Computer Graphics 1

3 Geometry

Summer Semester 2021

Ludwig-Maximilians-Universität München

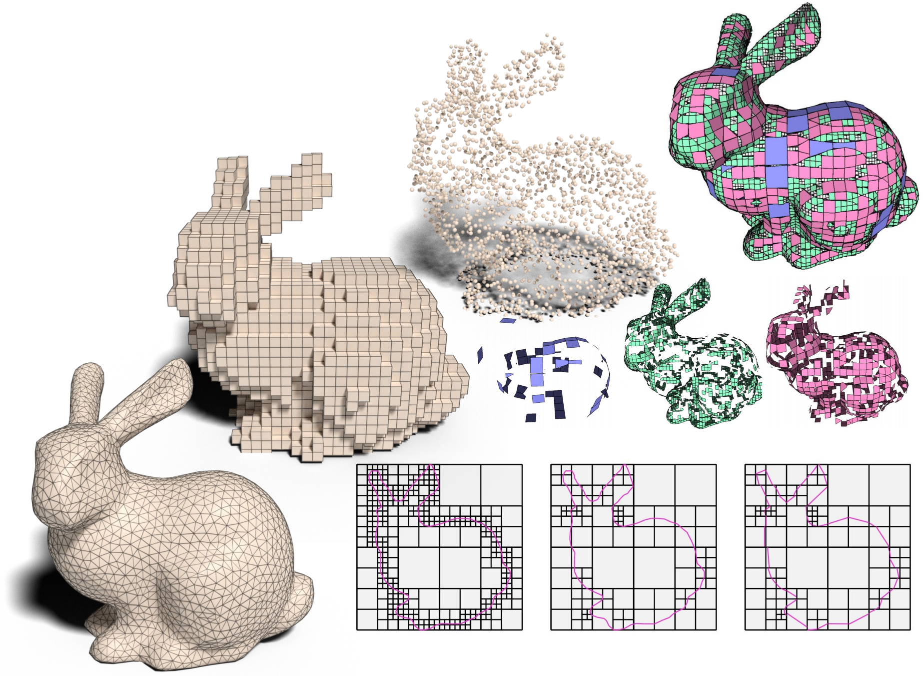
Tutorial 3: Geometry

- Geometric Representations
- Bézier Curves and Interpolation
- Polygon-based Surface Representation
- Mesh Sampling
- Summary

Geometric Representations

Geometry is the ***foundation*** of all graphics, and its representation gives the *language* for describing shape

- **Boundary representation:** deeply embed into modern graphics, and their algorithms are rich and mature
  - **Curve**: Bézier curves, B-splines...
  - **Surface**
    - Bézier surface
    - Polygon mesh: Triangles, quads, etc.



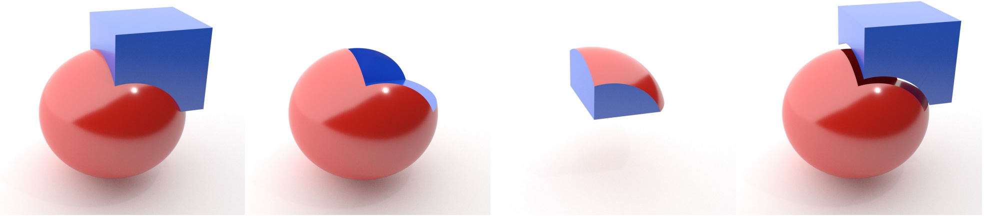
In active research:

- Volumetric representation, e.g. Voxel, tetrahedron, etc.
- Parametric representation
- Procedural/generative models

And there are more geometry representations of course!

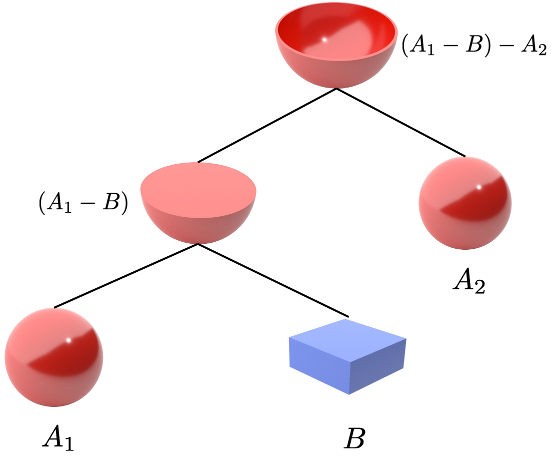
Example: Constructive Solid Geometry (CSG)

CSG is a implicit geometric representation that allows to represent complex models as a series of **boolean operations** between **primitives**.



union (OR) $A \cup B$ | difference (NOT) $A - B$ | intersection (AND) $A \cap B$ | exclusive or (XOR) $A \oplus B$

CSG objects can be represented by binary trees, where leaves represent primitives and nodes represent operations



Example: Advantages and Disadvantages of CSG

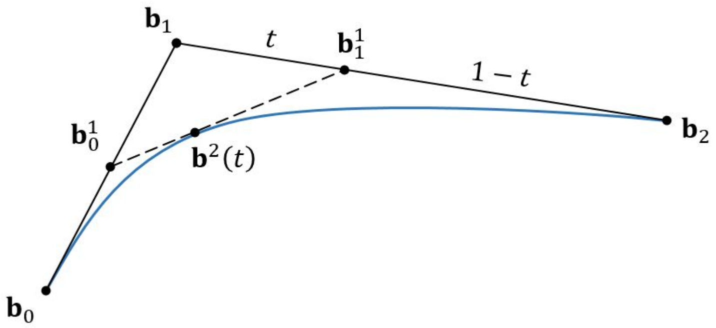
- Advantage
  - Minimum steps: represent solid objects as hierarchy of boolean operations
  - Easy to express a complex implicit surface
  - Low storage space needed: due to the simple tree structure and primitives
  - Easy to convert a CSG model to a polygon mesh (but not vise versa)
  - ...
- Disadvantage
  - Impossible to construct non-solid shape, e.g., organic models
  - High computational power needed to derive boundaries, faces and edges ⇒ needed for interactive manipulation
  - ...

Tutorial 3: Geometry

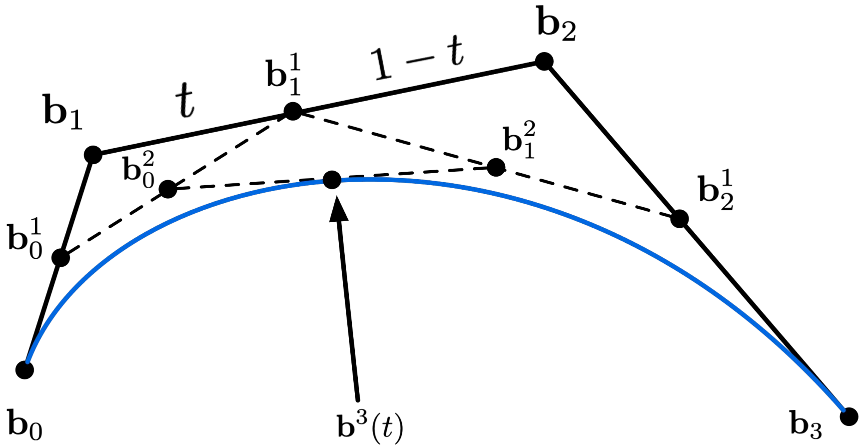
- Geometric Representations
- Bézier Curve and Surface
  - Bézier Curve: de Casteljau Algorithm and Algebraic Form
  - Piecewise Bézier Curves
  - Bézier Patches
- Polygon-based Surface Representation
- Mesh Sampling
- Summary

Bernstein-Bézier Curve

- (Bernstein-)Bézier curve is a *parametric* curve representation and the de facto standard for graphics design
- It has many important properties such as the *de Casteljau algorithm* and elegant geometric *interpolations*
- Applications
  - Describe camera paths to control camera movements
  - Describe animation curves to control object movements
  - ...



Quadratic Bézier
3 control points

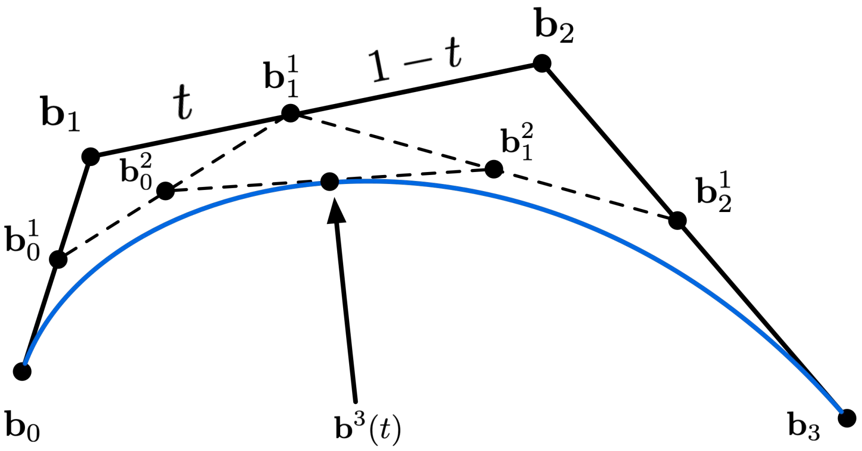


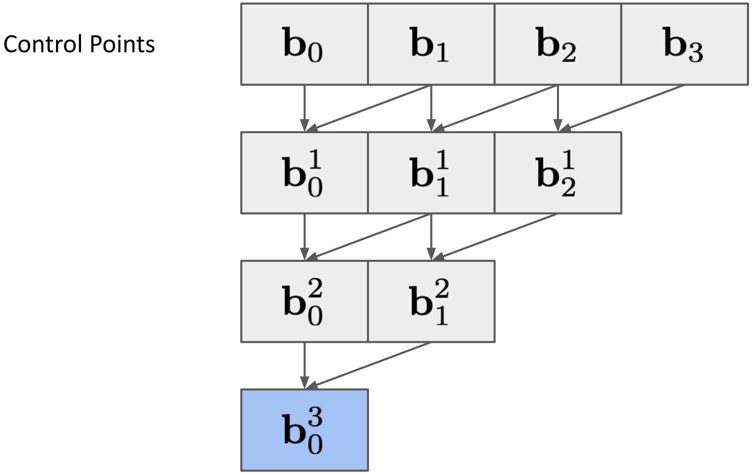
Cubic Bézier
4 control points

de Casteljau Algorithm

The de Casteljau algorithm offers the most intuitive way to describe a Bézier curve, but requires more computation.

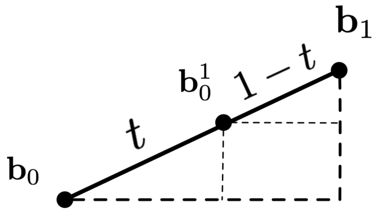
Consider four points (cubic Bézier) as an example:





Implement de Casteljau Algorithm: Interpolation

The coordinates of $\mathbf{b}_0^1$ is linearly interpolated via parameter t, i.e.:



$$\frac{|\mathbf{b}_0\mathbf{b}_0^1|}{|\mathbf{b}_0\mathbf{b}_1|} = t$$

Let $\mathbf{b}_0^1 = (x, y)^\top$, and $\mathbf{b}_0 = (x_0, y_0)^\top, \mathbf{b}_1 = (x_1, y_1)^\top$. Therefore, we have:

$$t = \frac{x - x_0}{x_1 - x_0} = \frac{y - y_0}{y_1 - y_0}$$

$$\begin{aligned} \Longrightarrow \quad & x = x_0 + t(x_1 - x_0) = (1-t)x_0 + tx_1 \\ & y = y_0 + t(y_1 - y_0) = (1-t)y_0 + ty_1 \end{aligned}$$

Breakout: Implement de Casteljau Algorithm

Enter folder demos/03-geometry/1-curve (live demo)

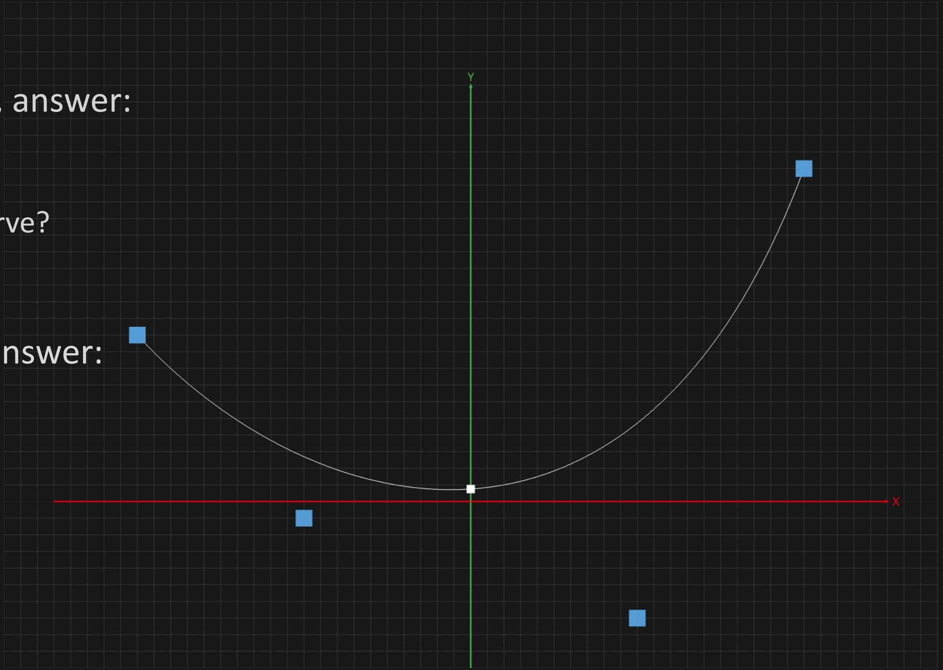
1. Look for TODO comment in the `main.ts`, and implement the `interpolate` function for the de Casteljau algorithm

2. Change the `sample` slider and see how Bézier is sampled, answer:

- What happens when `sample` is below 5?
- How many sample points are good enough to show the Bézier curve?

3. Toggle the `show` checkbox and change the parameter `t`, answer:

- What happens when `t = 0` and `t = 1`?



Bézier Curve: Algebraic Formula

Bézier curve can be further described in an algebraic formula.

For a quadratic Bézier curve:

$$
\begin{aligned}
\mathbf{b}_0^1(t) &= (1-t)\mathbf{b}_0 + t\mathbf{b}_1 \\
\mathbf{b}_1^1(t) &= (1-t)\mathbf{b}_1 + t\mathbf{b}_2 \\
\mathbf{b}_0^2(t) &= (1-t)\mathbf{b}_0^1 + t\mathbf{b}_1^1 \\
&= (1-t)((1-t)\mathbf{b}_0 + t\mathbf{b}_1) + t((1-t)\mathbf{b}_1 + t\mathbf{b}_2) \\
\implies \mathbf{b}_0^2(t) &= (1-t)^2\mathbf{b}_0 + 2t(1-t)\mathbf{b}_1 + t^2\mathbf{b}_2
\end{aligned}
$$

For a cubic Bézier curve:

$$\mathbf{b}_0^3(t) = (1-t)^3\mathbf{b}_0 + 3t(1-t)^2\mathbf{b}_1 + 3t^2(1-t)\mathbf{b}_2 + t^3\mathbf{b}_2$$

...

In general, Bézier curve can be written as

$$\mathbf{b}_0^n(t) = \sum_{i=0}^{n} B_i^n(t)\mathbf{b}_i$$

where

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Binomial Coefficient

is called Bernstein basis.

Bézier Curve: Properties

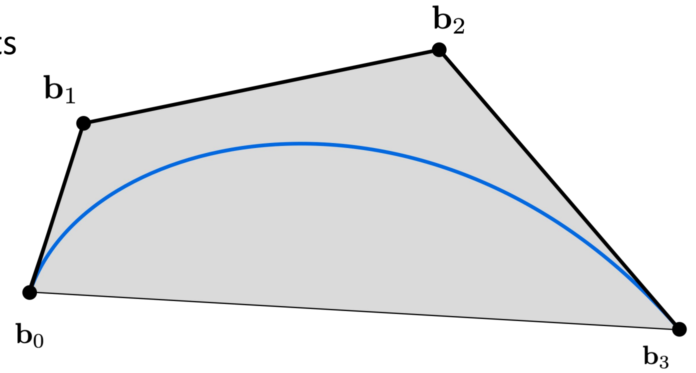
$$\mathbf{b}^n(t) = \sum_{i=0}^{n} B_i^n(t)\mathbf{b}_i$$

1. **An affine transformation of a Bézier curve is the same as transforming its control points** (Q: how to prove this?)

   No need to transform every point on a curve/surface ⇒ good performance!

$$f(\mathbf{b}^n(t)) = f(\sum_{i=0}^{n} B_i^n(t)\mathbf{b}_i) = \sum_{i=0}^{n} B_i^n(t) f(\mathbf{b}_i), f(x,y) = (ax+by+c, dx+ey+f)^\top$$

2. The curve lies within a *convex hull* of its control points



3. Interpolates endpoints (nice boundary condition)

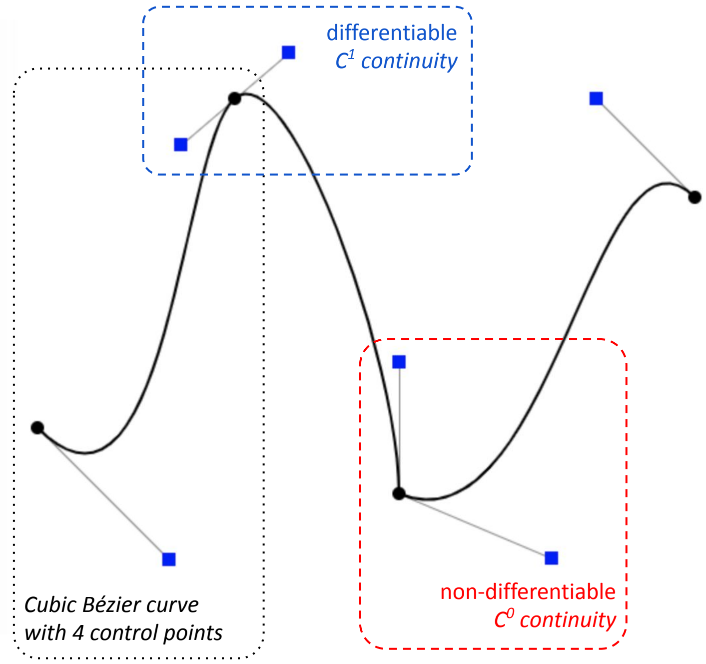
$$\mathbf{b}^n(0) = \sum_{i=0}^{n} B_i^n(0)\mathbf{b}_i = \mathbf{b}_0$$

$$\mathbf{b}^n(1) = \sum_{i=0}^{n} B_i^n(1)\mathbf{b}_i = \mathbf{b}_n$$

Piecewise Bézier Curves

- The Cubic Bézier curve with 4 control points is widely used (in almost every design software)
- The connection of the two head/tail control points forms a tangent of the Bézier curve
- A "seamless" curve is guaranteed if all given points are ***differentiable*** or ***C^1 continuity***

⇒ Left and right tangent slopes are equal for a connecting point



Breakout: Experiment with Bézier Curve

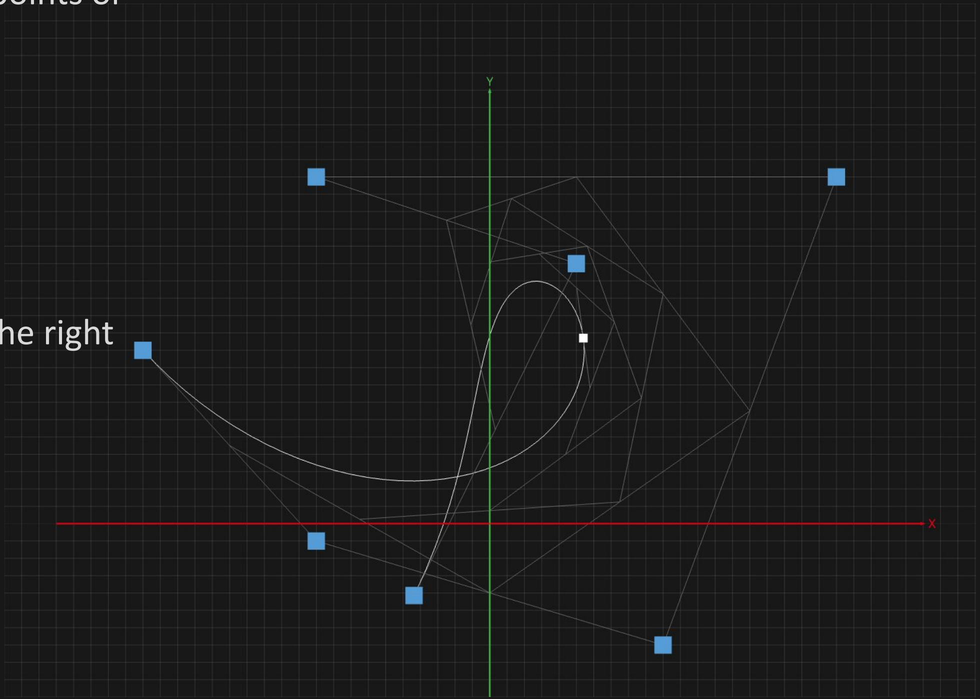
Enter folder demos/03-geometry/1-curve (live demo)

Play around with a Bézier curve created by many control points.

1. Look for TODO comment in the `main.ts` and add new points or remove existing ones from `this.controlPoints`.

2. Drag the control points directly from the visualization.

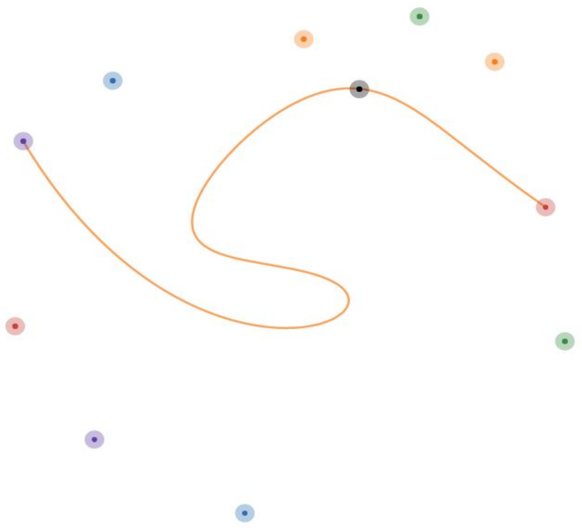
Spend 2 minutes to try to reproduce the Bézier curve on the right side.



Higher-order Bézier Curves

Key issue: Very hard to control!

Can you imagine which control point changes which part of the curve?



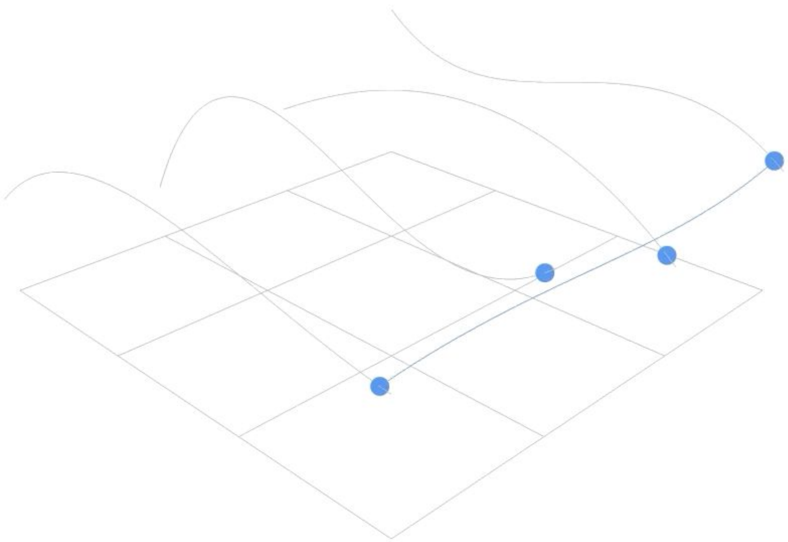
N-order Bézier Curve Playground: https://www.desmos.com/calculator/xlpbe9bgll

Bicubic Bézier Surface (Patch)

4 cubic Bézier curves determine a bicubic Bézier surface:

Each cubic Bézier curve needs 4 control points, with 4 curves, 4x4 = 16 control points in total.

Then, on an orthogonal direction, each Bézier curve contributes one control point.



http://acko.net/blog/making-mathbox/

Tutorial 3: Geometry

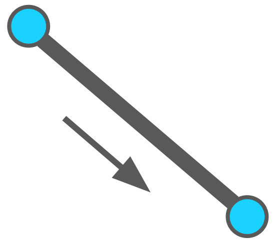
- Geometric Representations
- Bézier Curve and Surface
- Polygon-based Surface Representation
  - Meshes and Wavefront OBJ format
  - Geometry Buffers
- Mesh Sampling
- Summary

Linear Geometric Primitives

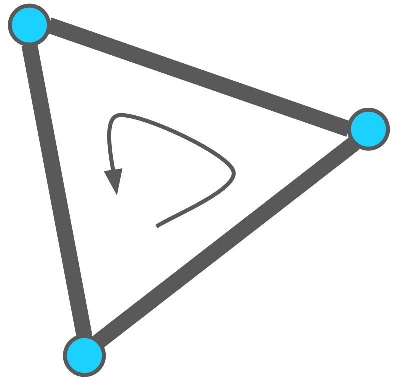
Vertex, edge, and face are the basic geometric primitives for constructing a polygonal-based surface

- A vertex is a point abstraction, and it does not only represent position, but can also contain other information
- An edge represents an *oriented* connectivity of two vertices
- A face is an *oriented* closed edge loop that can be either a triangle, quadrilateral, or arbitrary polygon

Vertex



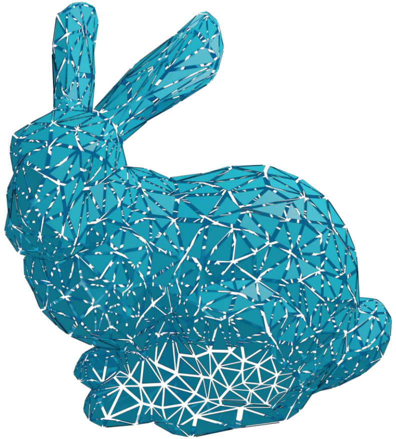
Edge



Face

Polygon-based Surface

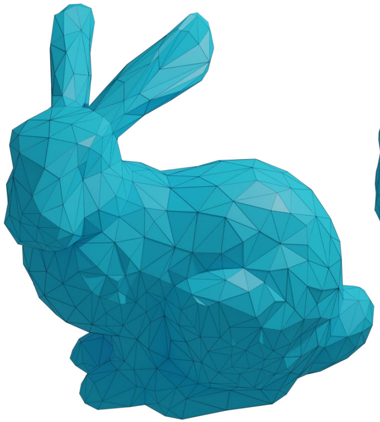
To represent a smooth surface in discrete settings, one can use a collection of polygons, which often refer to *polygon soup*. In a polygon soup, an edge only connects to a single face.



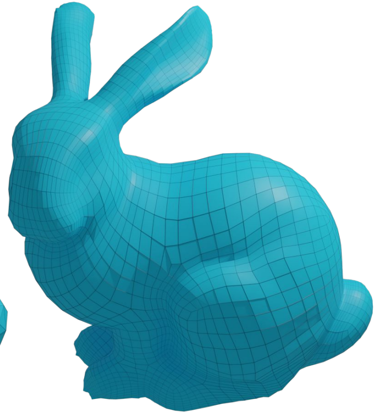
polygon soup

Polygon mesh adds more constraints on a polygon soup where an edge connects multiple faces, such as:

- Triangle mesh
- Quadrilateral mesh (or just quad mesh)
- Quad-dominant Mesh (often refer to a mixture of triangle and quad mesh but mostly quads)
- ...



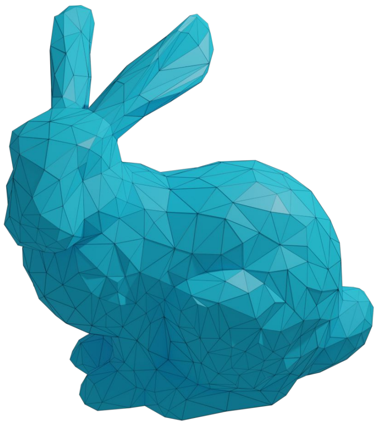
triangle mesh



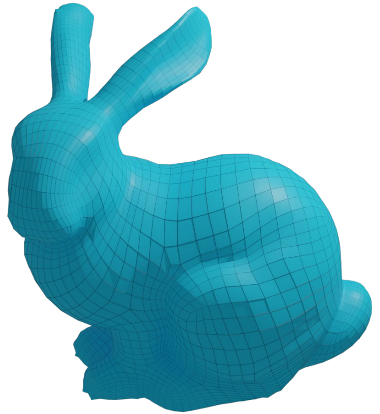
quad mesh

Triangle vs. Quad Mesh

- Triangle Mesh
    - a triangle is the simplest polygon, and other polygons can be turned into triangles
    - a triangle is guaranteed to be planar (linear element)
    - a triangle has well-defined interior (Q: How to check if a point is inside a triangle?)
    - it is easy to compute interactions between a triangle mesh and rays (***later in ray tracing***)
- Quad Mesh
    - quad meshes are much easier for modeling smooth and deformable surface
    - converting a quad mesh to a triangle mesh is a simple process (Q: Why?)
    - quad meshes have many sub-regions with grid-like connectivity (flow line or edge loop)
    - quad meshes are better for subdivisions than tri-meshes



triangle mesh



quad mesh

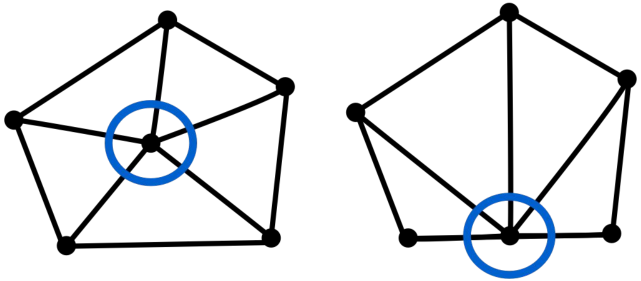
Manifold vs. Non-Manifold

Geometry requires a vertex at a given position in a given metric space, which is distance relevant

Topology concerns vertex connectivities only, which is distance irrelevant

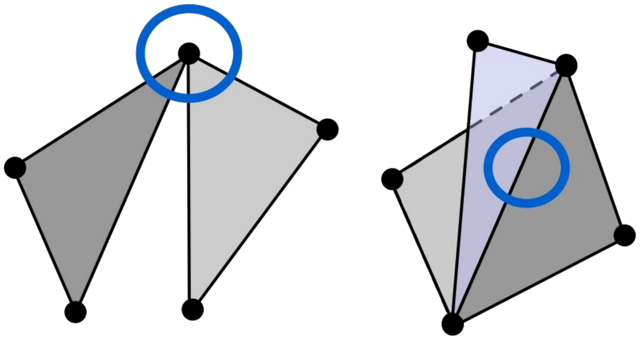
The connectivity information can appear in either vertex or edge, and constitutes an important concept:

Manifold polygon mesh: ***Each edge is incident (touches) to one or two faces***, and ***each faces is incident to a vertex from a closed or open fan***.



closed fan　　open fan

manifold polygon mesh



on vertex　　on edge

non-manifolds polygon mesh

Aside: Non-manifold geometry cause ill-posed situation where geometry processing algorithms often fail to deal, yet still challenging and in active research. Either in geometry modeling or computation a goal is to avoid non-manifold geometry (tedious and non-trivial).

Normals

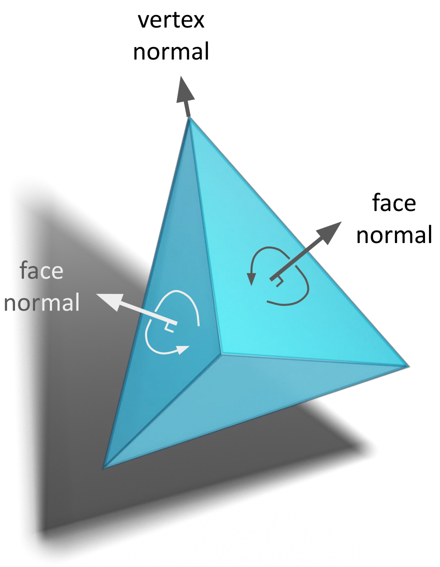
Normals are an important property on a continuous surface.

In discrete settings, there are two types of normals for a triangle mesh:

- Face normal: has *unit* length and is *orthogonal* with the given triangle
  - Face normal of a triangle is well defined (Q: why and how to compute?)
- Vertex normal: is an interpolation from the surrounding face normals
  - There are multiple (different) definitions for vertex normals
  - A possible definition is the average of the surrounding face normals
  - Vertex normals can also be manually defined, i.e. ground truth vertex normals

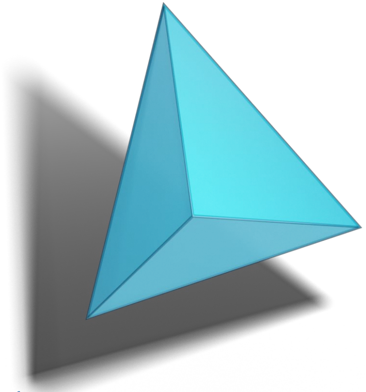
The orientation of a face describes a normal either inward-pointing or outward-pointing. Depending on left- or right handed system, we assume:

- Outward-pointing normals are determined by right-hand rule
- Inward-pointing normals point to the opposite direction of outward-pointing normals



The Wavefront Object File Format (.obj)

- The Wavefront object file format is one of the earliest developed polygon-based surface geometry definitions
- 3D software (eg. Blender) allows user to manually change geometry and exports the final result to a .obj file with predefined specification
- The format stores geometric information such as vertex *positions*, vertex *normals*, vertex *UV coordinates* (2D vector, later discuss in texture session) in an array together with a face *adjacency list*, that contains *oriented* vertex indices
- See an example of *tetrahedron* on the right side:



Vertices

```
1    v -0.363322 -0.387725 0.859330
2    v -0.550290 -0.387725 -0.682297
3    v -0.038214 0.990508 -0.126177
4    v 0.951827 -0.215059 -0.050857
```

UVs (later)

```
1    vt 0.436598 0.753560
2    vt 0.833648 0.512884
3    vt 0.833648 1.000000
4    vt 0.436598 0.464299
5    vt 0.000000 0.195168
6    vt 0.436598 0.000000
7    vt 0.000000 0.685842
8    vt 0.423825 0.464299
9    vt 0.423825 0.925956
10   vt 0.436598 0.251320
11   vt 0.823853 0.000000
12   vt 0.823853 0.512884
```

Normals

```
1    vn 0.3538 0.2340 -0.9056
2    vn 0.4727 0.4361 0.7658
3    vn 0.1202 -0.9926 -0.0146
4    vn -0.9454 0.3050 0.1147
```

Faces

```
1    f 3/1/1 4/2/1 2/3/1
2    f 3/4/2 1/5/2 4/6/2
3    f 4/7/3 1/8/3 2/9/3
4    f 2/10/4 1/11/4 3/12/4
```

OBJ File Format: Vertex Data

All vertices are ordered where the vertex index (starts from 1) is implied from their order in the file.

Lines starting with **v** represent vertex positions*:

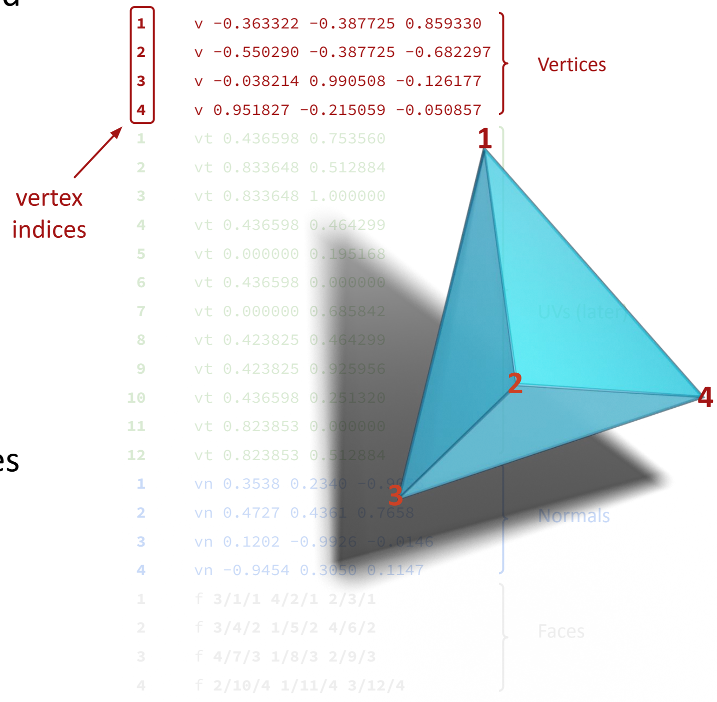
`v x y z`

where x, y, z are the position coordinates

Lines starting with **vt** represent vertex UV coordinates and **vn** describes vertex normal coordinates:

`vt x y`

`vn x y z`



*The actual OBJ file format contains more details such as a vertex position can use homogeneous representation, see here for a full format specification. For simplicity, we assume not using homogeneous representation in .obj file format.

OBJ File Format: Face Data

Lines starting with **`f`** represent a single face described by a list of vertices.

Each vertex concatenates its information using slash (/)

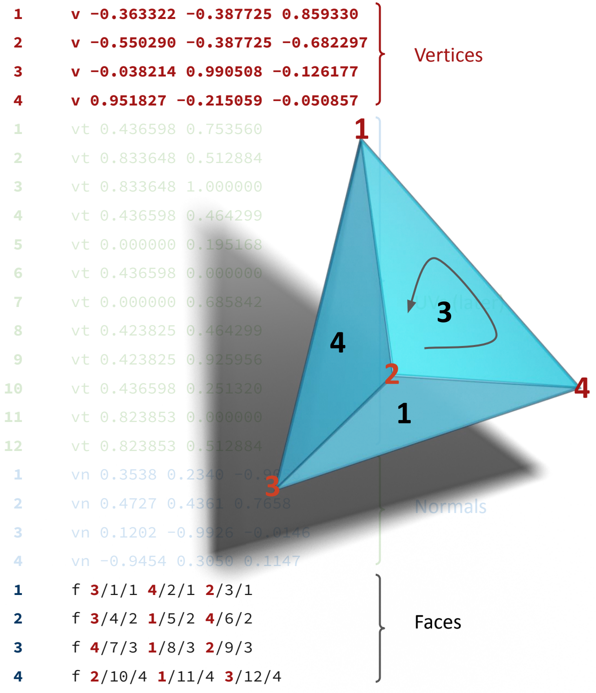
- Triangle face:

```
f v/vt/vn v/vt/vn v/vt/vn
```

- Quad face:

```
f v/vt/vn v/vt/vn v/vt/vn v/vt/vn
```

With more group of vertices, a face can be ngon (polygon with n edges.)



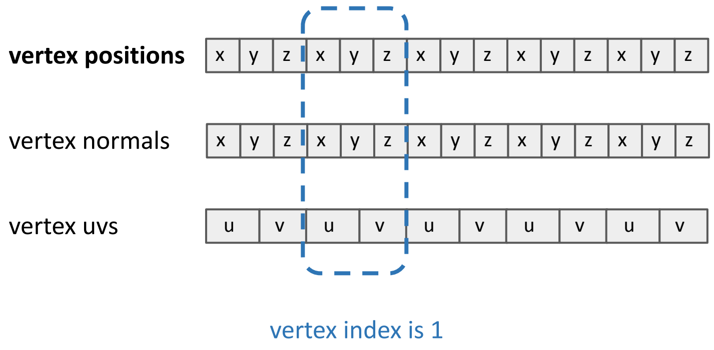
Vertex Buffer

Modern GPUs store a triangle mesh using a dense memory buffer, i.e. vertex buffer.

To create a geometry, one must interpret a vertex buffer using indices.

In three.js, BufferGeometry is the way of representing all (polygon-based) geometry. All geometry data is stored using BufferAttributes, and each BufferAttribute represents an array of one type of data: positions, normals, UVs, etc

By default, the vertex index starts from 0 (Note that this is different from an .obj file, where the index starts from 1)



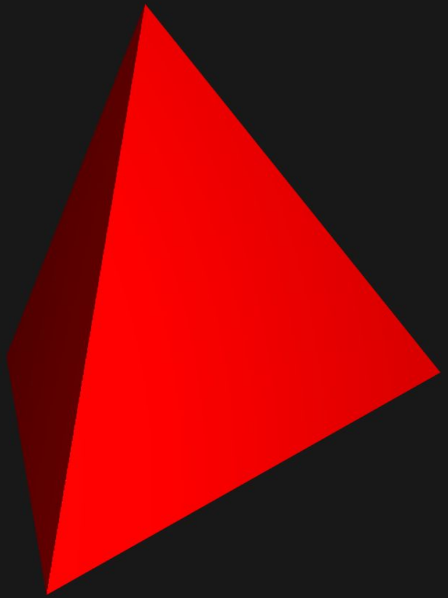
Breakout: Visualize Tetrahedron using BufferGeometry

Enter folder demos/03-geometry/2-buffers (live demo).

Complete the vertex indices that constructs the faces of a tetrahedron below the TODO comment.

```
// vertex buffer object
const vbo = new Float32Array([
  -0.363322, -0.387725, 0.85933,
  ...
]);
// create a buffer geometry
const g = new BufferGeometry();
g.setIndex([
  // TODO: fill the vertex indices

]);
g.setAttribute('position', new BufferAttribute(vbo, 3));
```



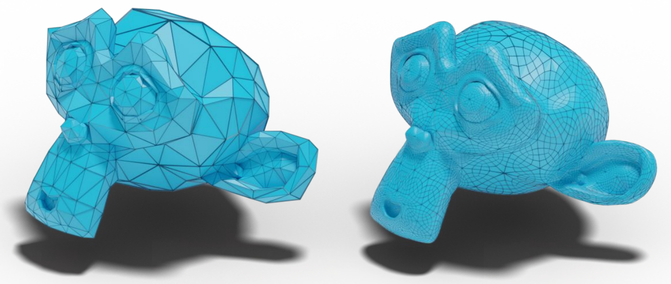
Tutorial 3: Geometry

- Geometric Representations
- Bézier Curve and Surface
- Polygon-based Surface Representation
- **Mesh Sampling**
  - Mesh Simplification
  - Mesh Subdivision
- Summary

Mesh Operators

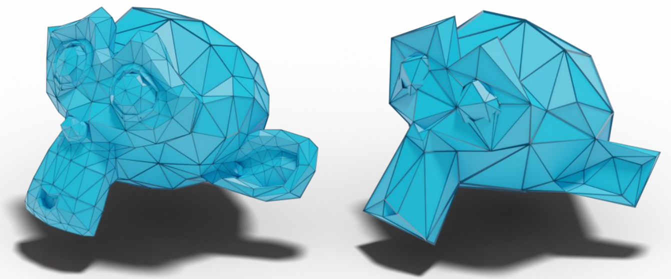
Mesh ***subdivision*** (upsample): Increase the number of polygons to smoothly *approximate its shape*

- Loop subdivision for triangle meshes [Loop, 1987]
- Catmull-Clark subdivision for quad or triangle meshes (in Blender)
- ...



Mesh ***simplification*** (downsample): Reducing number of polygons while *preserving the overall shape*

- **Vertex Clustering** [Rossignac and Borrel, 1993]
- **Melax's Curvature-based Simplification** [Melax, 1998] (in three.js)
- Quadric-error Metric Simplification (in Blender)
- ...



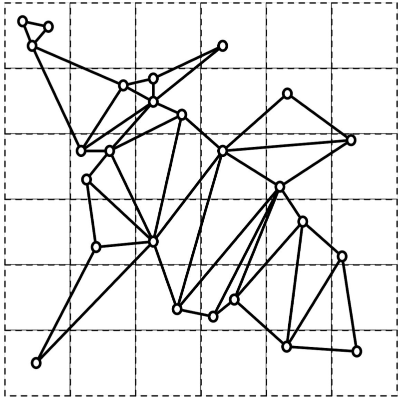
Rossignac, J. and Borrel, P., 1993. ***Multi-resolution 3D approximations for rendering complex scenes***. In Modeling in computer graphics (pp. 455-465). Springer, Berlin, Heidelberg.
Melax, S., 1998. ***A simple, fast, and effective polygon reduction algorithm***. Game Developer, 11, pp.44-49.
Loop, C.T., 1987. ***Smooth subdivision surfaces based on triangles***, Master's thesis Department of Mathematics. University of Utah.

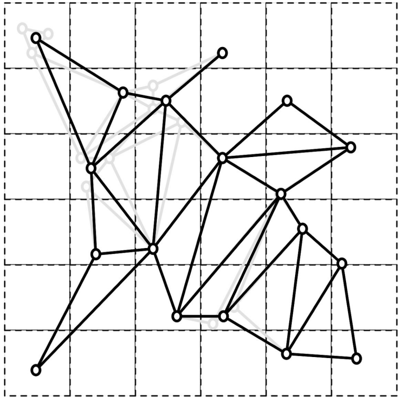
Vertex Clustering

Vertex clustering [Rossignac and Borrel, 1993] is one of the simplest mesh simplification algorithms. Procedure:

1. Divide the 2D/3D space into grids
2. For each cell
   a. replace all nodes by their barycenter (center of mass)
   b. reconnect all edges to the new point (barycenter)

Example: 22 triangles are simplified to 15 triangles (31.8%).

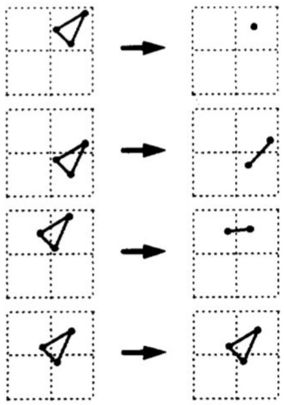




Rossignac, J. and Borrel, P., 1993. ***Multi-resolution 3D approximations for rendering complex scenes***. In Modeling in computer graphics (pp. 455-465). Springer, Berlin, Heidelberg.

Vertex Clustering: Inconsistency

Depending on the position of vertices, the same geometry can lead to inconsistent results:

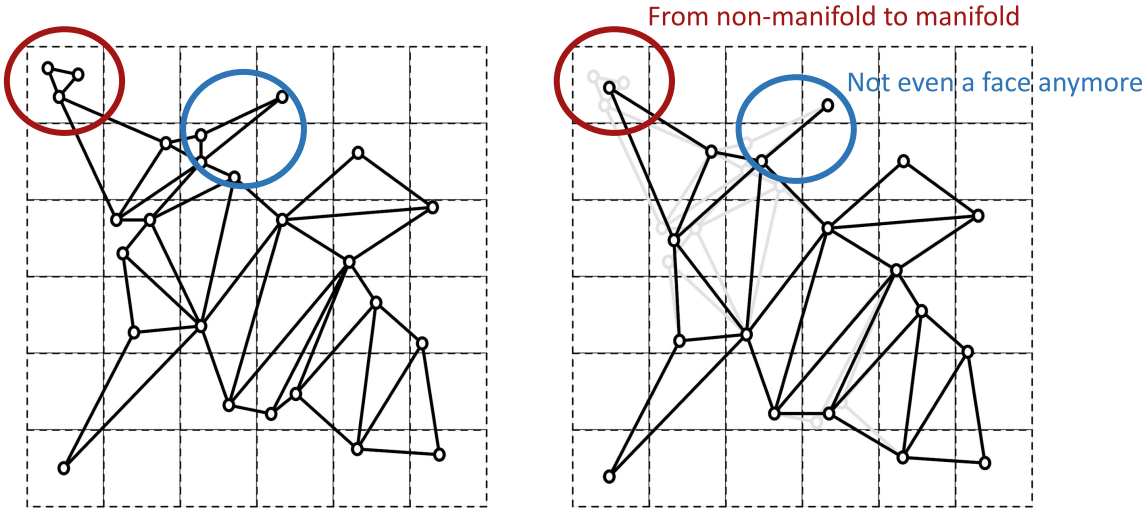


These inconsistency leads to the major drawback of vertex clustering: **topology has changed**

Kok-Lim Low and Tiow-Seng Tan. 1997. ***Model simplification using vertex-clustering***. In Proceedings of the 1997 symposium on Interactive 3D graphics (I3D '97). ACM, New York, NY, USA, 75–ff.

Vertex Clustering: Topology Change

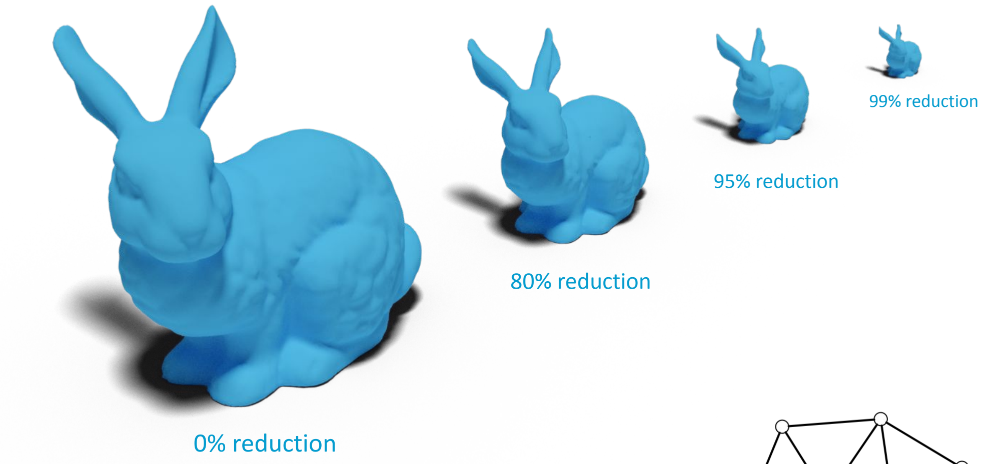
- Non-manifold → Manifold
- Manifold → Non-manifold (Q: Try to name an example)



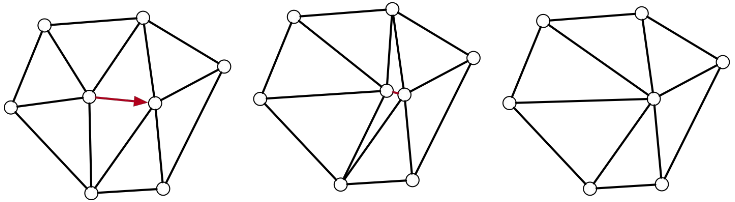
Non-manifold often causes problems both in editing and rendering.

Progressive Meshes and Levels of Detail (LOD)

- Instead of vertex clustering, *edge collapse* is more widely used to simplify a polygon mesh progressively
- Progressive mesh simplification generates different *levels of detail (LOD)*



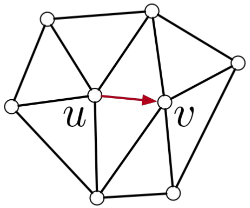
Basic idea: Collapse an edge then merge one vertex into the other



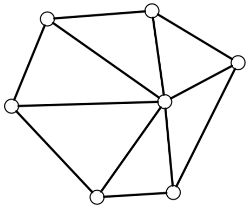
Q: How many vertices, faces and edges are removed in each *edge collapse*?

This process can proceed progressively by selecting the *best (?)* edge iteratively.

Select and Update Edge Cost



before



after

How much does it cost to collapse an edge?

A possible way: cost = edge length $\times$ curvature

$$\mathtt{cost}(u, v) = \underbrace{||u - v||}_{\text{distance}} \times \max_{f \in T_u} \left\{ \min_{n \in T_{uv}} \left\{ \underbrace{1 - f\,\mathtt{.normal} \cdot n\,\mathtt{.normal}}_{\text{curvature}} \right\} \right\}$$

where T_u is the set of triangles that contains u, T_{uv} is the set of triangles that contains both u and v.

We know the cost of collapsing an edge. But if we collapse an edge, the costs of neighbors can also be affected (why?)

How do we **efficiently** simplify a mesh progressively?

Data structure to use: *priority queue* or *min-heap*.

- cost of accessing the minimal element: ***O(1)***
- cost of manipulating the affected elements: ***O(log(n))***

Melax, S., 1998. ***A simple, fast, and effective polygon reduction algorithm***. Game Developer, 11, pp.44-49.

Breakout: Repeat *Upsampling* and *Downsampling*

An interesting procedure of processing a given mesh is to repeat upsampling and downsampling.

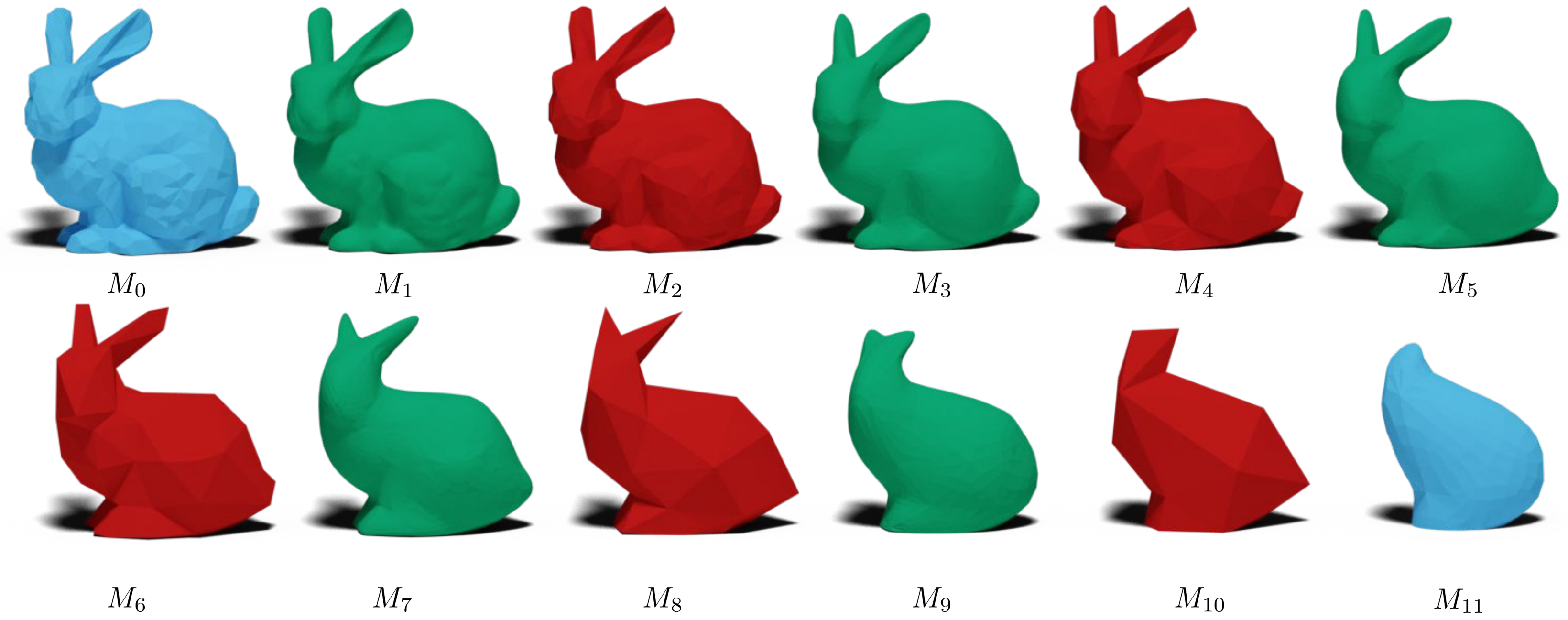
In the provided live demo, let's set subdivision number = 2 (or any others), reduction ratio = 95% (or any others).

Answer this question: what is the result of M_{10}?

$$M_0 \xrightarrow[\text{subdivision}]{} M_1 \xrightarrow[\text{simplification}]{} M_2 \xrightarrow[\text{subdivision}]{} M_3 \xrightarrow[\text{simplification}]{} M_4 \xrightarrow[\text{subdivision}]{} M_5 \longrightarrow \cdots$$

Repeated Up- and Downsampling

Subdivision number = 2, reduction ratio of number of vertices = **98%**:



Q: Is it possible to preserve the #faces and mesh quality when repeating simplification and subdivision?

Mesh Aliasing

Repeated up- and downsampling can be done in different ways:

1. subdivision $\rightarrow$ simplification $\rightarrow$ subdivision $\rightarrow$ simplification $\rightarrow \ldots$

    #vertices/#faces is reduced over iteration

    #vertices/#faces is increased over iteration

2. simplification $\rightarrow$ subdivision $\rightarrow$ simplification $\rightarrow$ subdivision $\rightarrow \ldots$

    #vertices/#faces is reduced over iteration

    #vertices/#faces is increased over iteration

Observation: The shape is still not exactly preserved (further verify by yourself)

When the upsampling is not an inverse operator of downsampling, and vise versa, it causes an *aliasing* issue.

⇒ Aliasing errors occur if the sampling pattern is not perfectly aligned with the features in the original mesh

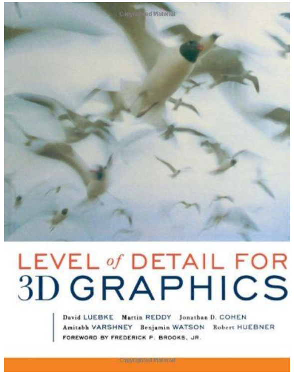
We will see more aliasing issues :)

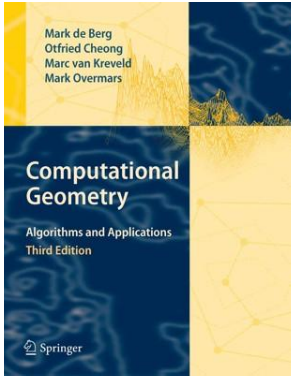
Tutorial 3: Geometry

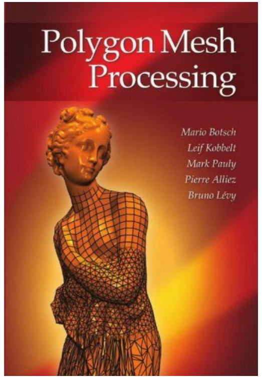
- Geometric Representations
- Bézier Curves and Interpolation
- Polygon-based Surface Representation
- Mesh Sampling
- Summary

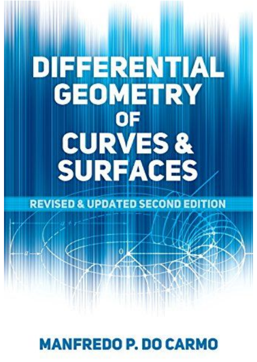
Summary

- We covered:
  - Different geometric representations and CSG as an example of implicit geometry representation
  - How to use the de Casteljau algorithm compute Bézier curves, and its algebraic definition
  - Polygon-based mesh surface, the wavefront object file format, and vertex buffer
  - Two common mesh sampling operations: simplification and subdivision, and an aliasing issue in mesh sampling
- If you are interested in geometry, there is an advanced course "*Geometry Processing*"
- We also highly recommend to check out these fascinating books:









Next

Camera Viewing Pipeline