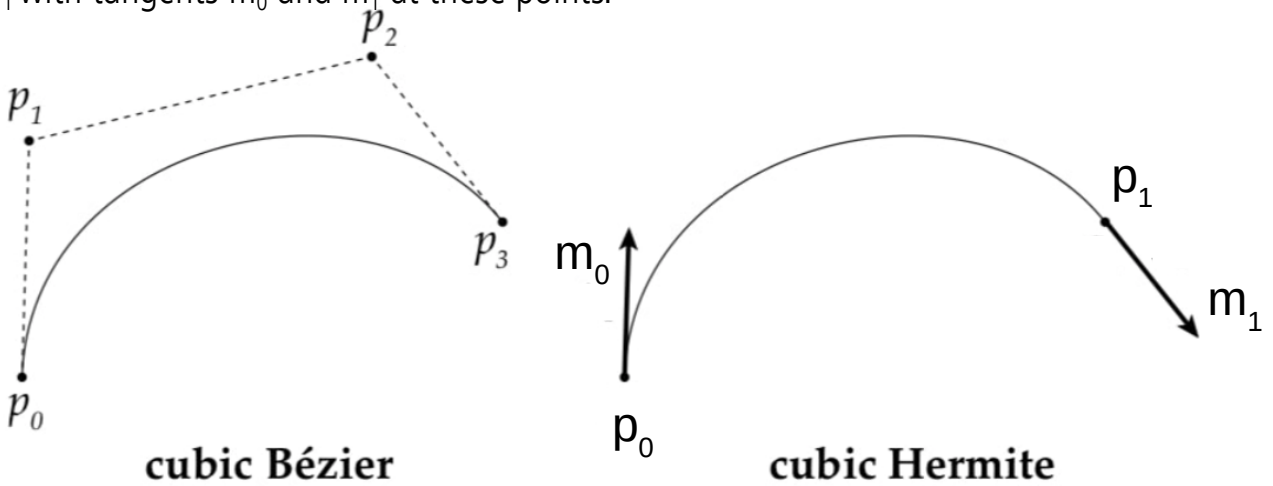
Joined curves continuity

- Geometrical continuity for describing joined curve smoothness
  - G^0 – **positional continuity**: holds when the end points of two curves or surfaces coincide
  - G^1 – **tangent continuity** - tangent vectors from curve segments that meet at joint should be parallel and have same direction – no sharp edges. Continuous edges make splines look natural – often sufficient measure
  - G^2 – **curvature continuity** - tangent vectors from curve segments that meet at joint should be of same length and rate of length change – perfectly smooth surface – two joined surfaces appear as one



Cubic Hermite curve

- Bezier curves are good for describing theory behind smooth curves but are not controllable for authoring.
- Cuves with cubic **Hermite interpolation** are easier to control and require:
  - 2 points. Starting and ending control points: p_0 and p_1
  - 2 vectors. Starting and ending tangents: m_0 and m_1 (longer tangents influence the shape)
- Cubic Hermite curve **interpolates** p_0 and p_1 with tangents m_0 and m_1 at these points.



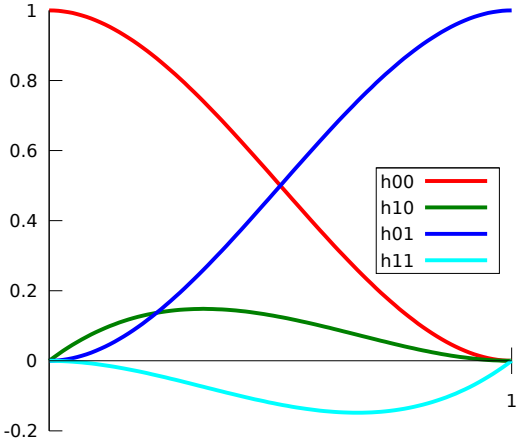
http://15462.courses.cs.cmu.edu/fall2020/article/10

Joining cubic Hermite curve

- Cubic Hermite curve (aka interpolant aka segment aka spline segment) p(t), t in [0,1]:

$$\boldsymbol{p}(t) = (2t^3 - 3t^2 + 1)\boldsymbol{p}_0 + (t^3 - 2t^2 + t)\boldsymbol{m}_0 + (-2t^3 + 3t^2)\boldsymbol{p}_1 + (t^3 - t^2)\boldsymbol{m}_1$$

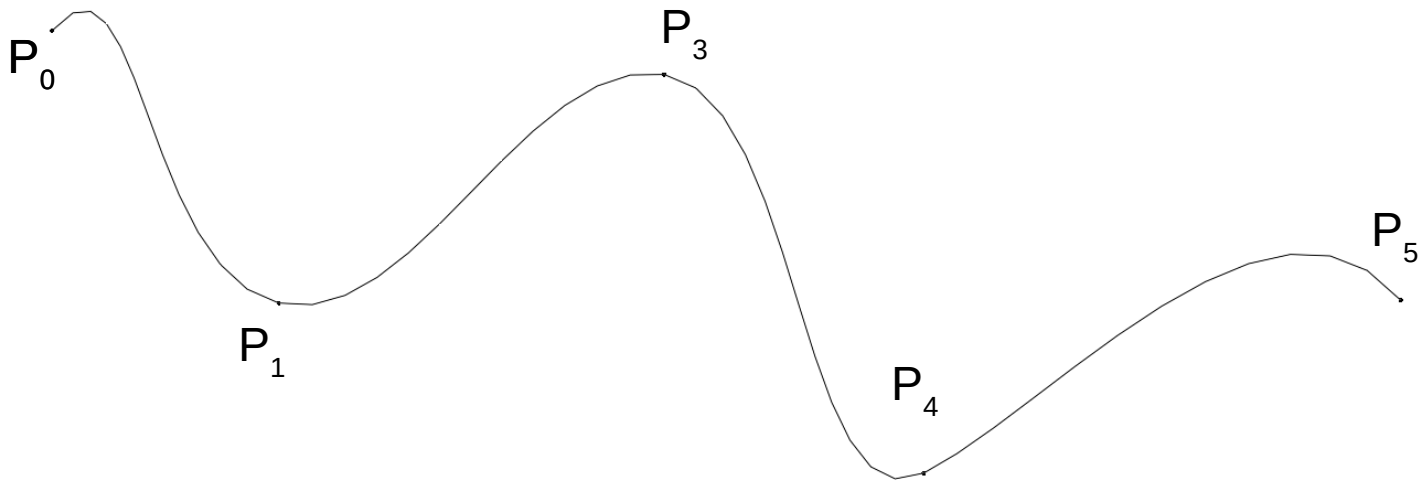
- Similarly as discussed with Bezier, Bernstein form exists with **Hermite basis function**
- When interpolating more than two points, several Hermite curves can be connected together (similarly as done with Bezier).



https://en.wikipedia.org/wiki/Cubic_Hermite_spline

Splines: assembly of curves

- Connecting multiple points $P_0 \ldots P_n$
  - n – 1 cubic Hermite curves must be used
  - resultant is **assembly of curves** called **spline**
  - elements of assembly are called **segments** (e.g., Hermite sements)



- Given points $P_0 \ldots P_n$, the goal is to calculate tangents for Hermite curves which would create spline: continuous differentiable curve passing through each point.
- Solution:
  - Catmull-Rom spline

Catmull-Rom spline

- **Kochanek-Bartels:** method for computing tangents at joints
- Assume that there is only one tangent per control point
  - Tangent at Pi can be computed as combination of two chords: P_i-P_{i-1} and P_{i+1}-P_i:

$$m_i = \frac{(1-a)(1+b)}{2}(p_i - p_{i-1}) + \frac{(1-a)(1-b)}{2}(p_{i+1} - p_i)$$

  - **Tension parameter - a**: length of tangent, higher values → sharper bends
  - **Bias parameter - b**: direction of tangent
- **Kochanek-Bartels method** can be extended with additional, continuity, parameter and which is incorporated with calculating 2nd tangent at joint*.

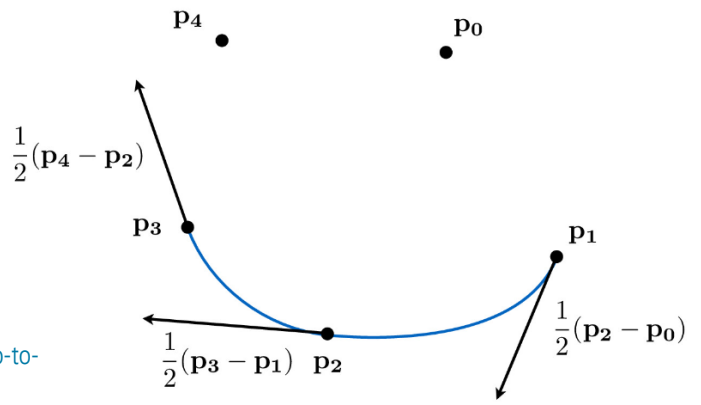
* Real-time rendering book, ch 17.1.

Catmull-Rom spline

- **Catmull-Rom** spline is a special case of **Kochanek-Bartels Spline** where tension and bias parameters are set to 0. Tangents in $P_0 \ldots P_n$ are calculated using:

$$m_i = \frac{1}{2}(p_i - p_{i-1}) + \frac{1}{2}(p_{i+1} - p_i)$$

- Properties:
  - Rapid finding points on Catmull-Rom spline is possible
  - Interpolating curve: passing through all control points but doesn't stay inside convex hull of points



https://cs184.eecs.berkeley.edu/sp22/lecture/7-50/intro-to-geometry-splines-and-be

B-spline

- B-spline is similar to Bezier curve: it is function of t and weighted control points.
  - FORMULA
  - Segment can be expressed in matrix form
- Often cubic B-Spline is used
- B-spline is similar to Catmull-Rom, except:
  - It is C^2
  - It is non-interpolating (it is passing near control points, not through them)
- Two flavours:
  - Uniform
  - Non-Uniform
- Generalizations:
  - Rational
- NURBS – non-uniform rational B splines

Uniform cubic B-spline

- Uniform – spacing between control points is uniform
- Basis function
  - FORMULA
  - Has C2 continuity everywhere: if several B-splines are joined, the composite curve will be C2
    - Curve of degree n has C^{n-1} continuity
  - Basis function is built using integration of previous basis function
- Multiple uniform cubic B-splines curves can be joined together as a spline
  - C2 continuity everywhere
  - No guarantee that it is interpolating the control points
  - TODO

Non-uniform rational B-splines

- Non-Uniform – spacing between control points is not uniform
- Very often used in CAD tools
- EXAMPLE

Parametric surfaces

Parametric surfaces

- After getting familiar with curves, natural extension are **parametric surfaces**
  - Similarly as triangle or polygon is extension of a line segment
- Very useful for modeling curved surfaces
  - EXAMPLE
- As curves, parametric surfaces are defined with small number of control points
- Model made with parametric surfaces is tessellated for efficient rendering process.
  - Surface can be tessellated in any number of triangles making it perfect for tuning trade-off between quality and speed (more triangles → better shading and silhouettes)
  - Another advantage is that animation can be done on control points and then surface is tessellated for rendering.
- Parametric surfaces:
  - Bezier patches
  - Bezier triangle
  - B-spline patch

Bezier Patches

- Bezier curve is extended so it has two parameters (u,v) which define surface
- Similarly as we started with Bezier curve by explaining linear interpolation, we explin Bezier patch by explaining **bilinear interpolation***.
  - IMAGE: bilinear interpolation using 4 points: a,b,c,d
  - e(u) = lerp(a,b,u), f(u) = lerp(c,d,u), p(u,v) = lerp(e(u), p(u), v)
  - p(u,v) is simplest, non-planar parametric surface with (u,v) in [0,1]. It has **rectangular domain** and thus resulting surface is called a **patch**.

* Bilinear interpolation is crucial for computations in computer graphics. It is extensively used and one example is texture mapping.

Bezier Patches

- Similarly as we added more points to linear interpolation to obtain Bezier curve, we add more points to bilinear interpolation to obtain Bezier patch
  - EXAMPLE: biquadratic Bezier patch
  - Nine points arranged in 3x3 grid

Bezier patches

- Repeated bilinear interpolation is extension of de Casteljau's algorithm to patches.
- De Casteljau patches
  - FORMULA
  - Degree of surface: n
  - Control points Pij where i and j belong to [0...n]
- Point on Bezier Patch can be described in **Bernstein form** using Bernstein polynoms
  - EXAMPLE
  - Parameters m and n: bilinear interpolation is performed n times and linear interpolation m-n times
- Properties:
  - Passes through only corner control points
  - Boundary of the patch is described with Bezier curve of degree n formed by the points on the boundary
  - Tangents at border points are described with Bezier curve at border points – each corner control point has two tangents: for u and v direction
  - Patch lies within convex hull of its control points.
  - Control points can be generated and then points on patch will be rotated when evaluated (faster than other way around)
  - Derivative is straightforward.

Rational Bezier patches

- Extension of bezier patches (similarly as for Bezier curves)
- TODO

Bezier patches: examples

- EXAMPLE: how surface look defined with several control points. How moving of the points influences the surface.

Other parametrized surfaces

- **Bezier triangles**
  - Useful when parametric surface is constructed from a triangle using PN triangles of Phong tessellation methods*.
  - Control points are located in triangulated grid
  - Based on repeated interpolation: de Casteljau, Bernstein triangles
  - Constructing complex object requires stitching Bezier triangles so that composite surface contains desired properties and look: continuity

<EXAMPLES>

* Game engines (e.g., unity and unreal) support those methods since triangle mesh is basic building primitive.

Other parametrized surfaces

- **Point-Normal (PN) Triangles**
  - Given triangle mesh with normals at each vertex, the goal is to construct "better looking" surface using just triangles
  - This data is enough to construct surface
  - PN methods tries to improve mesh shading and silhouettes by creating **curve surface to replace each triangle**
- Properties:
  - Creases in PN triangles are hard to control
  - Continuity between Bezier triangles is C0 but looks acceptable for certain applications
- <EXAMPLES>

Other parametrized surfaces

- **Phong tessellation**
  - Similar as PN triangles, given the triangle points with normals, construct surface
  - Phong tessallation attempts to create geometric version of Phong shading normal using repeated interpolation resulting in Bezier triangles.
  - <EXAMPLES>

B-Spline surfaces

- **B-Spline curves** can be extended to B-Spline surfaces which are similar to Bezier Surface
- Often bicubic B-Spline surface is used:
  - to form composite surface
  - Essencial for Catmull-Clark subidivison surfaces
  - <EXAMPLES>
- Non-uniform rational B-Spline surface is often used in 3D modeling software
  - EXAMPLES

Rendering of curved surfaces

- For rendering purposes (both rasterization- and raytracing-based) it is beneficial to transform curved surface into triangulated mesh – tessellation
- EXAMPLES

Modeling with parametric surfaces

- CAD software

Storing and transferring parametric surfaces

- STEP file format
- IGES file format

Exploring parametric curves and surfaces

- Blender NURBS surfaces:
  https://docs.blender.org/manual/en/latest/modeling/surfaces/introduction.html
- Blender NURBS and Bezier curves:
  https://docs.blender.org/manual/en/latest/modeling/curves/index.html
- Tutorial on curves in Blender:
  https://behreajj.medium.com/scripting-curves-in-blender-with-python-c487097efd13
- Houdini: https://www.sidefx.com/docs/houdini/nodes/sop/curve.html
- Library for creating and manipulating NURBS surfaces and curves: http://verbnurbs.com/
- More examples: https://www.realtimerendering.com/#curves

Further into topic

- Parametric curves and surfaces are highly used and researched method in computer graphics. Until now we have covered foundations. There are many other topics. Some of those will be discussed later, some are out of scope for this course:
  - NURBS:
    https://www.gamedeveloper.com/programming/using-nurbs-surfaces-in-real-time-applications

Literature

- https://github.com/lorentzo/IntroductionToComputerGraphics/wiki/Foundations-of-3D-scene-modeling