Tutorial 3

Geometry

Computer Graphics

Summer Semester 2020

Ludwig-Maximilians-Universität München

Agenda

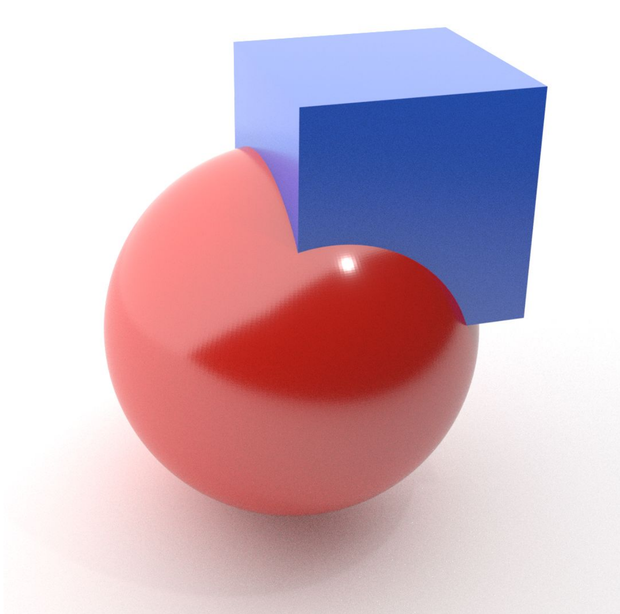
- Geometric Representations
  - Constructive Solid Geometry
  - Polygonal Mesh
- Bézier Curves and Interpolation
  - Bézier Curve
  - The de Casteljau Algorithm
  - Piecewise Bézier Curves
  - Bézier Patches
- Mesh Sampling
  - Mesh Simplification
  - Mesh Subdivision

Tutorial 3: Geometry

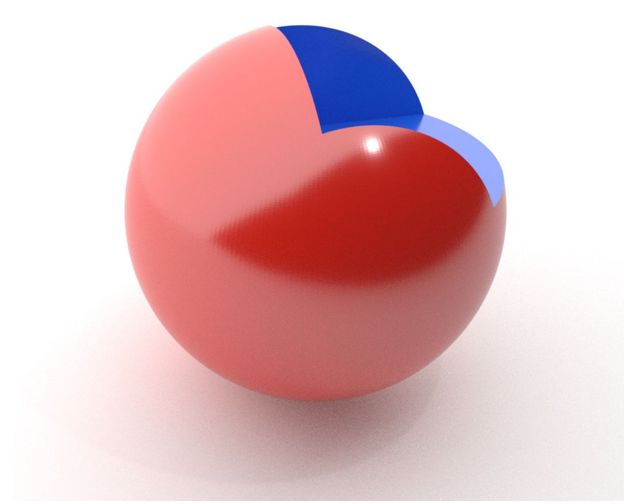
- Geometric Representations
  - Constructive Solid Geometry
  - Polygonal Mesh
- Bézier Curves and Interpolation
  - Bézier Curve
  - The de Casteljau Algorithm
  - Piecewise Bézier Curves
  - Bézier Patches
- Mesh Sampling
  - Mesh Simplification
  - Mesh Subdivision

Constructive Solid Geometry (CSG)

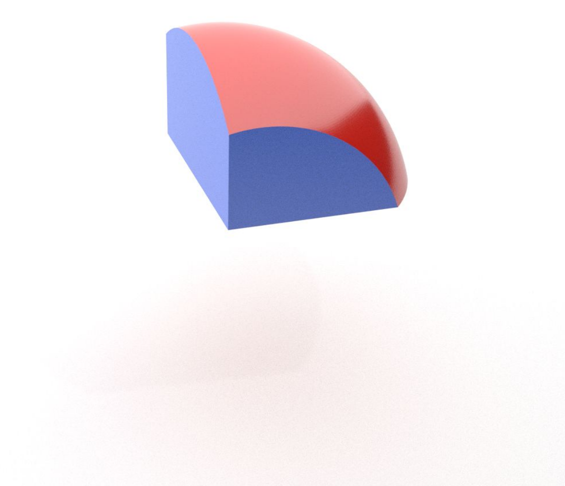
CSG allows to represent complex models as a series of **boolean operations** between **primitives**.



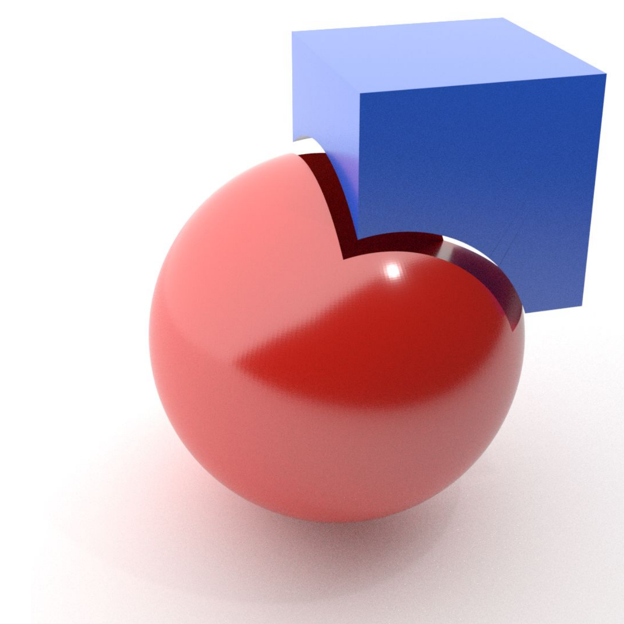
union
(OR)
$A \cup B$



difference
(NOT)
$A - B$

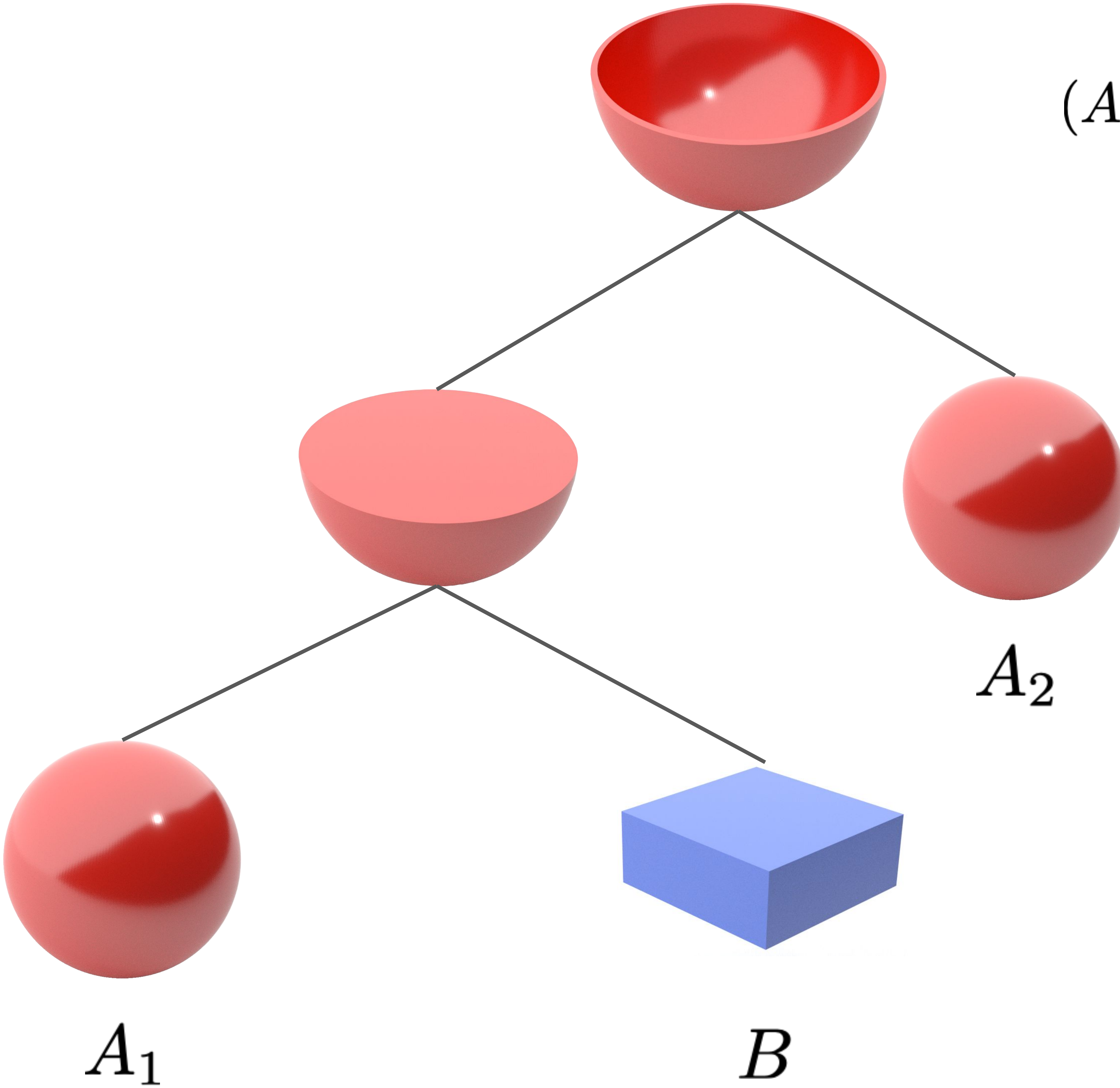


intersection
(AND)
$A \cap B$



exclusive or
(XOR)
$A \oplus B$

Task 1 a) Representation: *CSG Tree*



$$(A_1 - B) - A_2$$

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

Why CSG and Why not CSG?

- Why?
  - Minimum steps: represent solid objects as hierarchy of boolean operations
  - A lot easier to express some complex implicit surface
  - Less storage: due to the simple tree structure and primitives
  - Very easy to convert a CSG model to a polygonal mesh but not vise versa
  - …
- Why not?
  - Impossible to construct non-solid shape, e.g. organic models
  - Require a great deal of computation to derive boundaries, faces and edges ⇒ needed for interactive manipulation
  - ...

Polygonal Mesh

By definition, polygonal mesh is a collection of **vertices**, **edges** and **faces** that defines the shape of a **polyhedra** object.

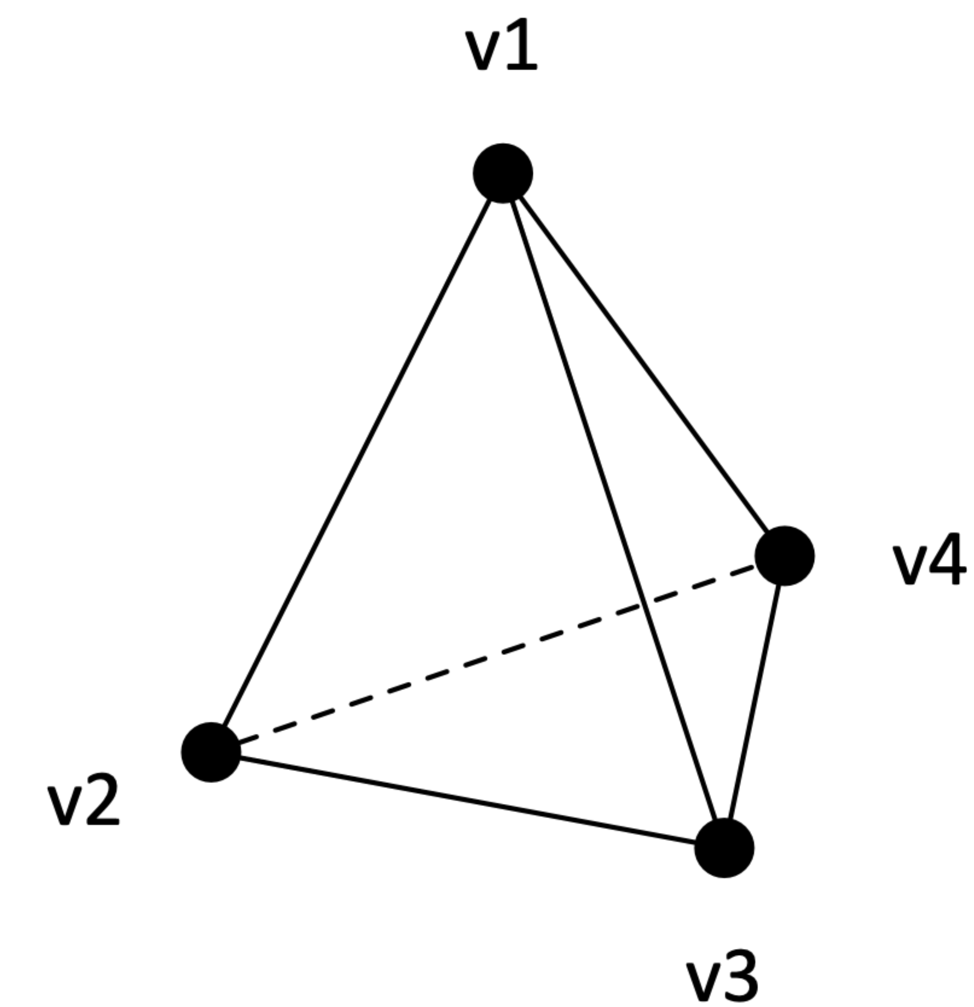
Task 1 b)

Face List

f1	v1 v3 v2
f2	v1 v4 v3
f3	v1 v2 v4
f4	v4 v2 v3

Vertex List

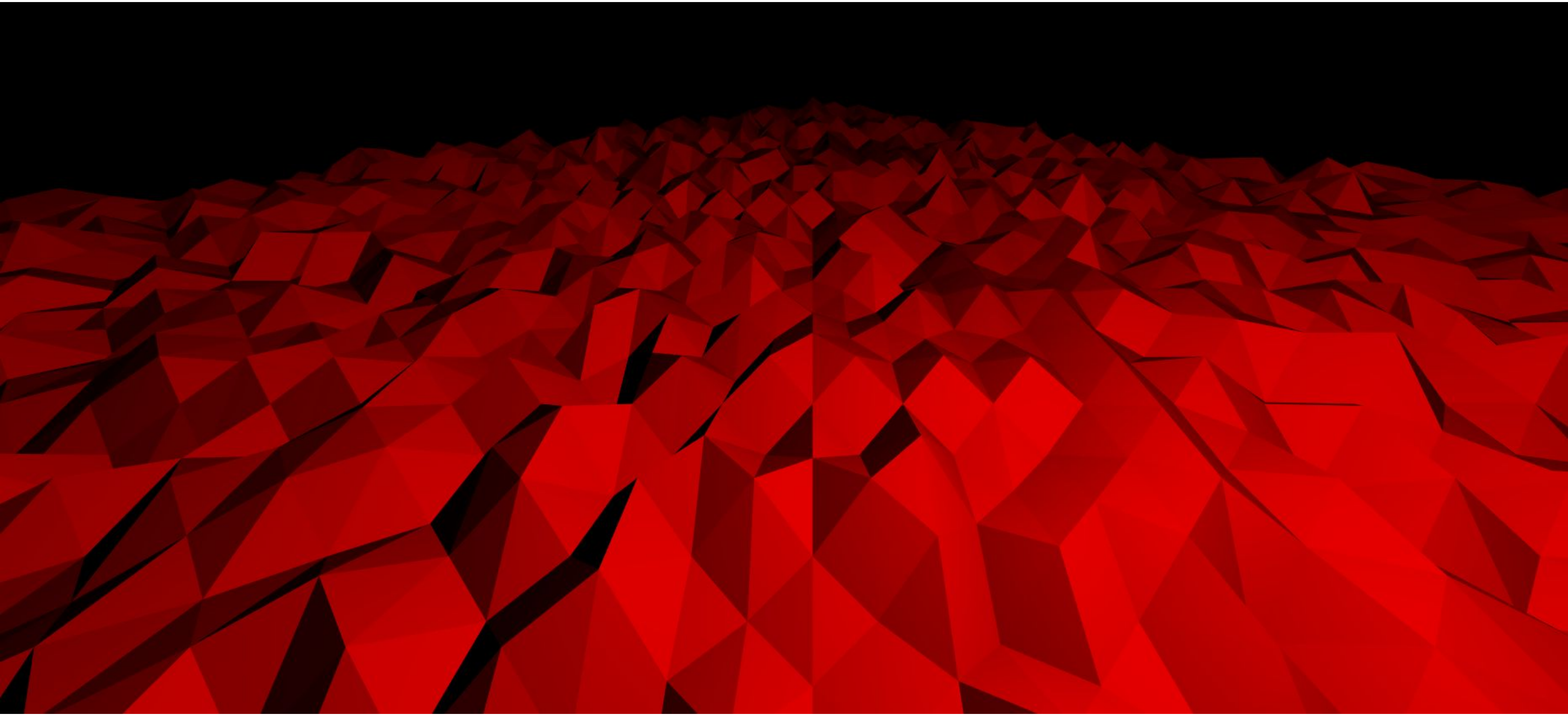
v1	f1 f2 f3
v2	f1 f3 f4
v3	f1 f4 f2
v4	f2 f4 f3



Q: What's the order when list vertices and faces? Which vertex and face should be listed first?

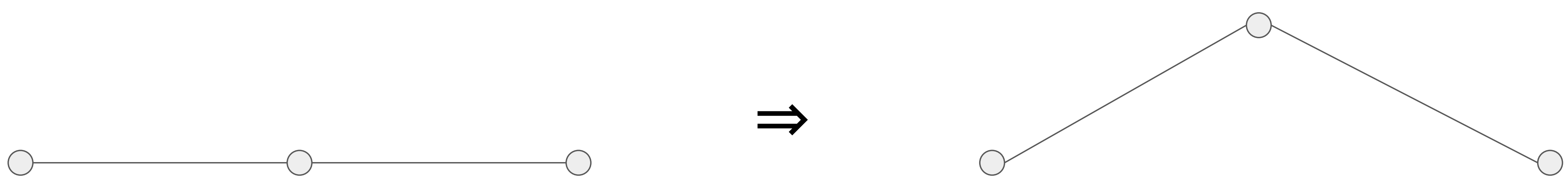
A: Depends. But the *order should be consistent* e.g. in .OBJ, it is counterclockwise.

Task 1 c) Apparently this is a mesh...



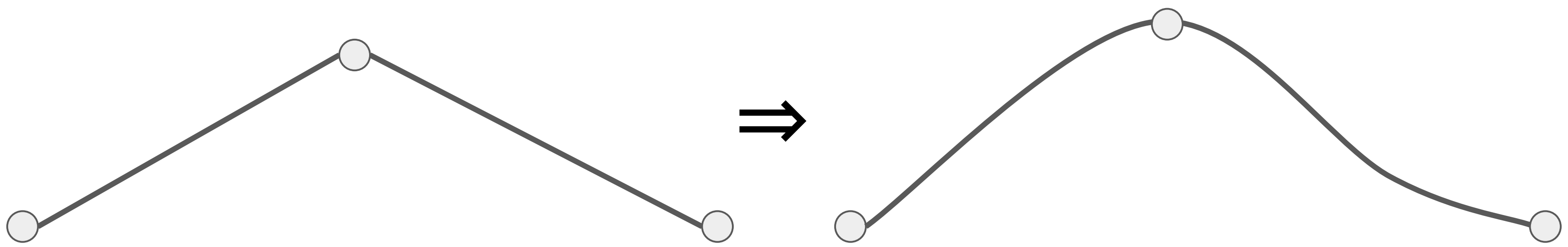
Task 1 d)

A hilly terrain can be derived from a x-y plane by changing the z value of each vertex. In three.js, one can use `PlaneGeometry`.

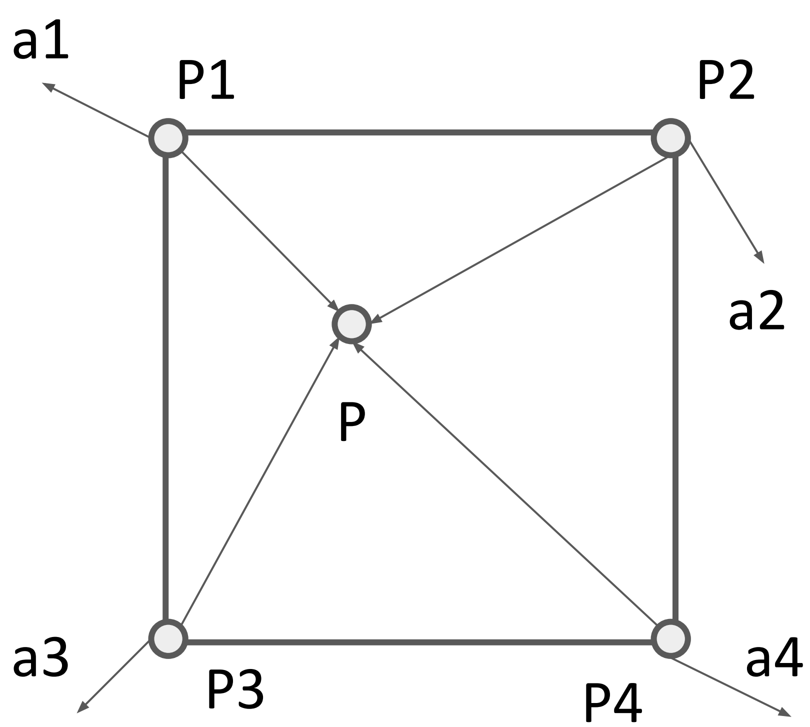


Perlin Noise

- Motivation: smoothly random interpolation

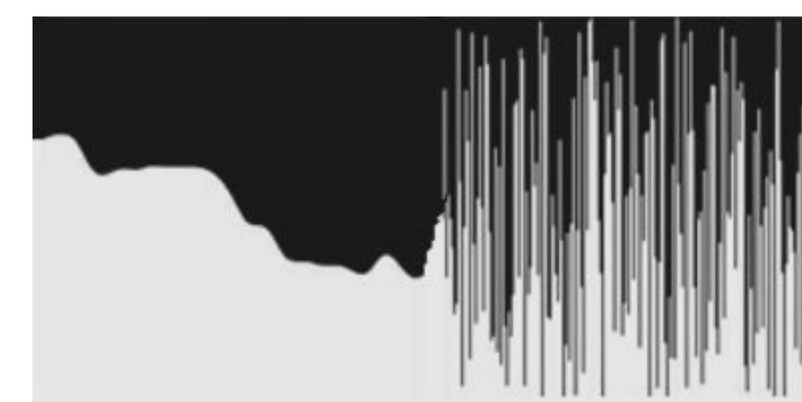


- How?



$$v_{P_i} = \mathbf{a}_i \overrightarrow{P_i P} (i = 1, 2, 3, 4)$$

Then P equals linear interpolation of P1-P4



Perlin v.s. random noise

Ken Perlin. 1985. ***An image synthesizer.*** SIGGRAPH Comput. Graph. 19, 3 (Jul. 1985), 287–296. DOI:https://doi.org/10.1145/325165.325247

Ken Perlin. 2002. ***Improving noise.*** ACM Trans. Graph. 21, 3 (July 2002), 681–682. DOI:https://doi.org/10.1145/566654.566636

Task 1 e)

```
export default class Terrain extends Renderer {
 ...
 init() {
   ...
   // TODO: Implement a terrain. Hint: use PerlinNoise.
   const l = new PointLight(params.lightColor, 1, 100)
   l.position.copy(params.lightPos)
   this.scene.add(l)

   const g = new PlaneGeometry(params.size, params.size, params.fragment, params.fragment)
   const plane = new Mesh(g, new MeshStandardMaterial({flatShading: true, side: DoubleSide}))
   plane.rotateX(Math.PI/2)




    this.scene.add(plane)
 }
}
```

Q: What happens if you don't give these two parameters?
`flatShading`: make sure color doesn't change on a single face
`DoubleSide`: the plane is colored on both sides

You will learn more about shading behaviors in the future lectures.

Task 1 e)

```
export default class Terrain extends Renderer {
 ...
 init() {
   ...
   // TODO: Implement a terrain. Hint: use PerlinNoise.
   const l = new PointLight(params.lightColor, 1, 100)
   l.position.copy(params.lightPos)
   this.scene.add(l)

   const g = new PlaneGeometry(params.size, params.size, params.fragment, params.fragment)
   const plane = new Mesh(g, new MeshStandardMaterial({flatShading: true, side: DoubleSide}))
   plane.rotateX(Math.PI/2)

   const n = new PerlinNoise()
   for (let i = 0; i < g.vertices.length; i++) {
     g.vertices[i].z = 2*n.gen(g.vertices[i].x, g.vertices[i].y) // Add noise to z coordinate of each vertex
   }
   this.scene.add(plane)
 }
}
```

Task 1 f) Why triangles?

- The most basic polygon
- Other polygons can be turned into triangles
- Unique properties
- Guaranteed to be planar
- Well-defined interior (Q: How to check if a point is inside a triangle?)
- Easier to compute interaction with rays (***later in ray tracing***)
- … too many reasons!

Task 1 f) Why quadrilateral?

- Quad meshes is a lot easier for modeling smooth and deformable surface
- Converting quadrangles to triangles is a simple process
- Quad meshes have many sub-regions with grid-like connectivity (flow line or edge loop)
- Quad meshes are better for subdivisions than tri-meshes
- …

⇒ Many subdivided surfaces are quad meshes (spline surface, e.g. Bézier patches)

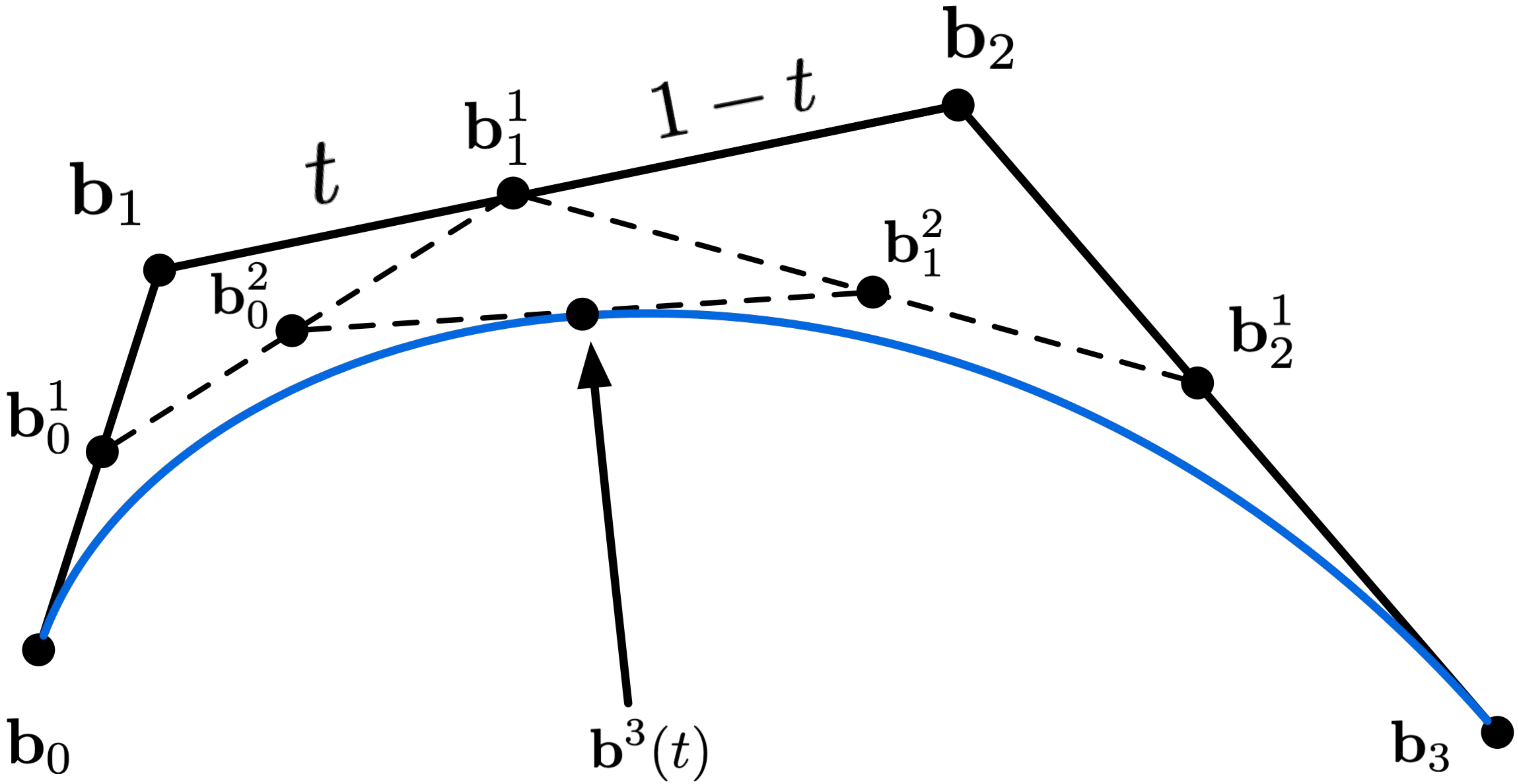
… *Bézier patches*?

Tutorial 3: Geometry

- Geometric Representations
  - Constructive Solid Geometry
  - Polygonal Mesh
- Bézier Curves and Interpolation
  - Bézier Curve
  - The de Casteljau Algorithm
  - Piecewise Bézier Curves
  - Bézier Patches
- Mesh Sampling
  - Mesh Simplification
  - Mesh Subdivision

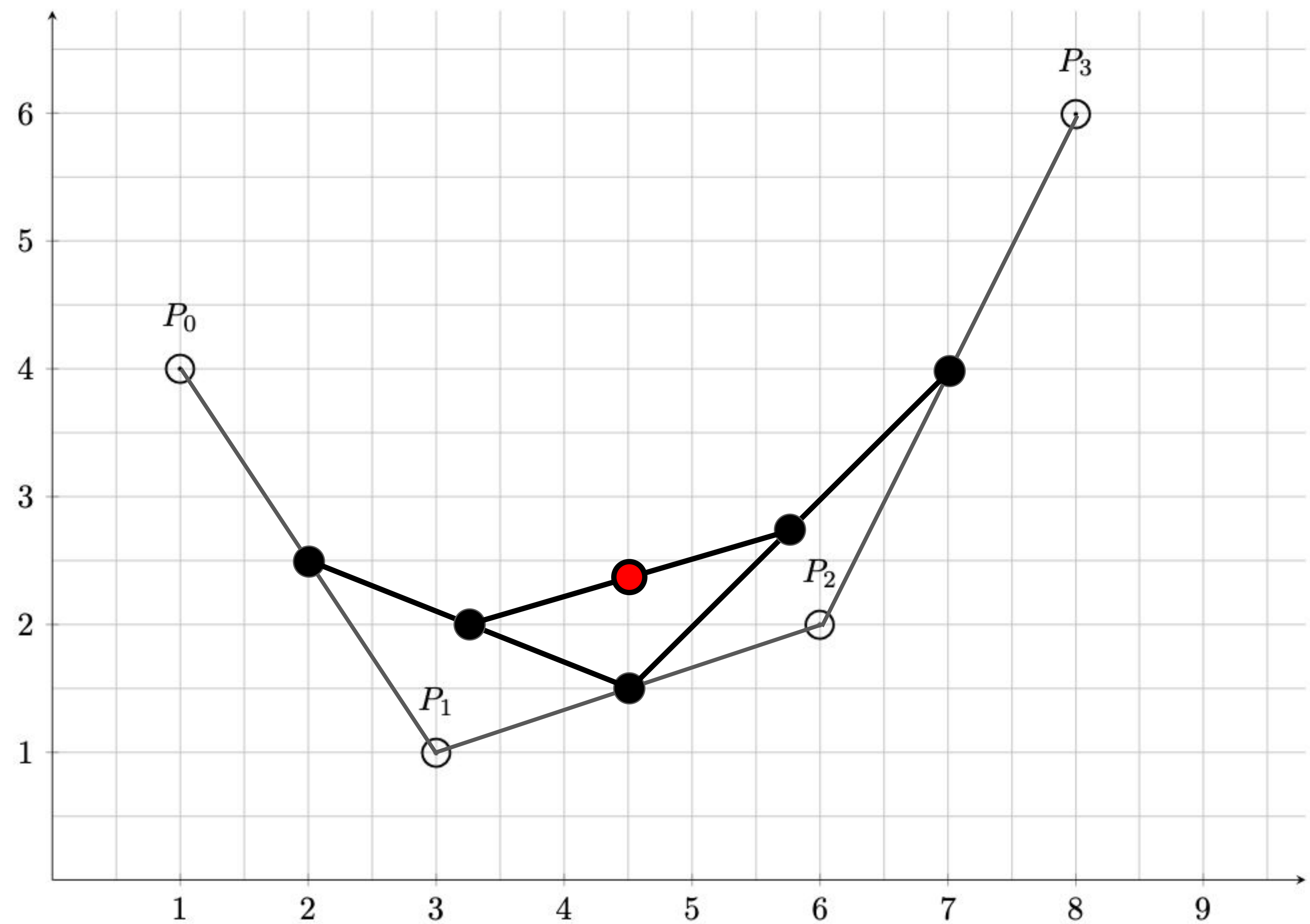
Cubic Bézier Curve - de Casteljau

4 control points

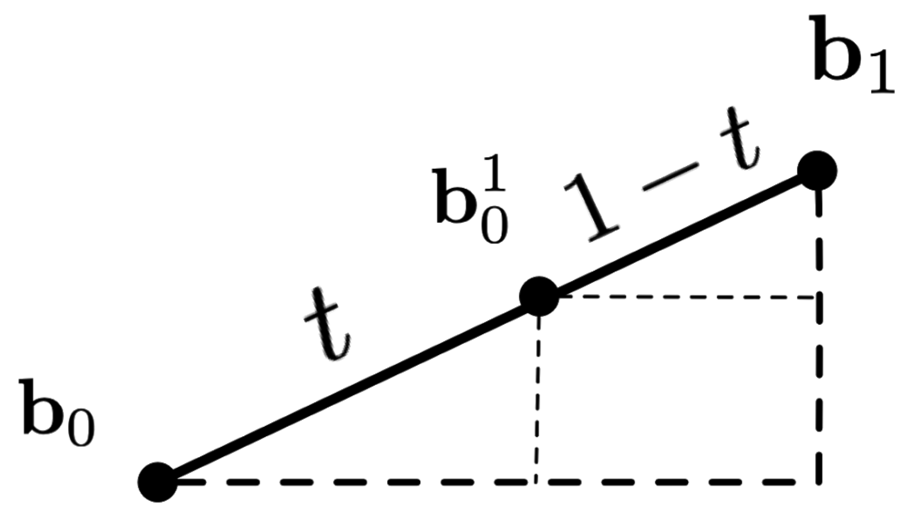


Task 2 a)

t = 0.5 ⇒ midpoint



Task 2 b)



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

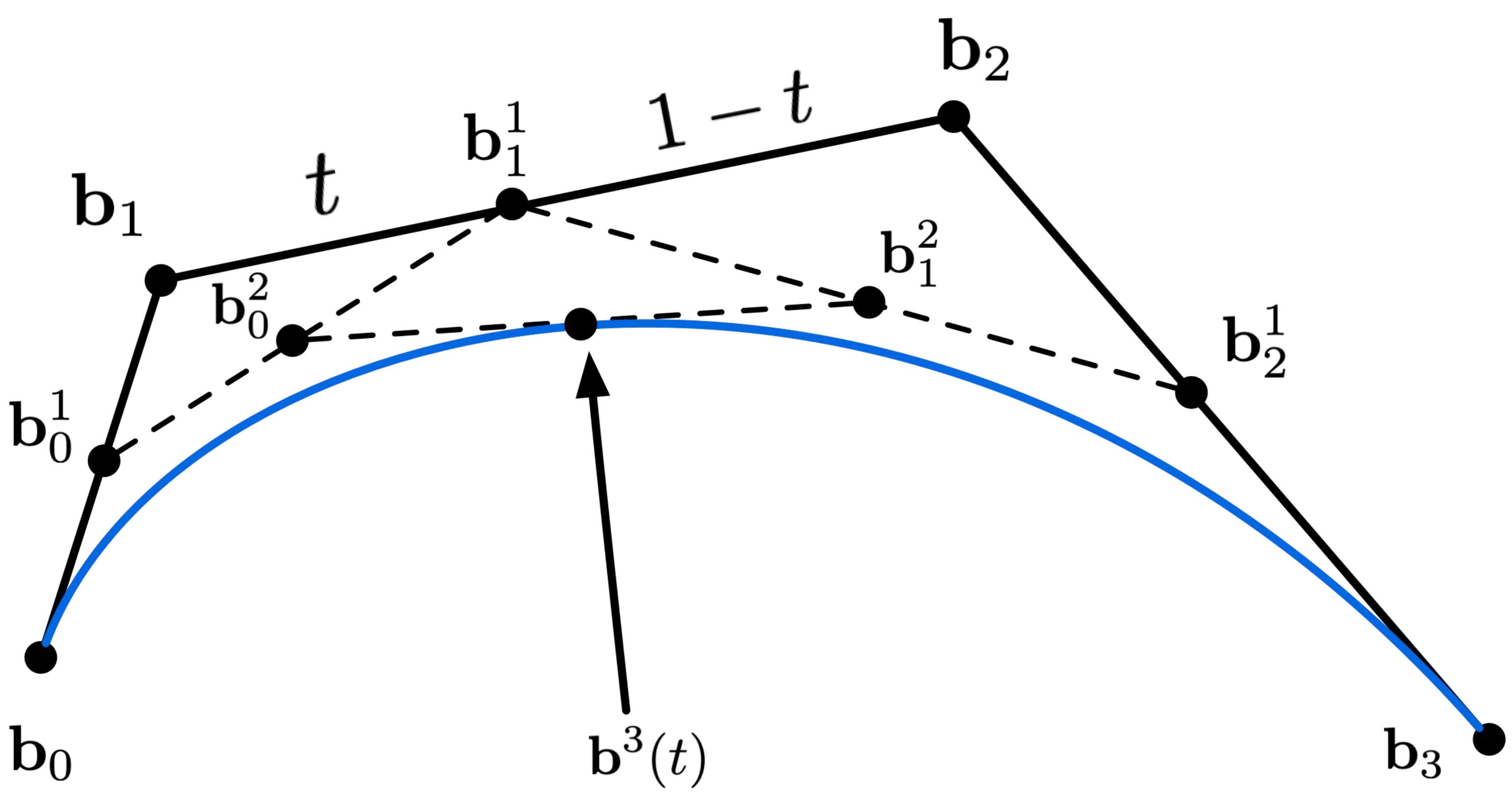
$$t = \frac{x - x_0}{x_1 - x_0} = \frac{y - y_0}{y_1 - y_0}$$

$$\Longrightarrow \quad x = x_0 + t(x_1 - x_0) = (1-t)x_0 + tx_1$$

$$y = y_0 + t(y_1 - y_0) = (1-t)y_0 + ty_1$$

Task 2 c) de Casteljau Algorithm

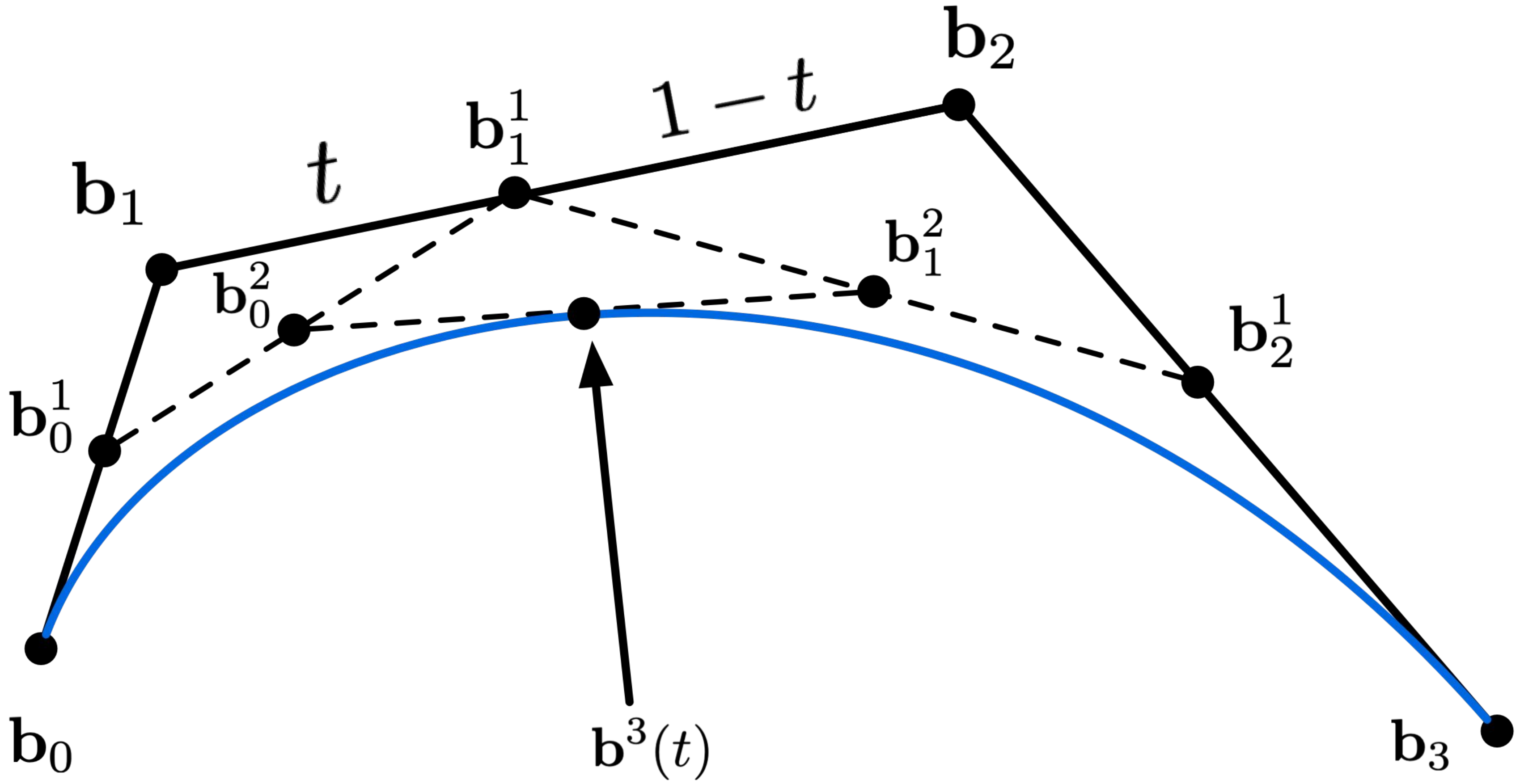
Take cubic Bézier as an example:

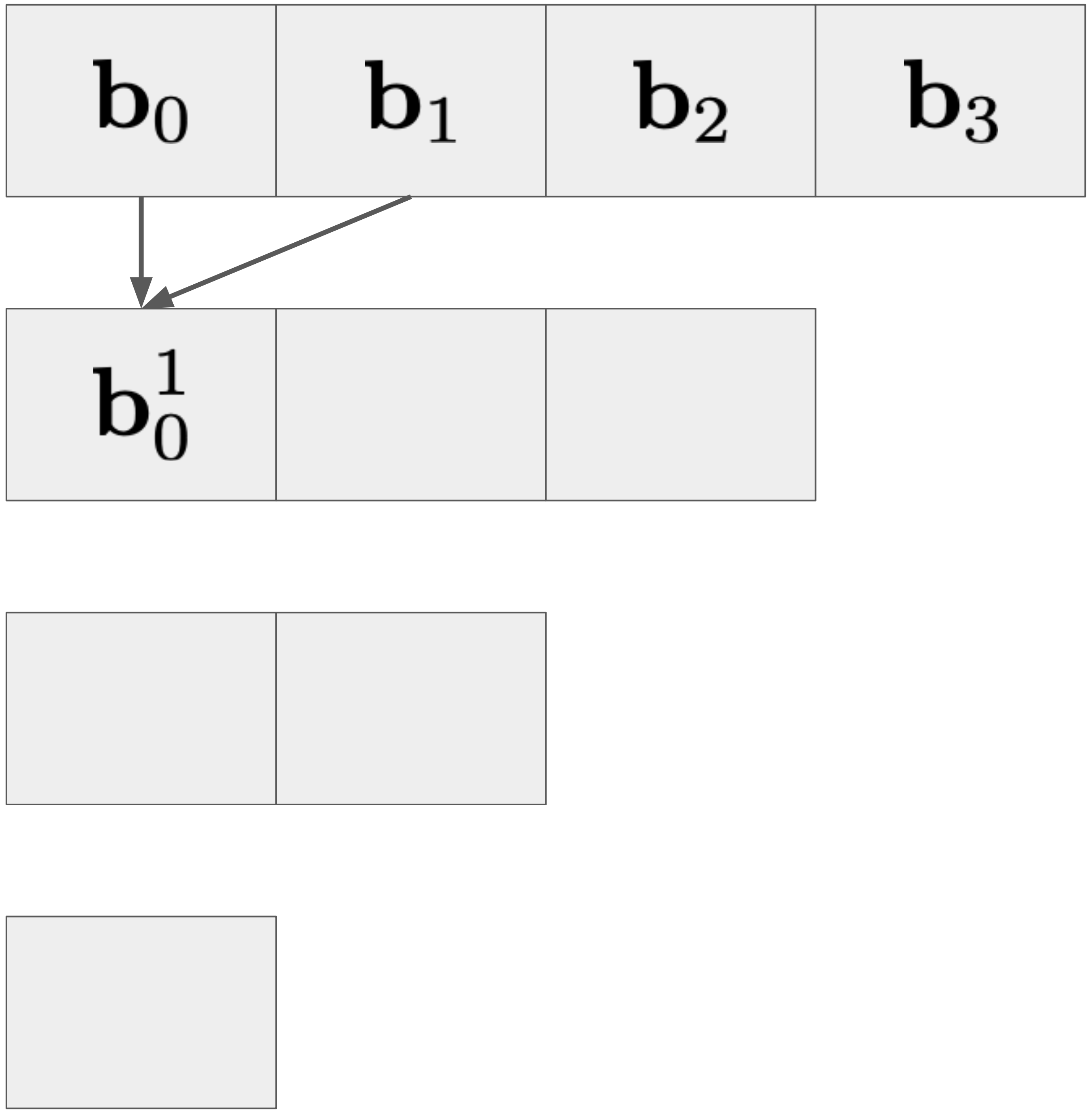


$\mathbf{b}_0$	$\mathbf{b}_1$	$\mathbf{b}_2$	$\mathbf{b}_3$

Task 2 c) de Casteljau Algorithm

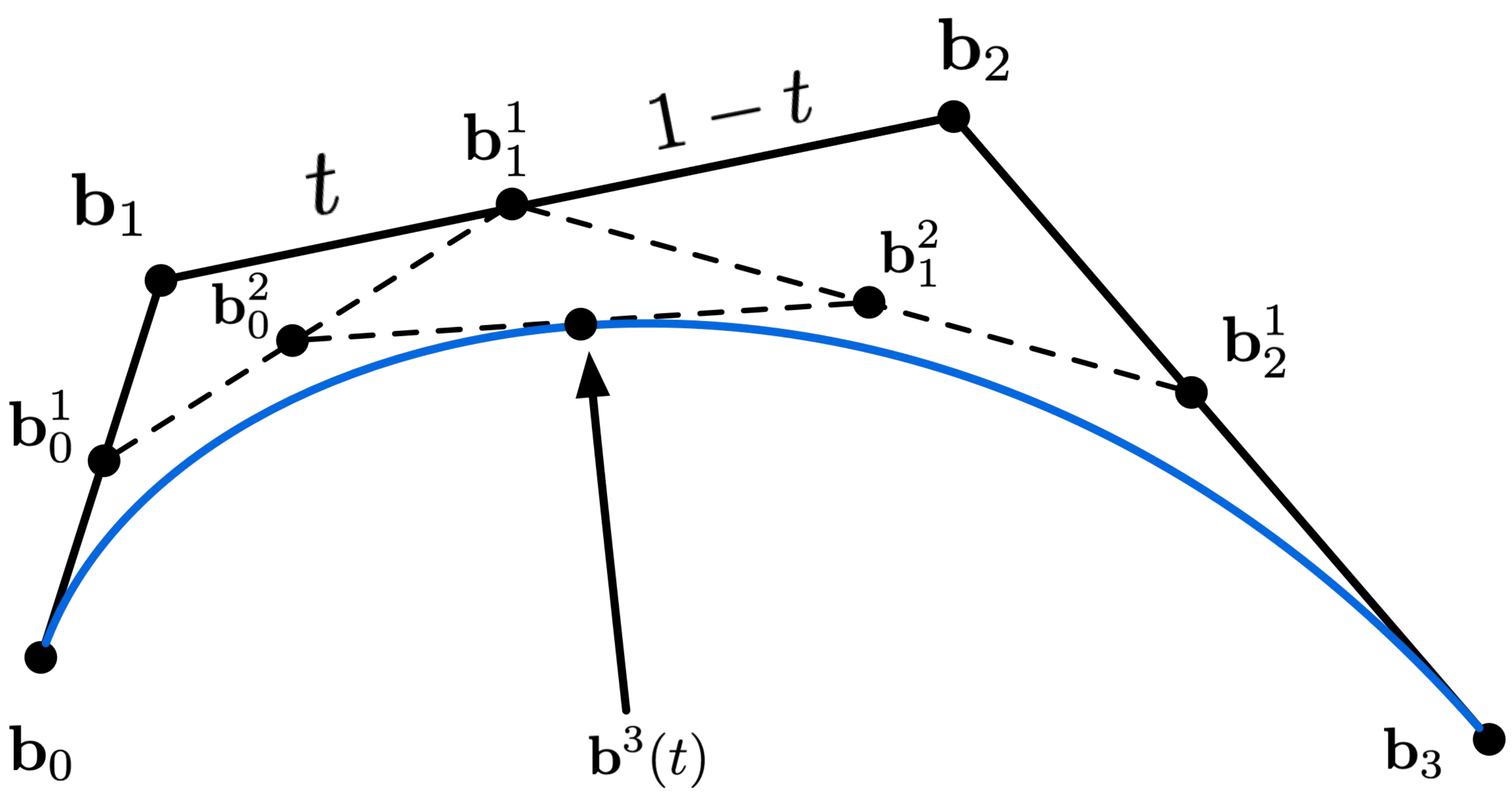
Take cubic Bézier as an example:

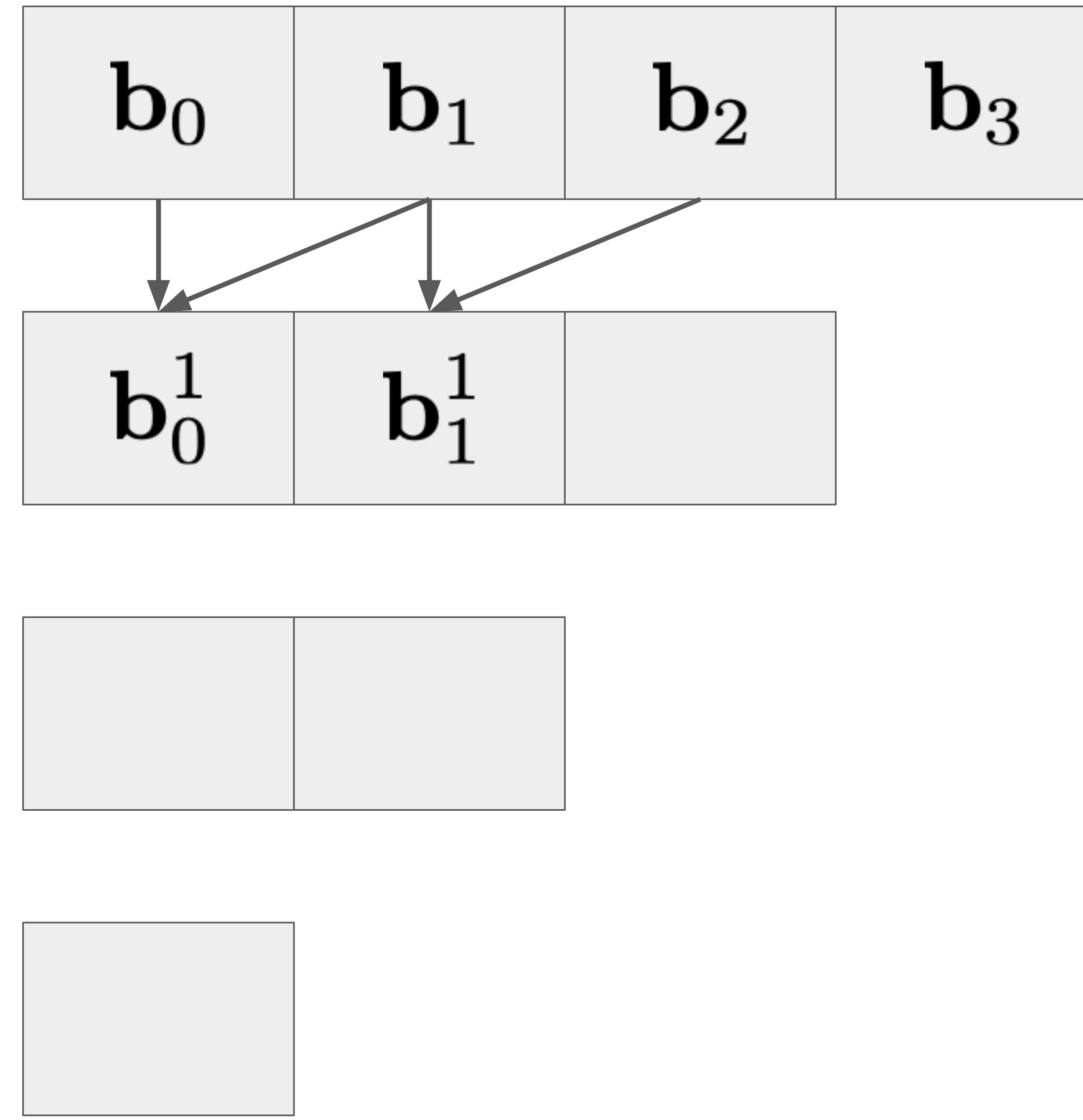




Task 2 c) de Casteljau Algorithm

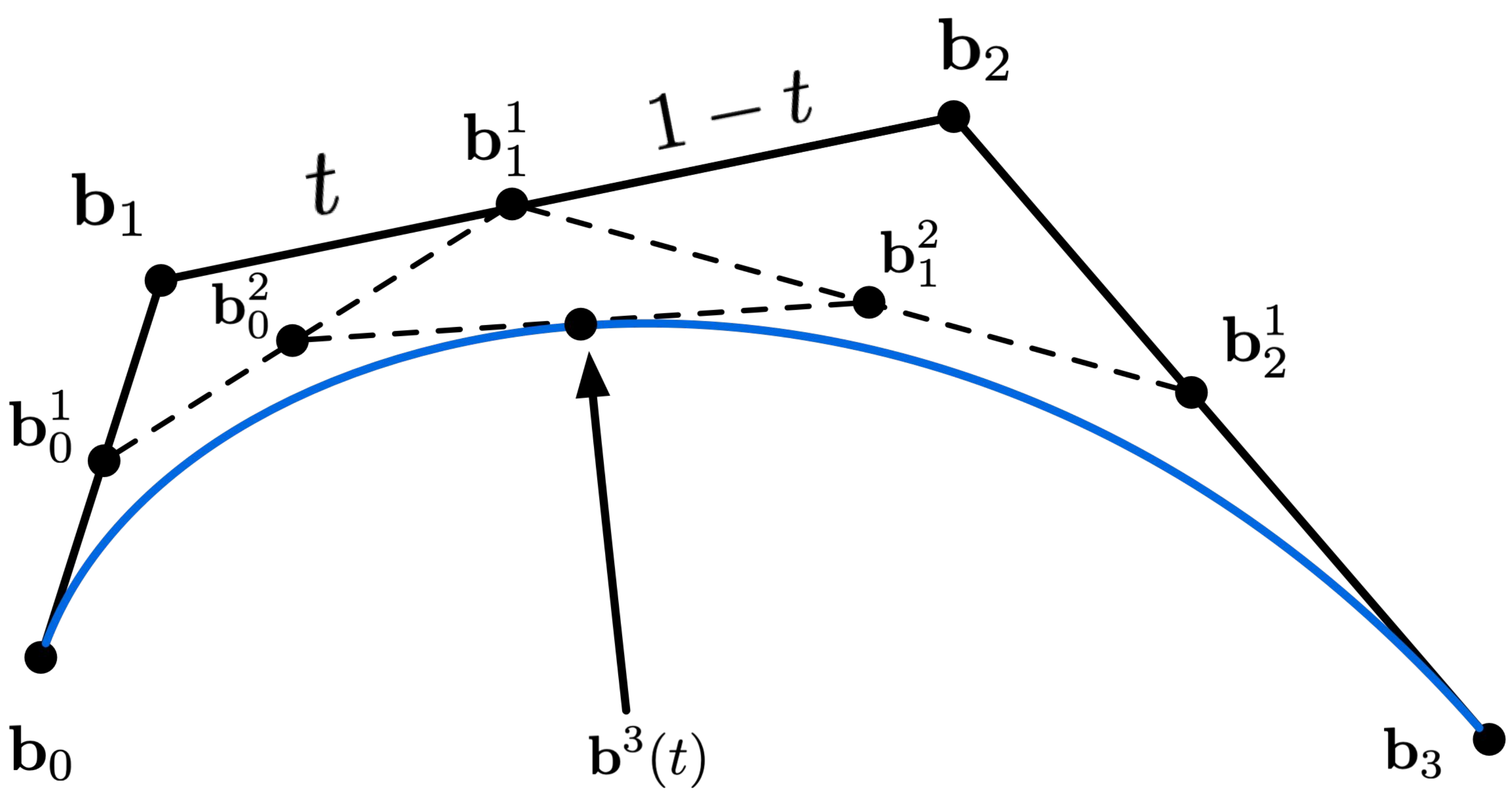
Take cubic Bézier as an example:

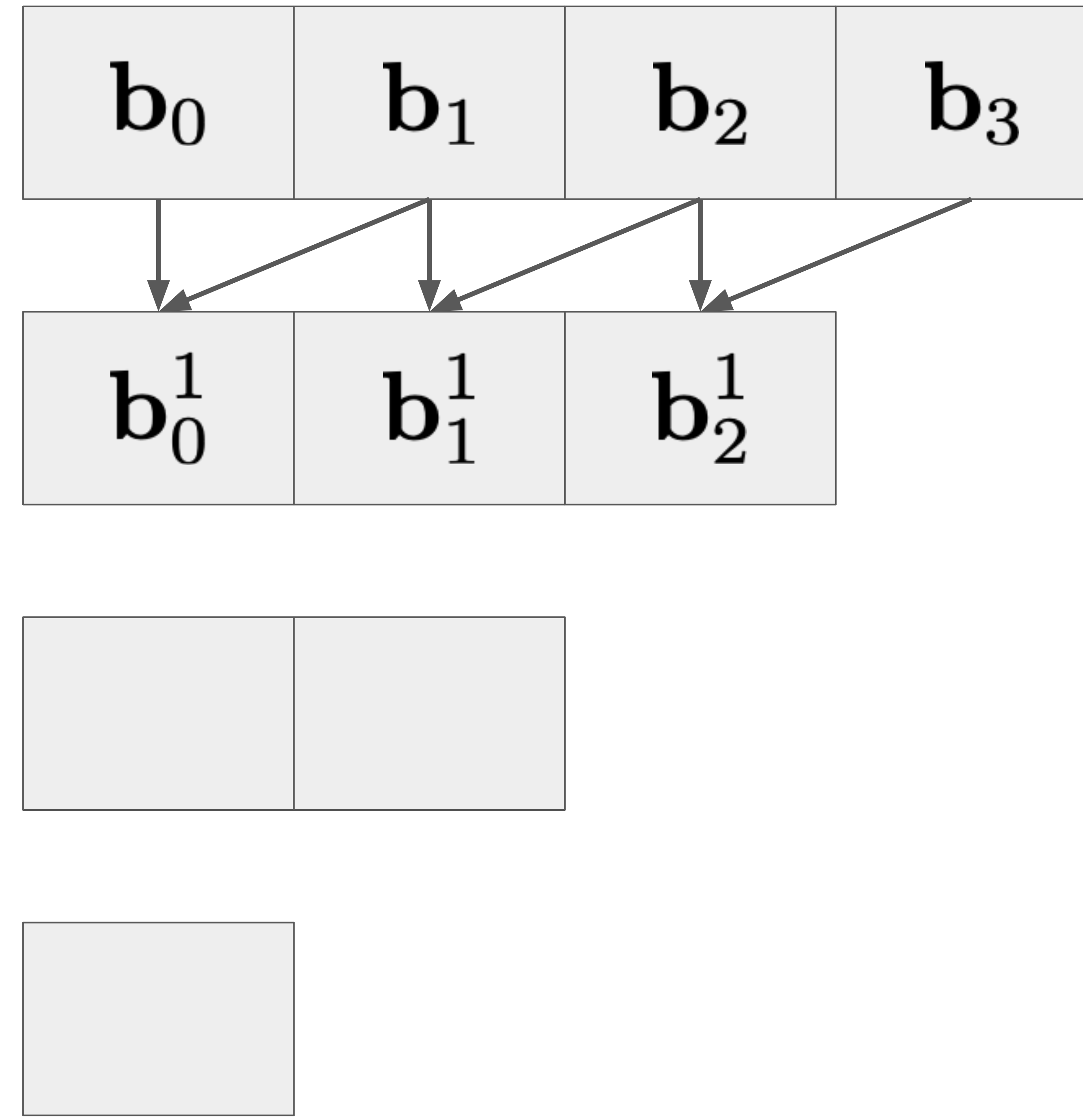




Task 2 c) de Casteljau Algorithm

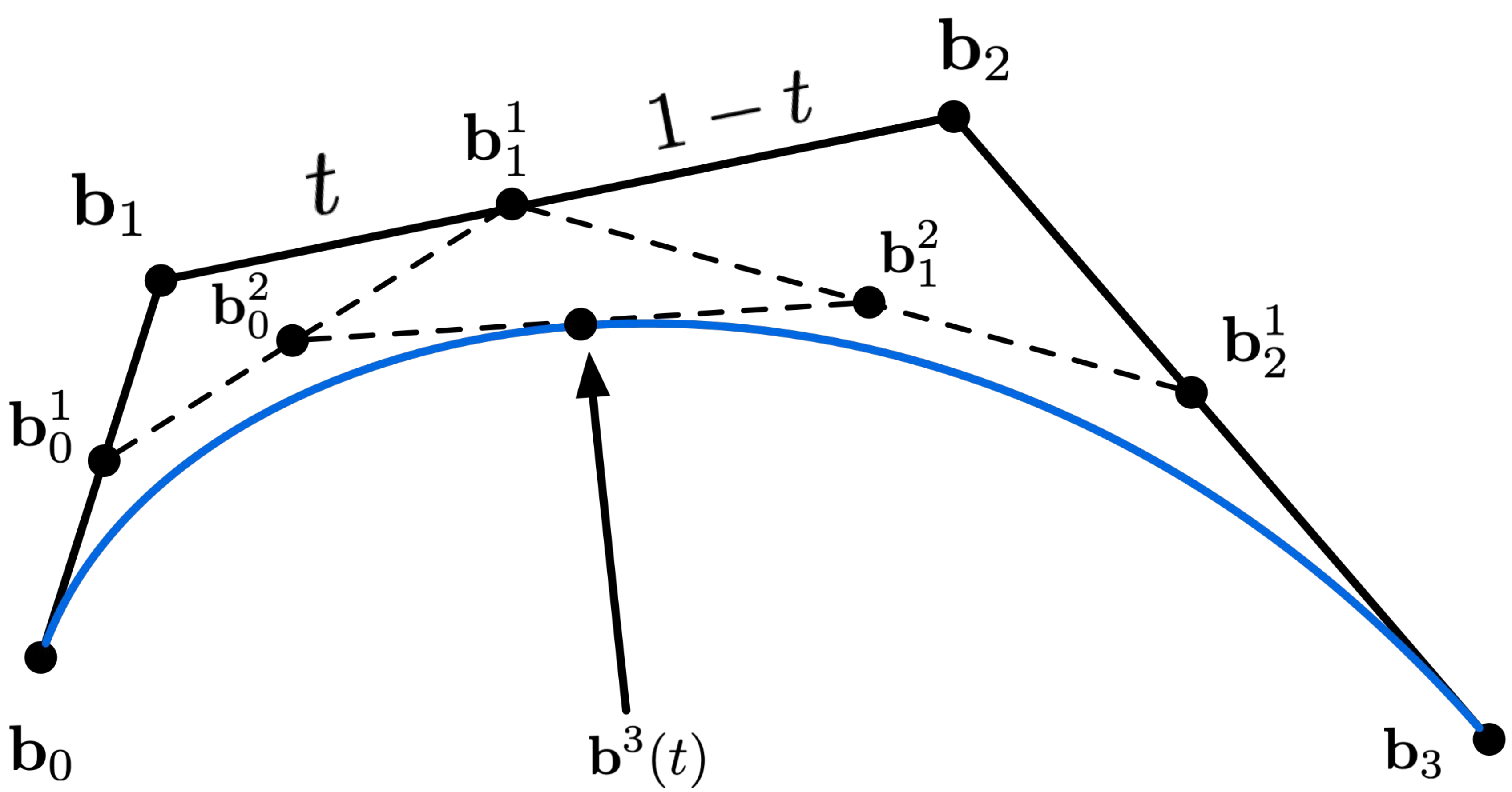
Take cubic Bézier as an example:

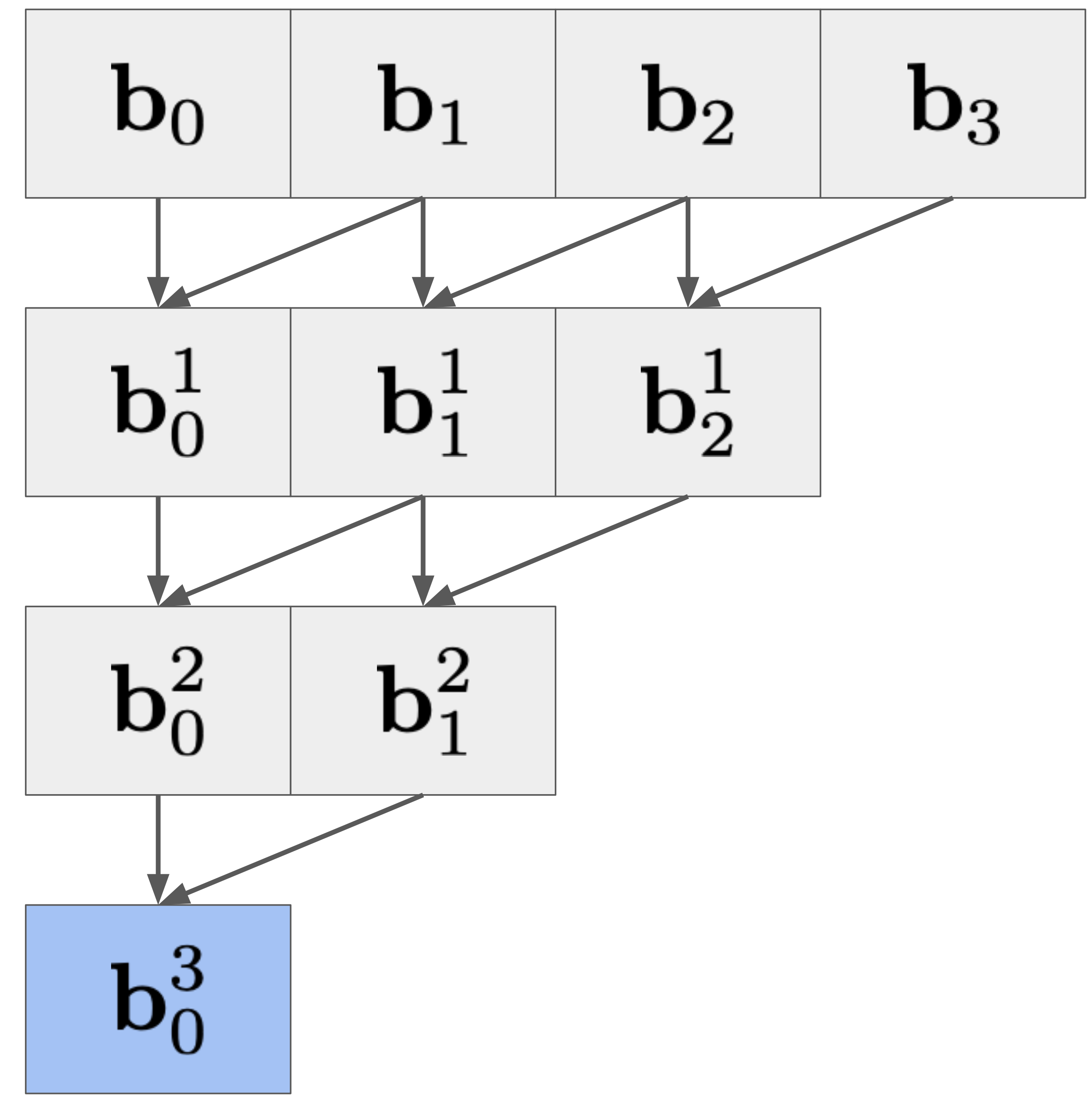




Task 2 c) de Casteljau Algorithm

Take cubic Bézier as an example:





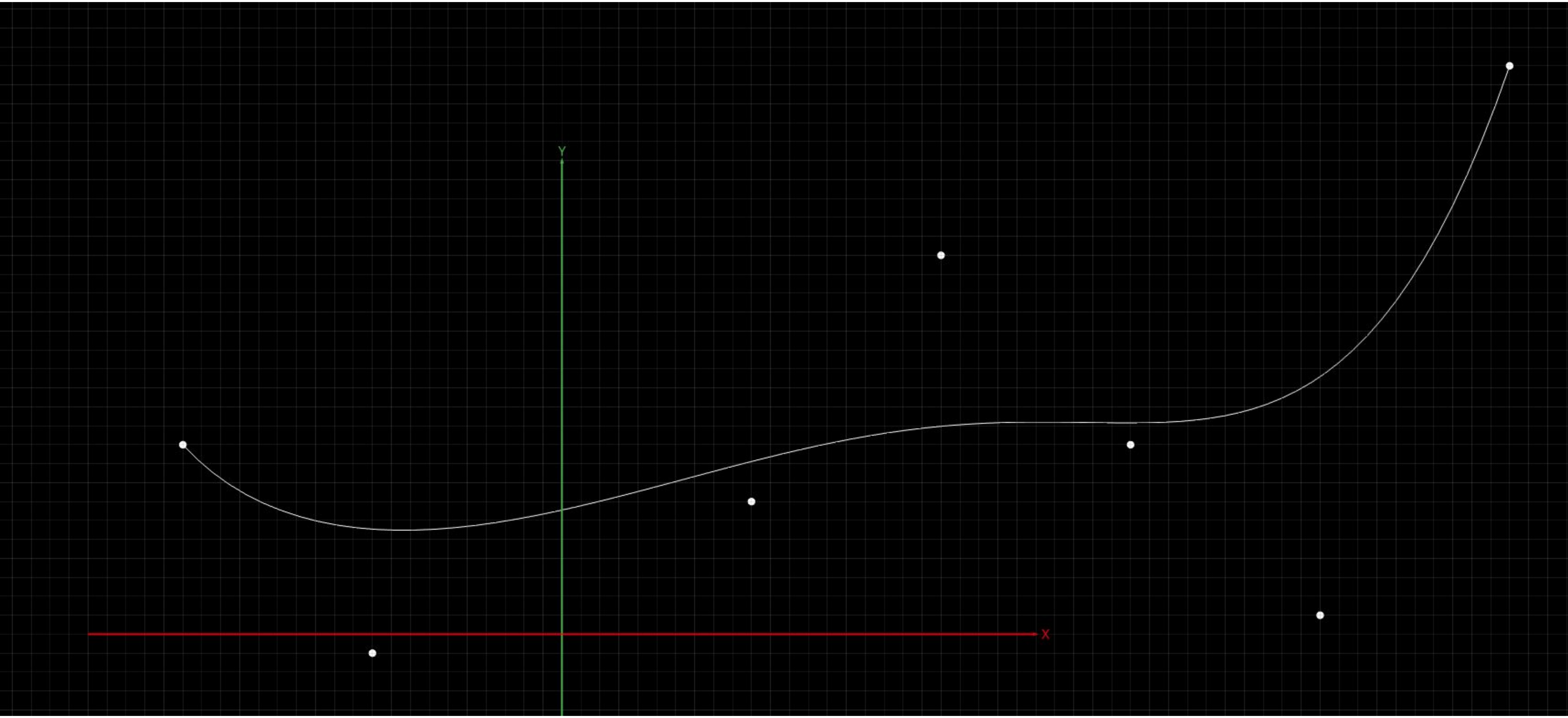
Task 2 c) de Casteljau Algorithm

```
createDeCasteljauPointAt(t) {
  // TODO: implement de Casteljau's algorithm
  // use this.controlPoints to access the given control points
  const n = this.controlPoints.length
  const tc = new Array(n)
  for(var i = 0; i < n; i++){
    tc[i] = this.controlPoints[i].clone()
  }
  for (let j = 0; j < n; j++) {
    for (let i = 0; i < n-j-1; i++) {
      tc[i].x = (1-t)*tc[i].x + t*tc[i+1].x
      tc[i].y = (1-t)*tc[i].y + t*tc[i+1].y
    }
  }
  return tc[0]
}
```

$$x = x_0 + t(x_1 - x_0) = (1-t)x_0 + tx_1$$

$$y = y_0 + t(y_1 - y_0) = (1-t)y_0 + ty_1$$

Task 2 c) de Casteljau Algorithm - Result



Bézier Curve - Algebraic Formula

Quadratic Bézier curve

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

Bézier Curve - Algebraic Formula

Quadratic Bézier curve

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

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

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

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

Cubic Bézier curve

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

Bézier Curve - Algebraic Formula

Quadratic Bézier curve

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

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

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

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

Cubic Bézier curve

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

...

General Bézier curve

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

Bernstein basis

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

combination

Task 2 d) Properties of Bézier Curves

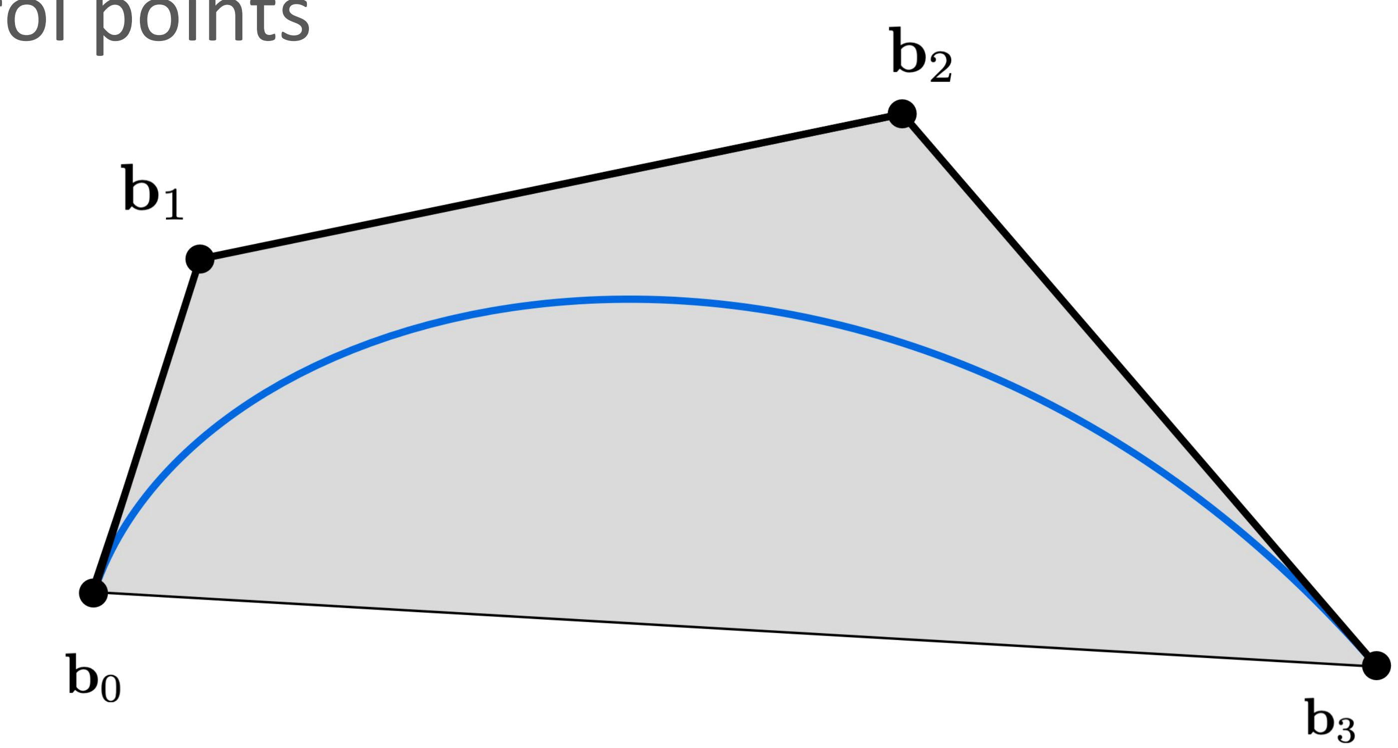
$$\mathbf{b}^n(t) = \sum_{i=0}^{n} B_i^n(t)\mathbf{b}_i$$

1. **Affine transform curve by transforming control points** (try to verify by yourself)

   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. Curve is within *convex hull* of control points



3. Interpolates endpoints

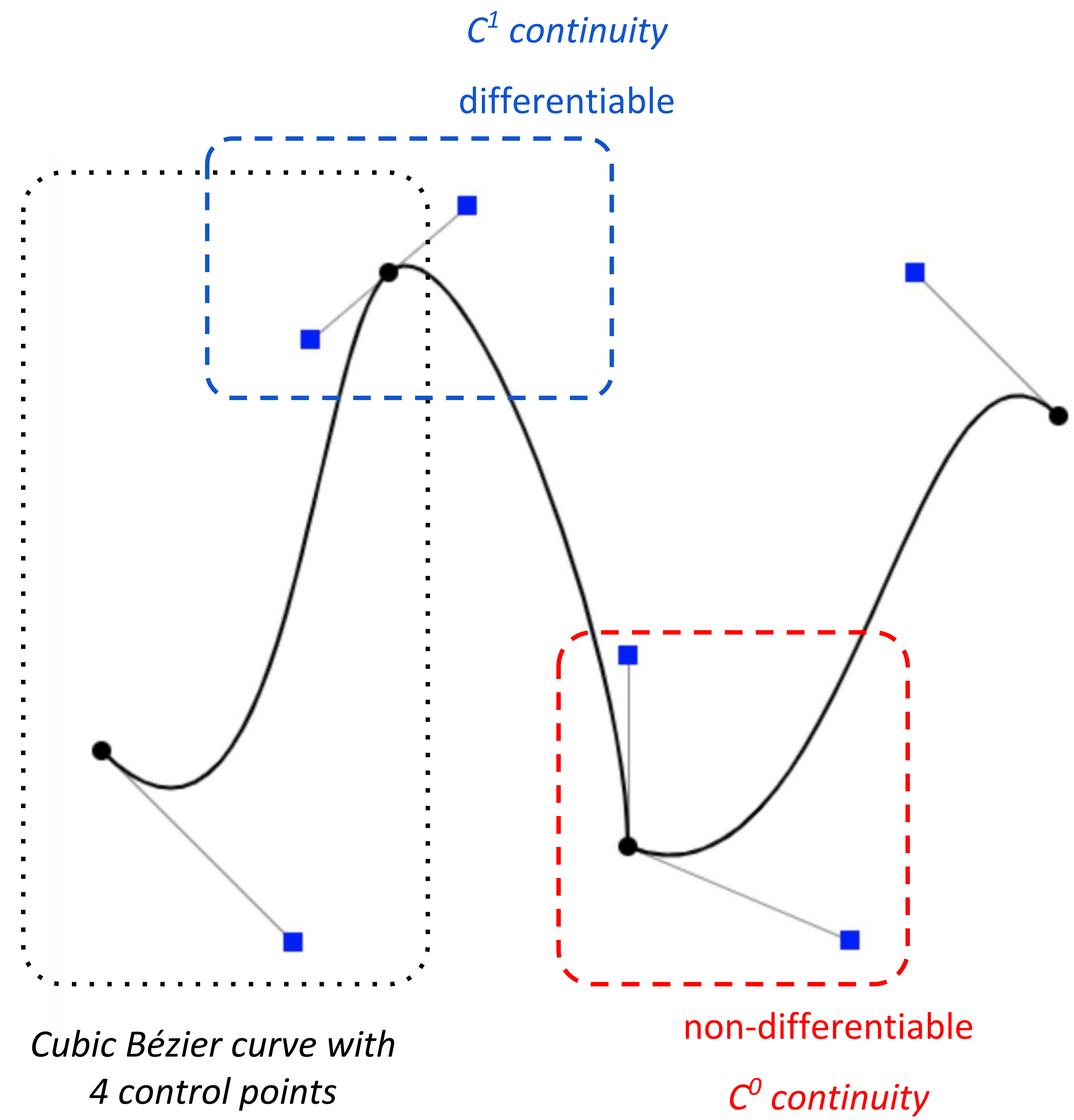
$$\mathbf{b}^n(0) = \sum_{i=0}^{n} B_i^n(0)\mathbf{b}_i = \mathbf{b}_0$$

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

Task 2 e) Piecewise Bézier Curves

- The Cubic Bézier curve with 4 control points is widely used (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***

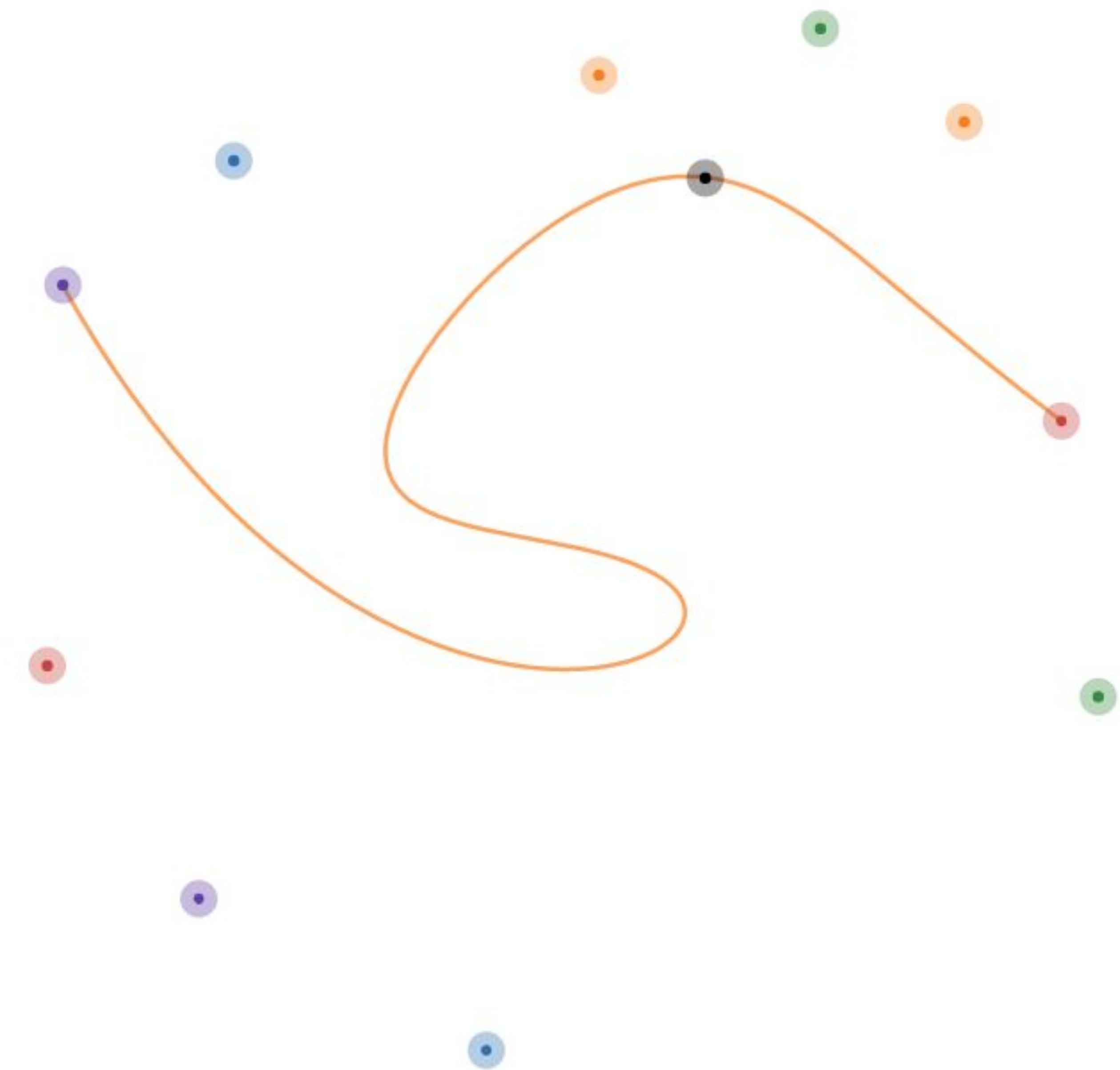
⇒ Left and right tangent slopes are equal for a connecting point



Task 2 f) Higher-order Bézier Curves

Very hard to control!

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



N-order Bézier Curve Playground:

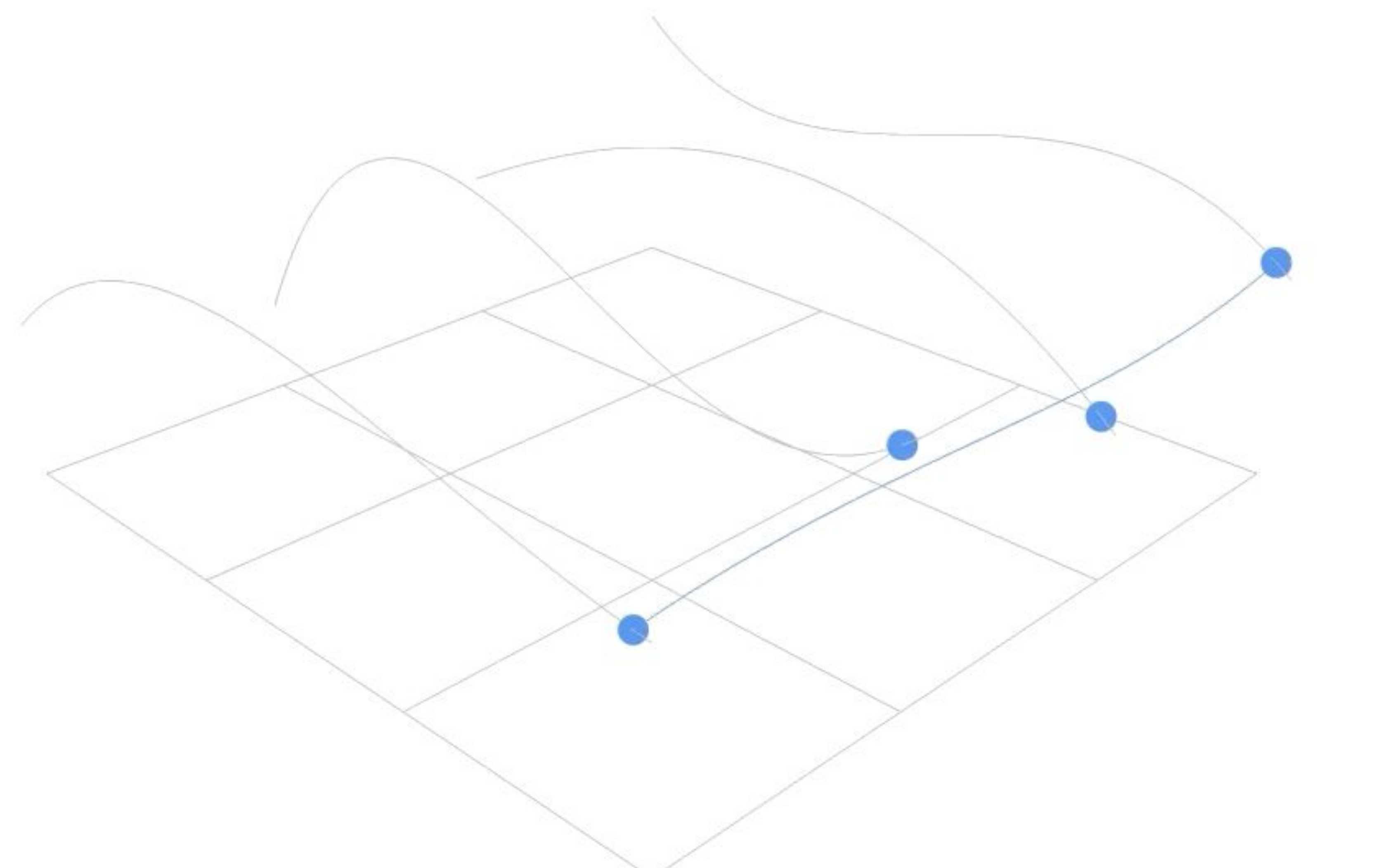
https://www.desmos.com/calculator/xlpbe9bgll

Task 2 g) Bicubic Bézier Surface (Patch)

4 cubic Bézier curves determines 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.



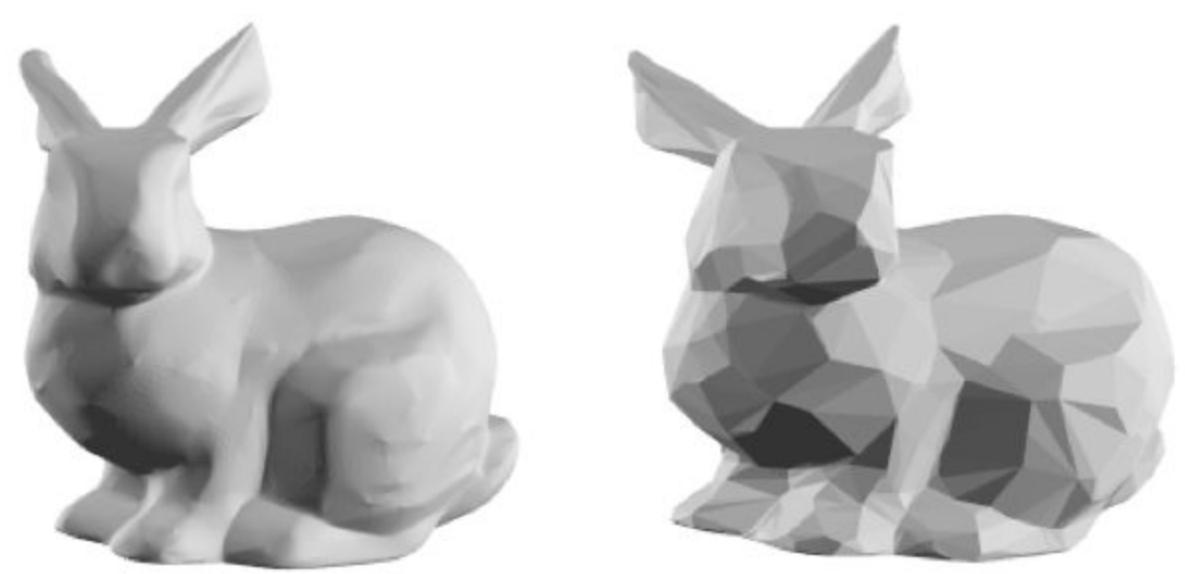
http://acko.net/blog/making-mathbox/

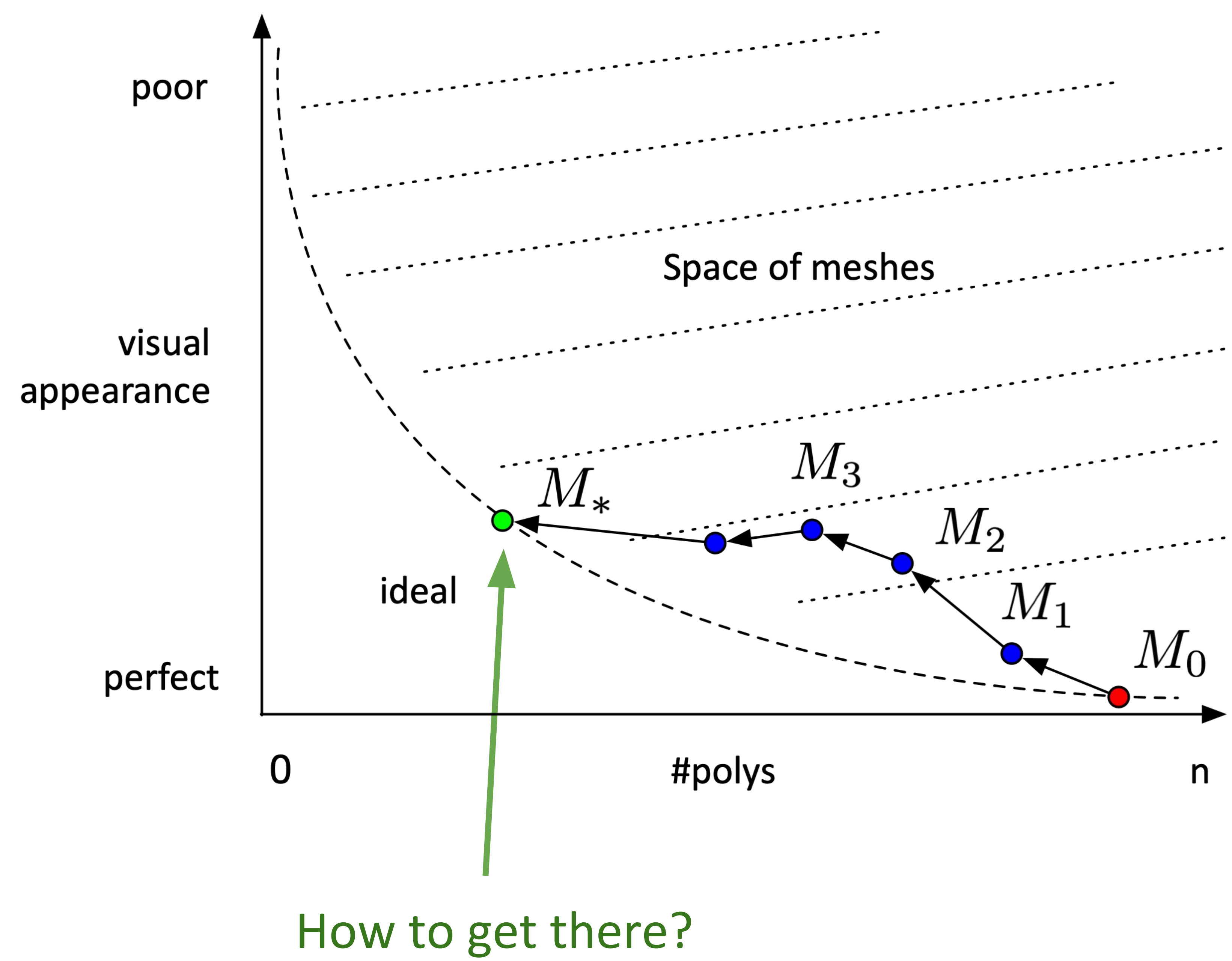
Tutorial 3: Geometry

- Geometric Representations
  - Constructive Solid Geometry
  - Polygonal Mesh
- Bézier Curves and Interpolation
  - Bézier Curve
  - The de Casteljau Algorithm
  - Piecewise Bézier Curves
  - Bézier Patches
- **Mesh Sampling**
  - Mesh Simplification
  - Mesh Subdivision

Mesh Simplification (downsample)

Reducing #polygons while *preserving the overall shape*



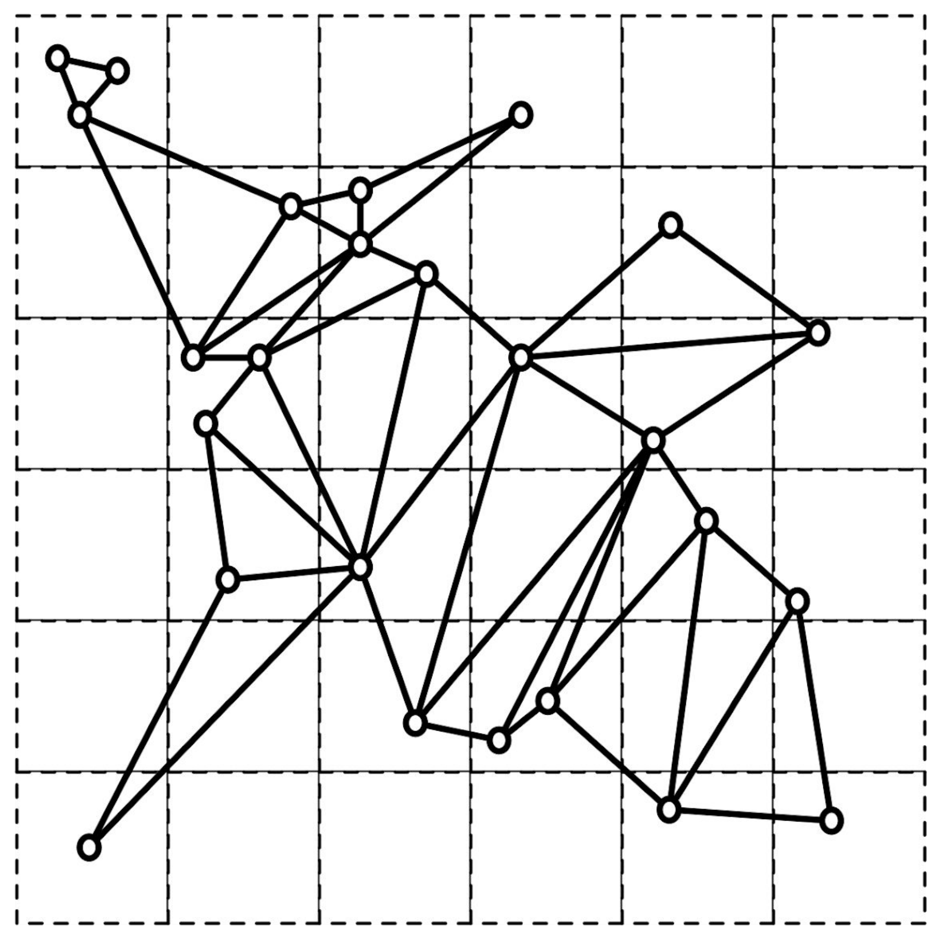


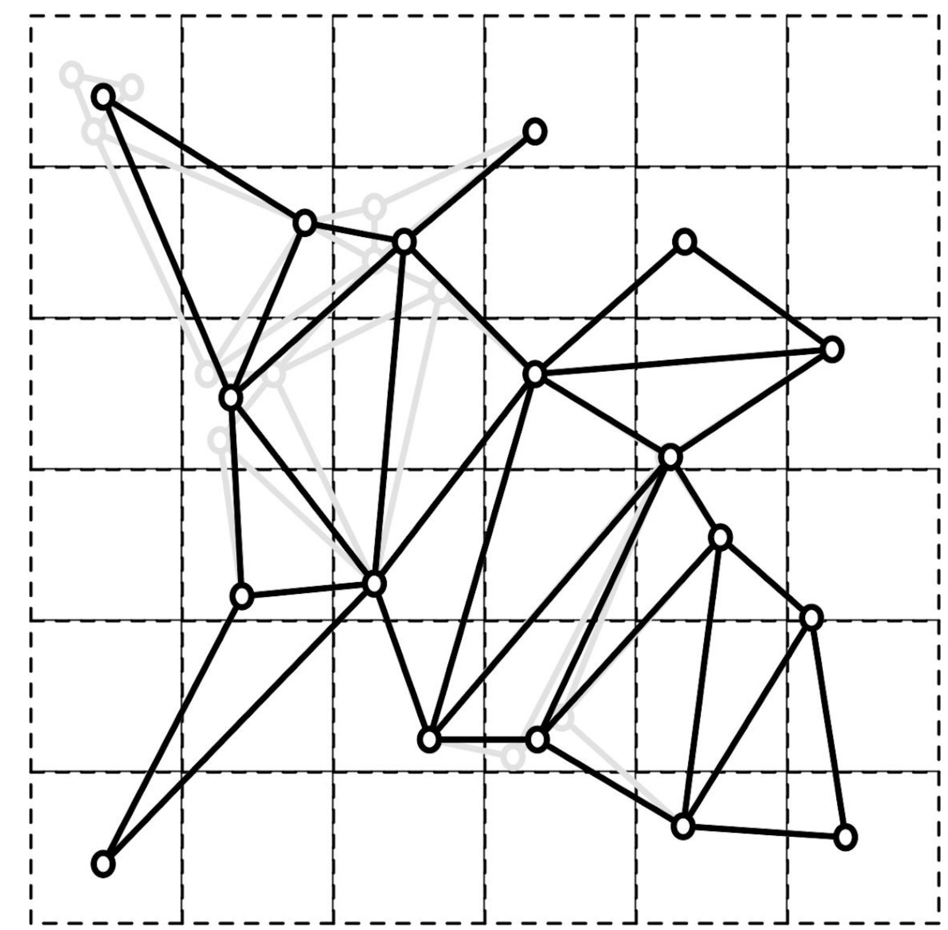
Mesh Simplification: Vertex Clustering

1. Divide 2D/3D space into grids
2. For each cell
   a. replace all nodes by their barycenter
   b. reconnect all edges to the barycenter

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.

Task 3 a) and b)

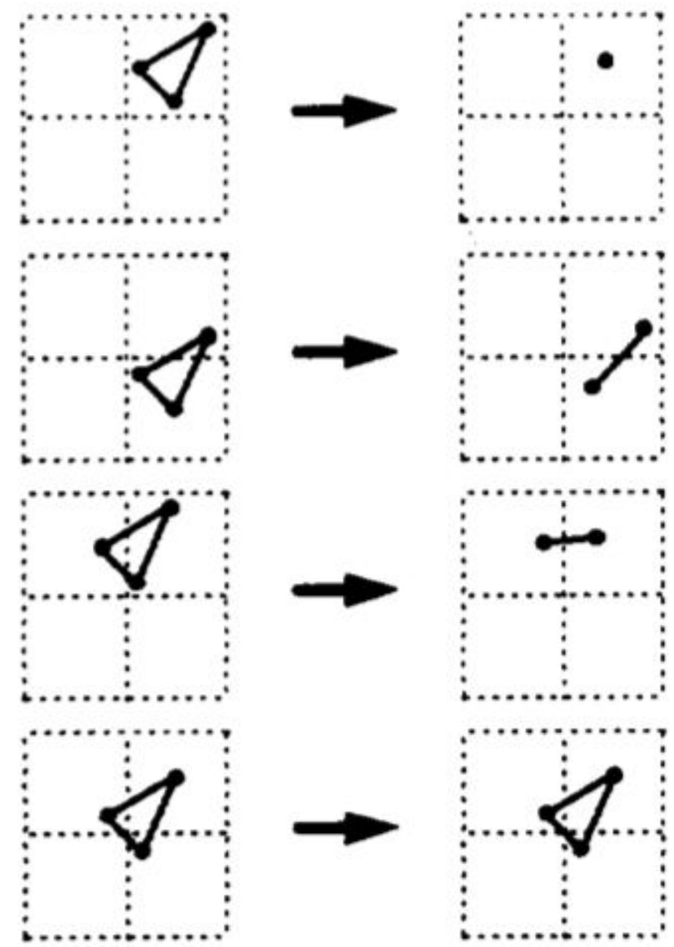




- Before simplification: #triangles = 22
- After simplification: #triangles = 15
- Reduction ratio = (before - after) / before = (22-15)/22 ≈ 31.8%

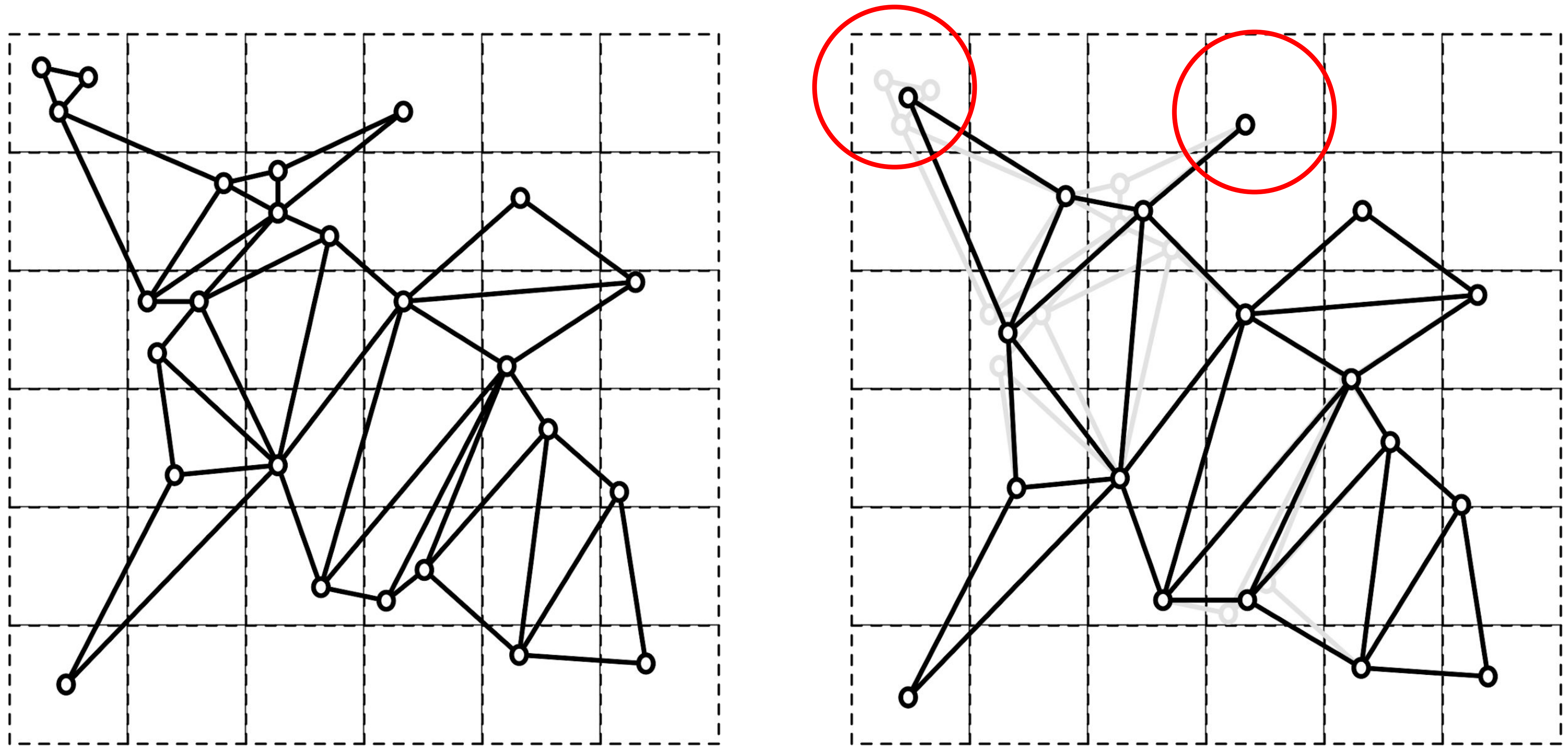
Vertex Clustering: Inconsistency

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



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). Association for Computing Machinery, New York, NY, USA, 75–ff. DOI:https://doi.org/10.1145/253284.253310

Task 3 c)



- If you are doing simplification, details will be lost for sure
- Major drawback: **geometric topology has changed**

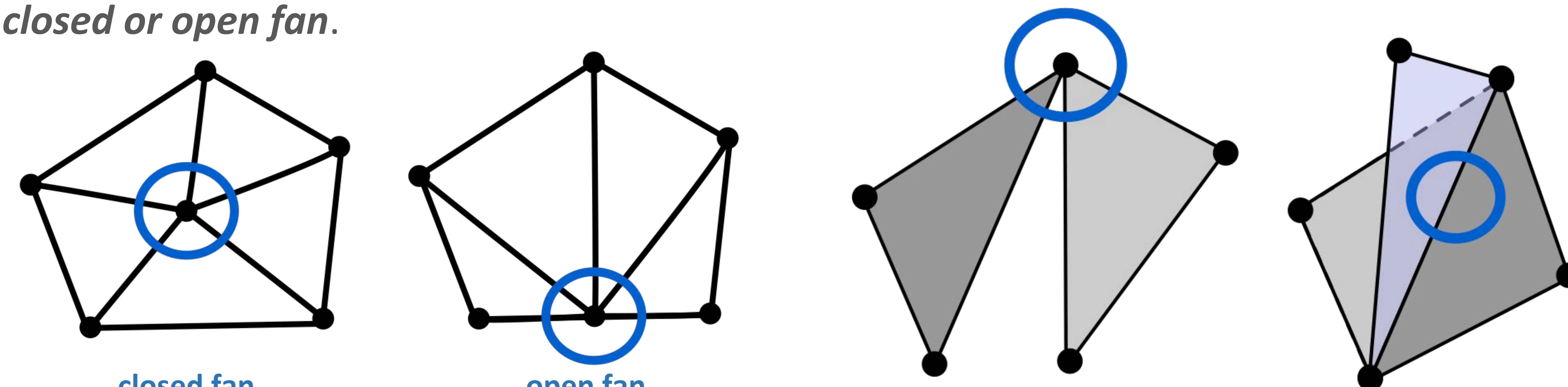
Geometry vs. Topology

Geometry: The vertex is at (x, y, z) ⇒ distance relevant

Topology: These vertices are connected ⇒ distance irrelevant

Manifold & Non-Manifold

Manifold: Each ***edge is incident to one or two faces***, and ***faces incident to a vertex from a closed or open fan***.

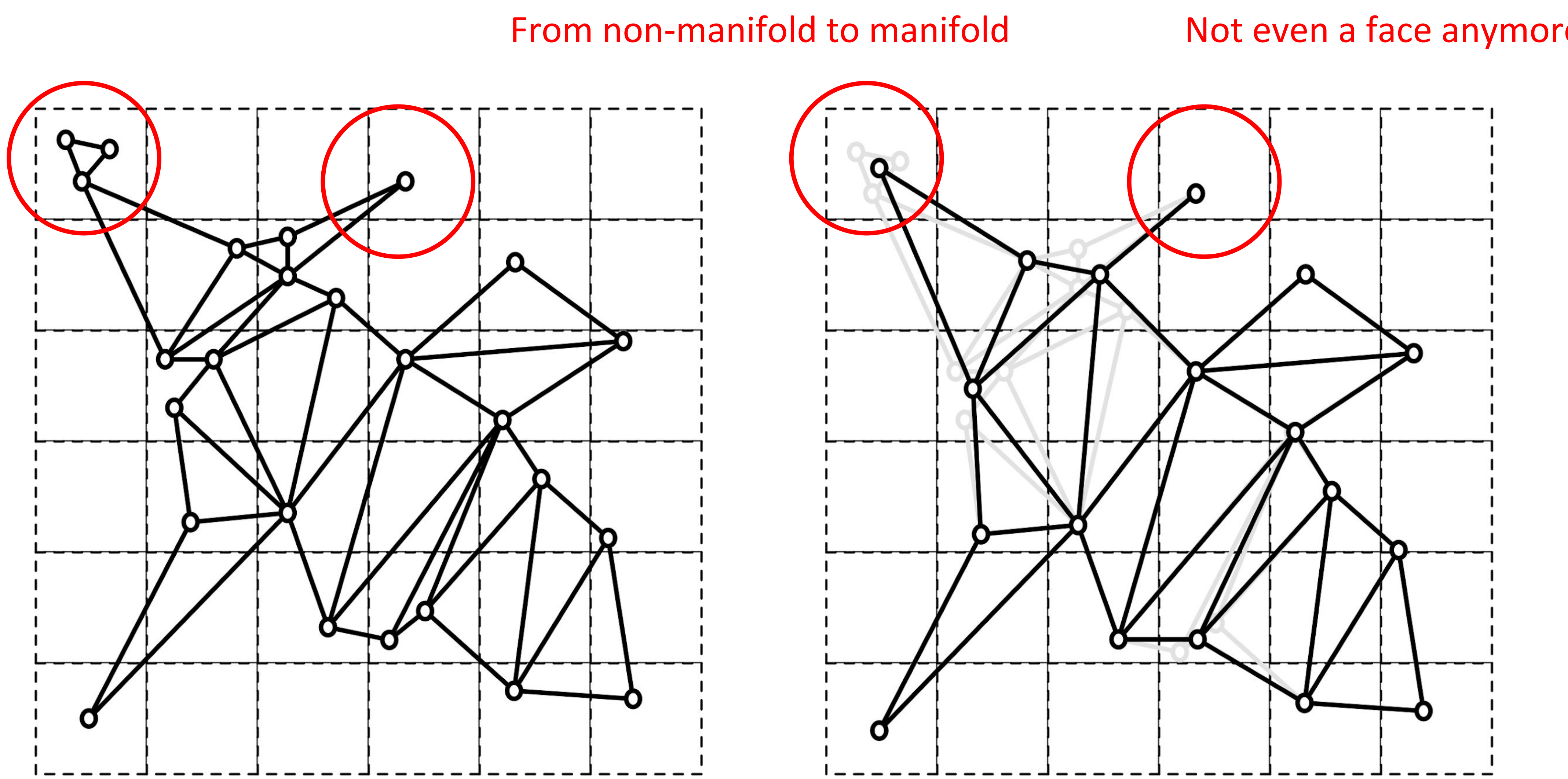


closed fan open fan

manifolds non-manifolds

Topology Change?

- Manifold ⟶ Non-manifold
- Non-manifold ⟶ Manifold
- ...



Non-manifold often causes problematic editing and rendering

Q: Can you name an example that vertex clustering change manifold to non-manifold?

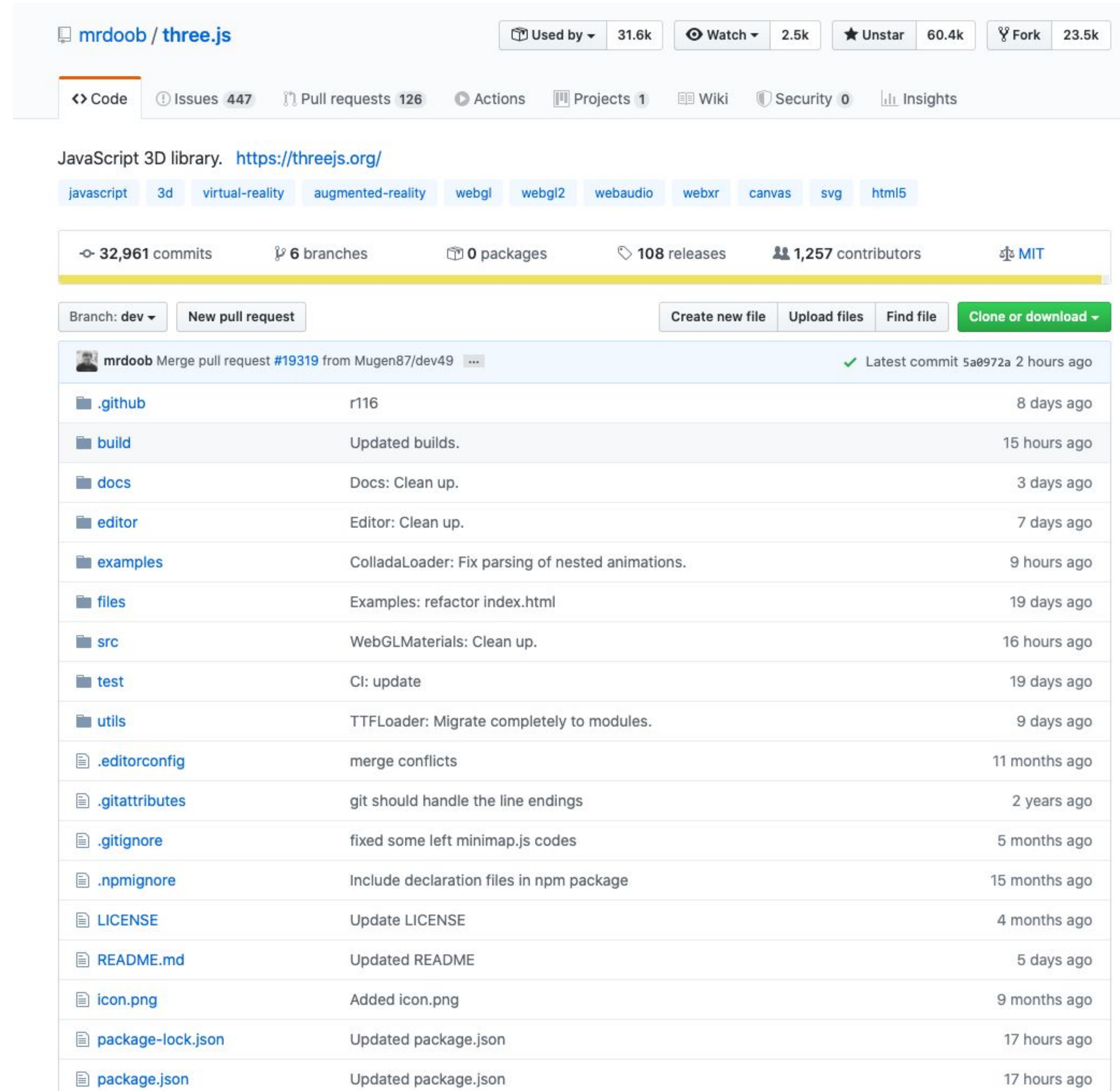
Task 3 d) Ways into source code

Most of the modern developments rely on a huge number of dependencies, these dependencies are written by others. All you can do is to **trust(?)** their implementation.

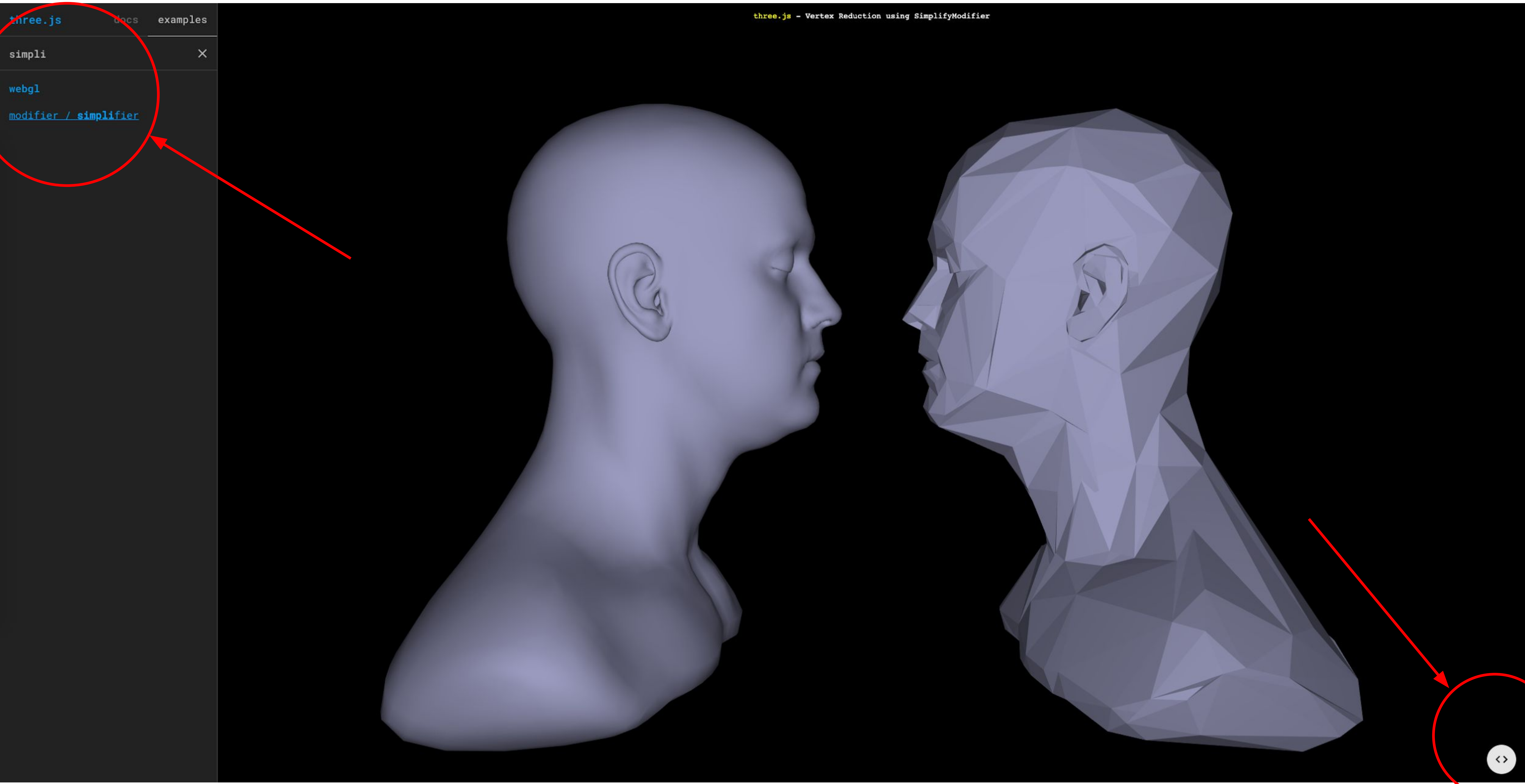
Most of the time, you don't have to worry about the things that you have used. But if a problem occurs, you will need to ask for help. *In the worst case, nobody can help you (e.g. lack of response, abandoned by maintainer, etc.) then you will have to read the source code on your own and understand what's under the hood.*

Task 3 d) Ways into source code

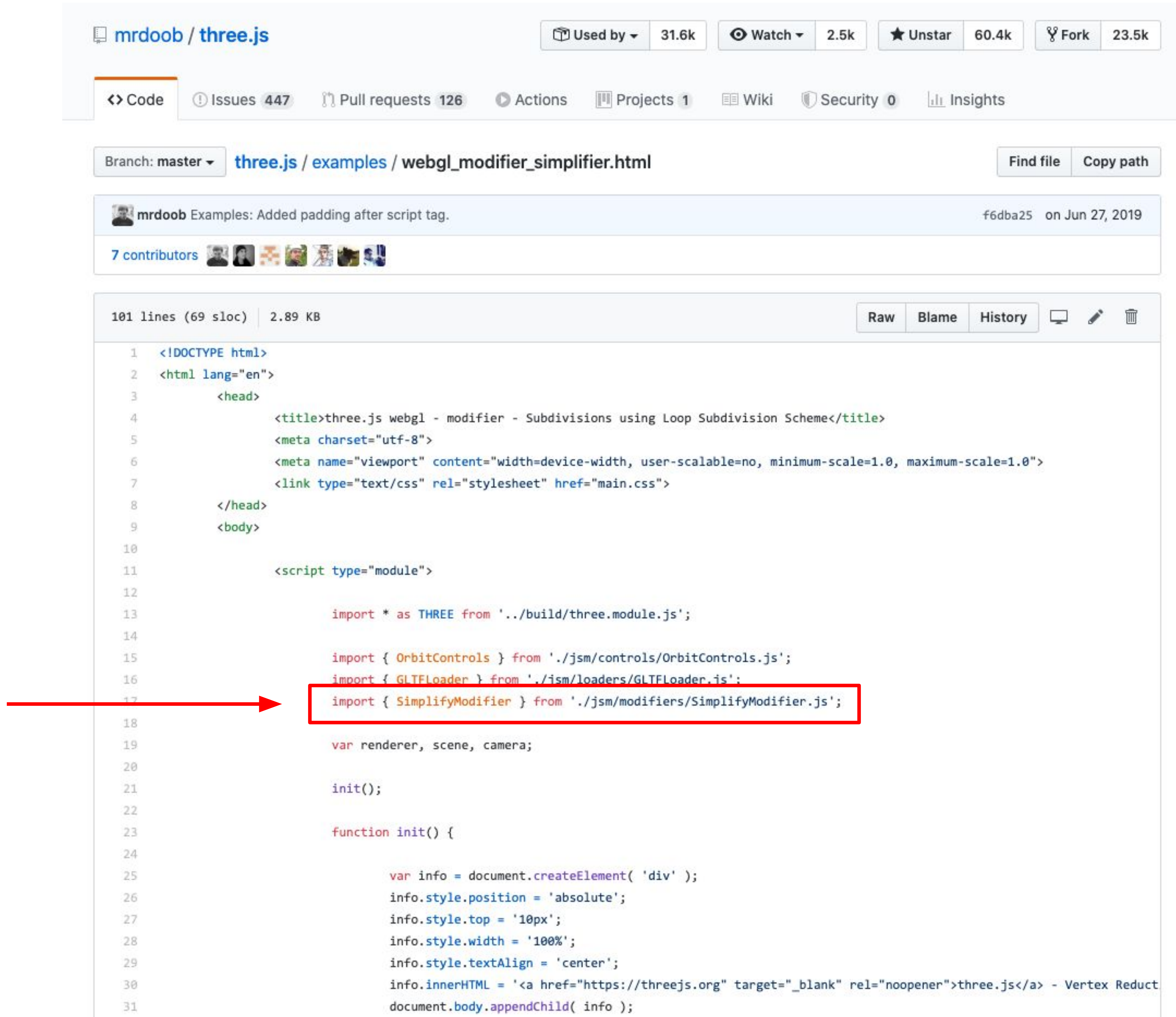
- With open source, you have the freedom to explore everything you need to understand
- Where can I find the `SimplifyModifier` and `SubdivisionModifier`?



Task 3 d) Looking for examples



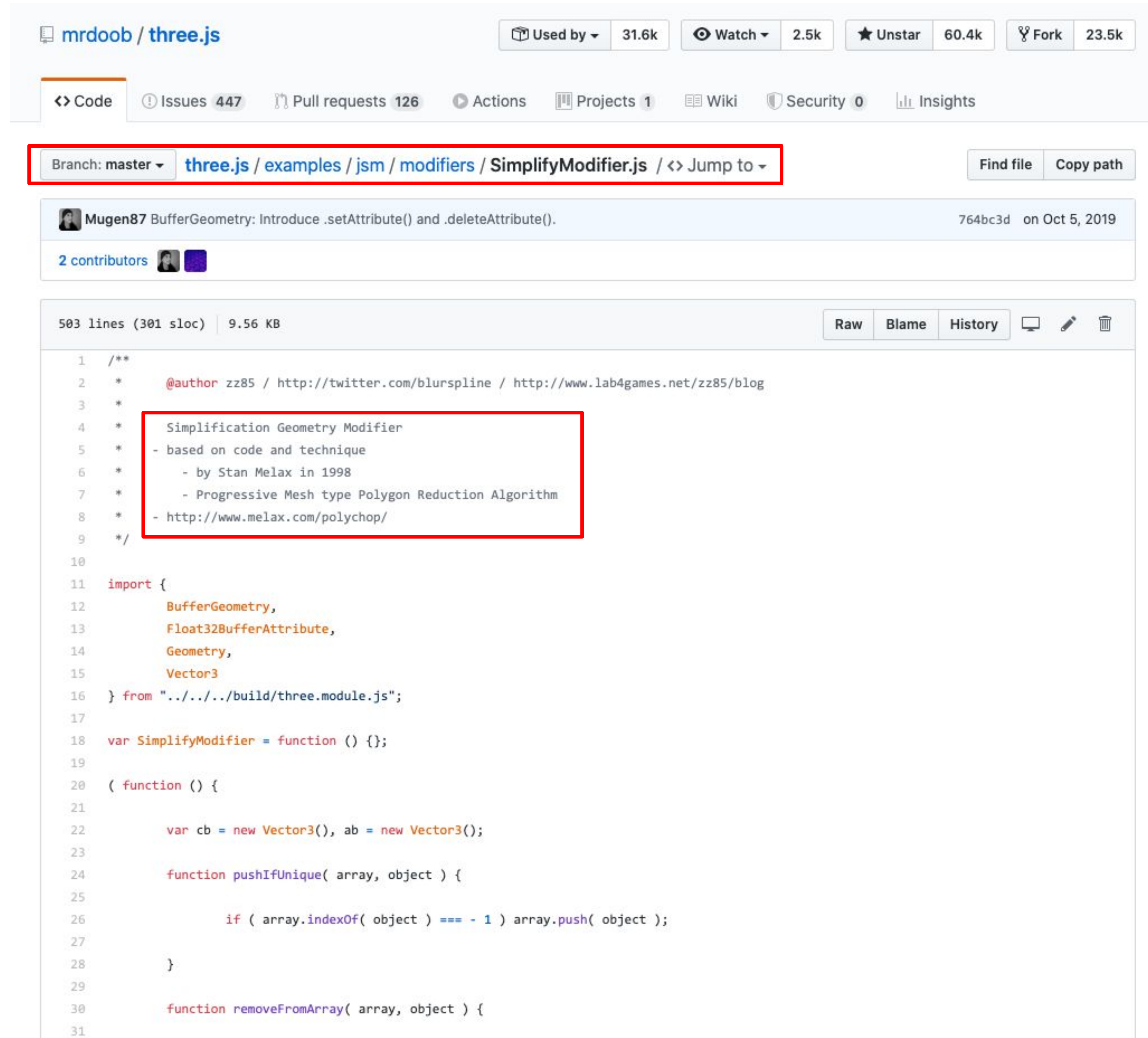
Task 3 d) Find where the dependency is introduced



Task 3 d) Read source code

Thankfully, the code is well documented.

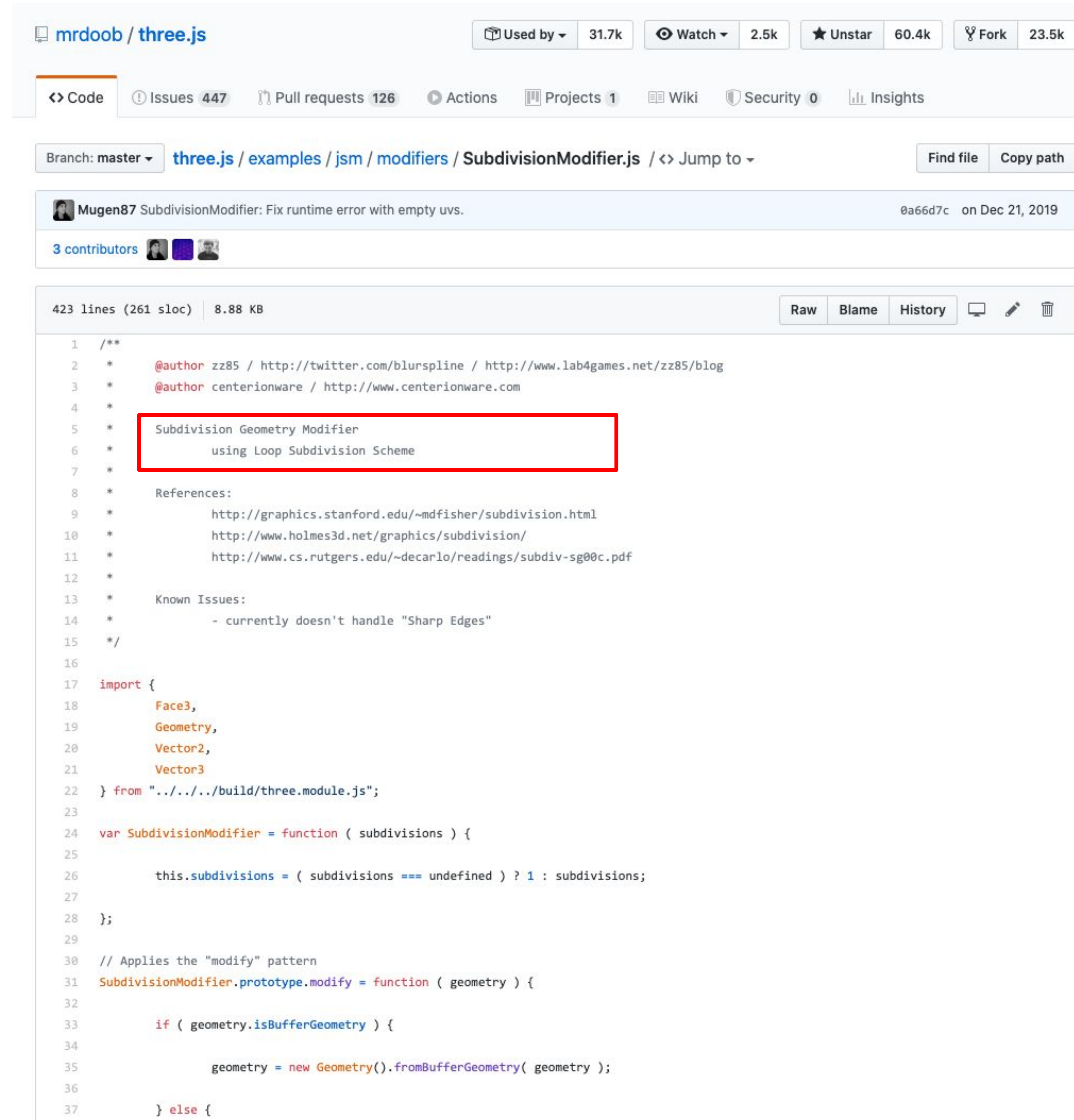
`SimplifyModifier` uses ***Progressive Polygon Reduction*** *by* ***Stan Melax***



Task 3 d) Read source code

Same way, `SubdivisionModifier` uses

Loop Subdivision



Mesh Simplification & Subdivision in *three.js*

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

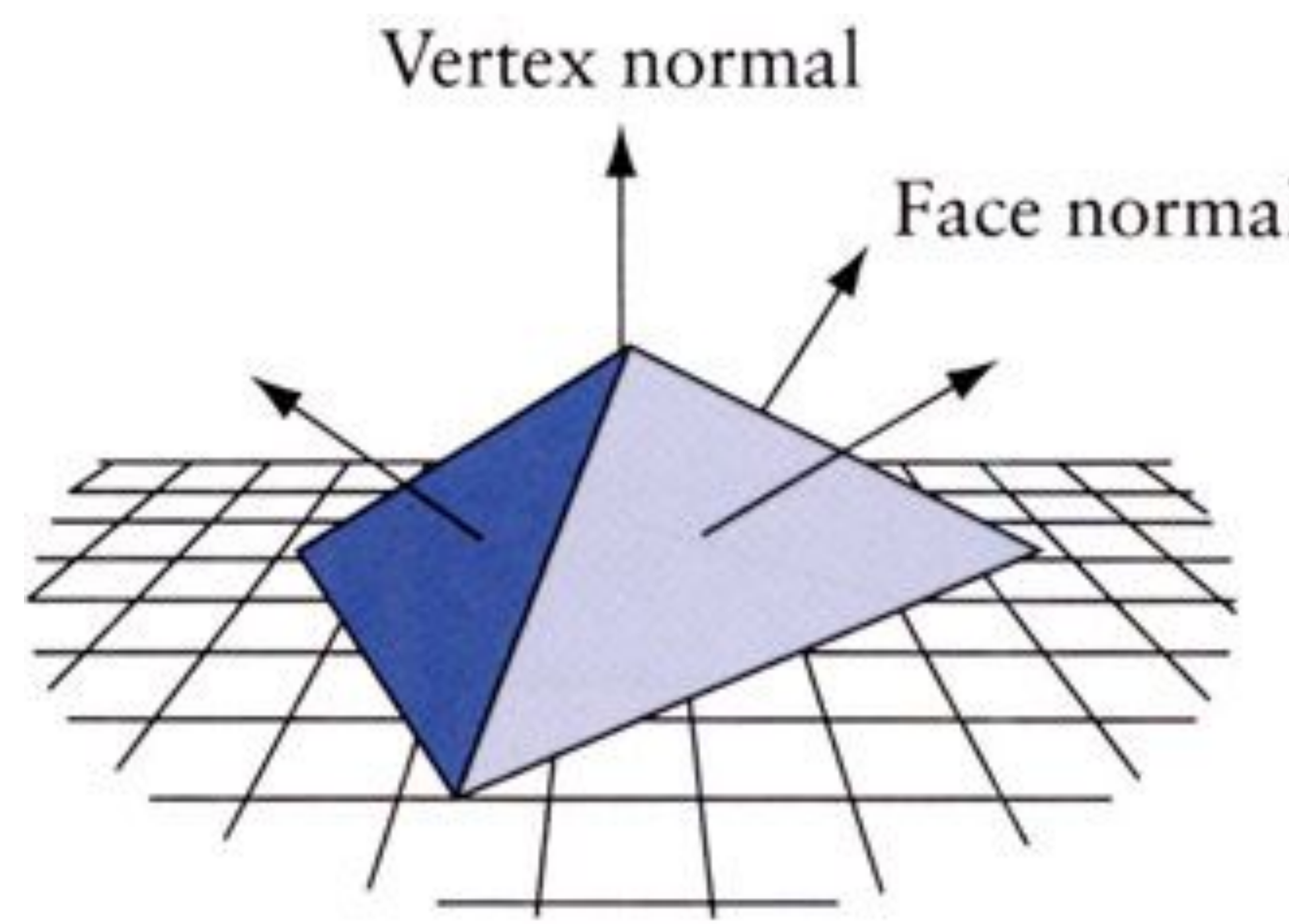
Loop, C.T., 1987. ***Smooth subdivision surfaces based on triangles***, Master's thesis Department of Mathematics. University of Utah.

Face Normal & Vertex Normal

Face normal: unit length and orthogonal with given face

Vertex normal: interpolation vector from surrounding face normals

(computation depends on the definition)

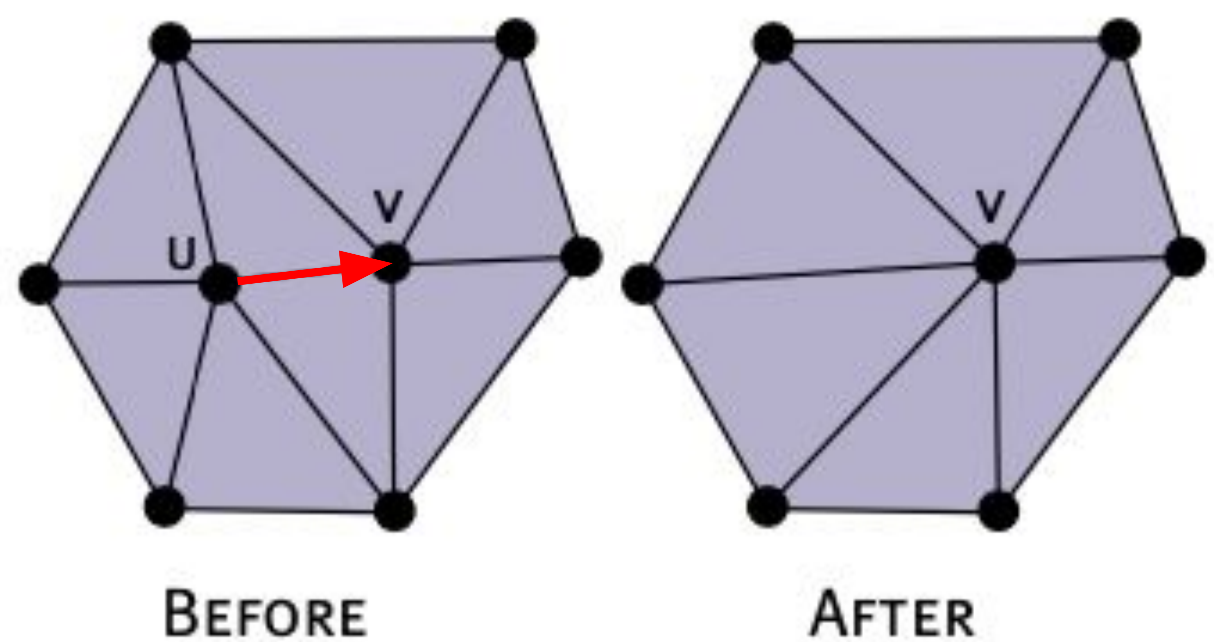


Why? Influence shading (*later lectures for more details*)

`flatShading` uses face normals, smooth shading uses vertex normals

Edge Collapse

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



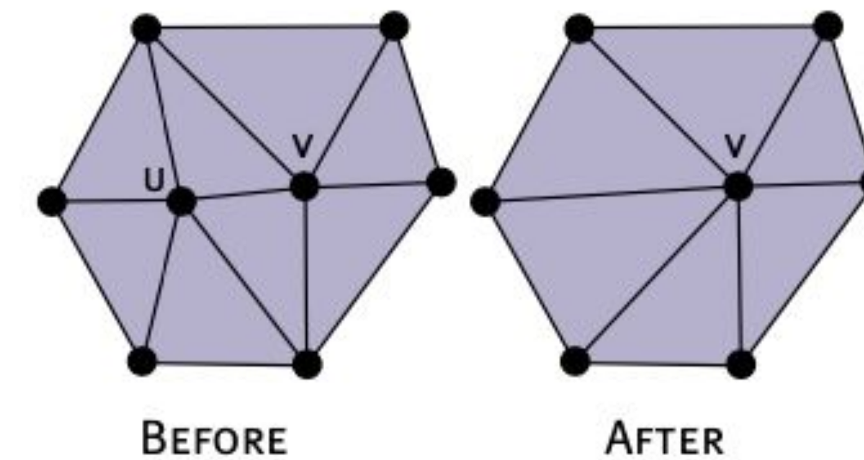
Q: How many vertices, faces and edges are removed in each *edge collapse*?

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

Pick an Edge

How much does it cost to collapse an edge?

A possible way: cost = edge length * curvature



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

where ***Tu*** is the set of triangles that contains ***u***, ***Tuv*** is the set of triangles that contains both ***u*** and ***v***.

curvature by definition: 1 - f.normal.dot(n.normal)

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

Pseudocode

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

```
const u = Vector3(...)
const v = Vector3(...)
const Tu = [...]  // faces contains u
const Tuv = [...] // faces contains u and v

let maxCurvature = 0
for (let i = 0; i < Tu.length; i++) {
   let minCurvature = 1
   for (let j = 0; j < Tuv.length; j++) {
       const curvature = 1 - Tu[i].normal.dot(Tuv[j].normal)
       if (curvature < minCurvature) {
           minCurvature = curvature
       }
   }
   if (minCurvature > maxCurvature) {
       maxCurvature = minCurvature
   }
}

const cost = u.sub(v).norm() * maxCurvature
```

Melax's Progressive Polygon Reduction - Optimization

We know the cost of collapse an edge.

But if we collapse an edge, costs of neighbors can also be affected (why?)

How to **efficiently** simplify a mesh progressively?

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

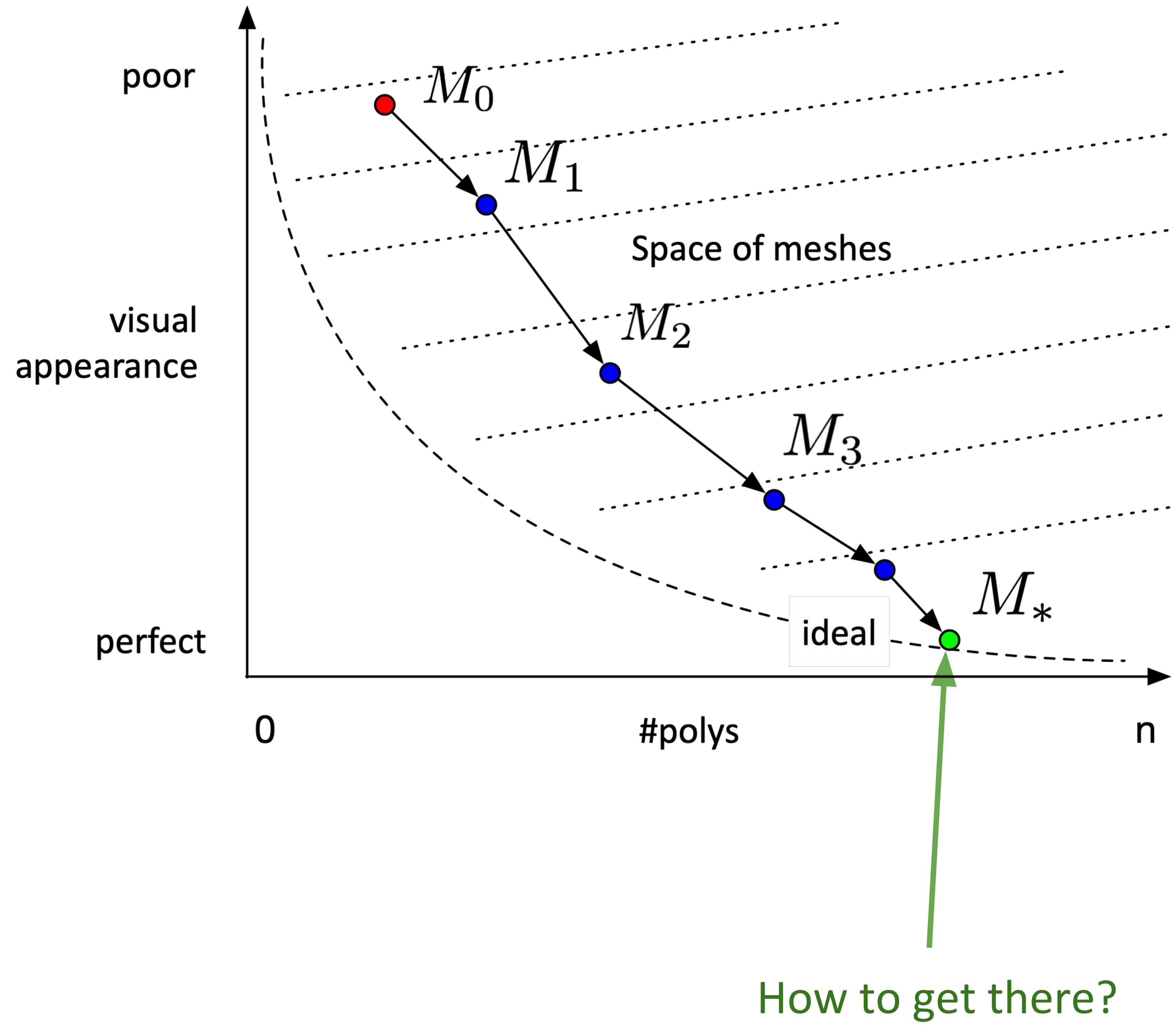
- cost of access min element: ***O(1)***
- cost of affected elements manipulation: ***O(log(n))***

Mesh Subdivision (Upsample)

Increase #polygons that smoothly approximate *its shape*

Triangle: Loop

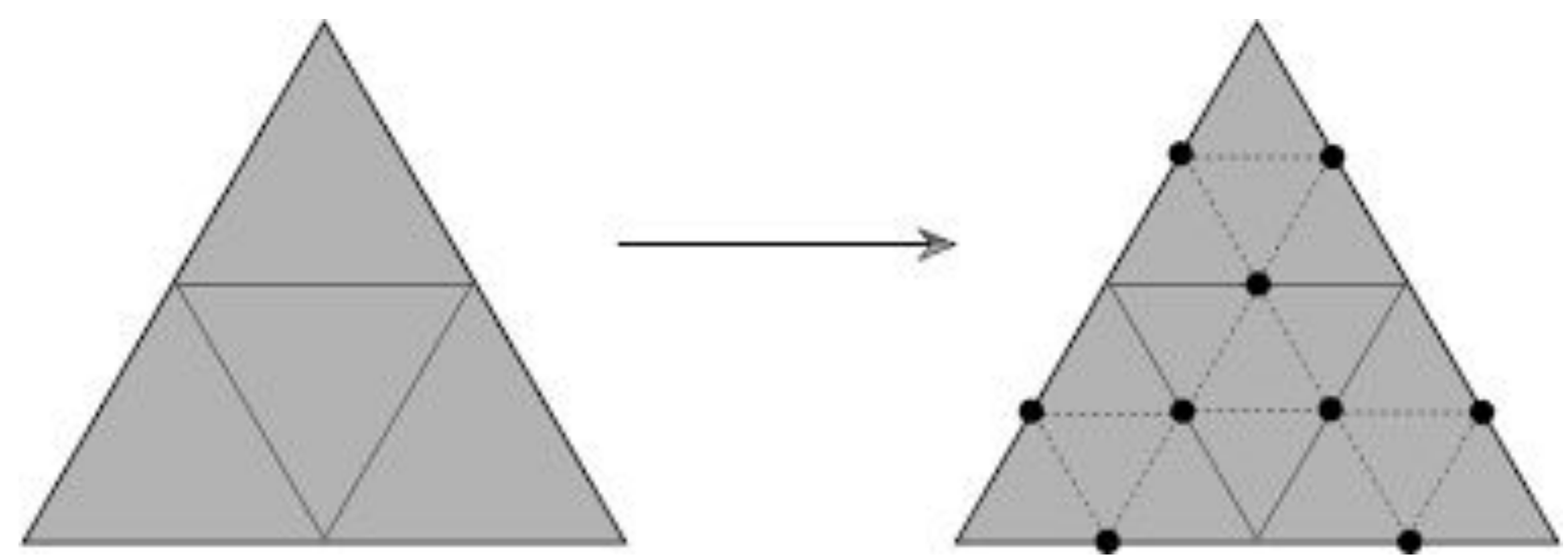
Quad: Catmull-Clark



Mesh Subdivision: Loop Subdivision

Basic idea: interpolating at every midpoint

#poly *= 4^(subdivision number)



What if...

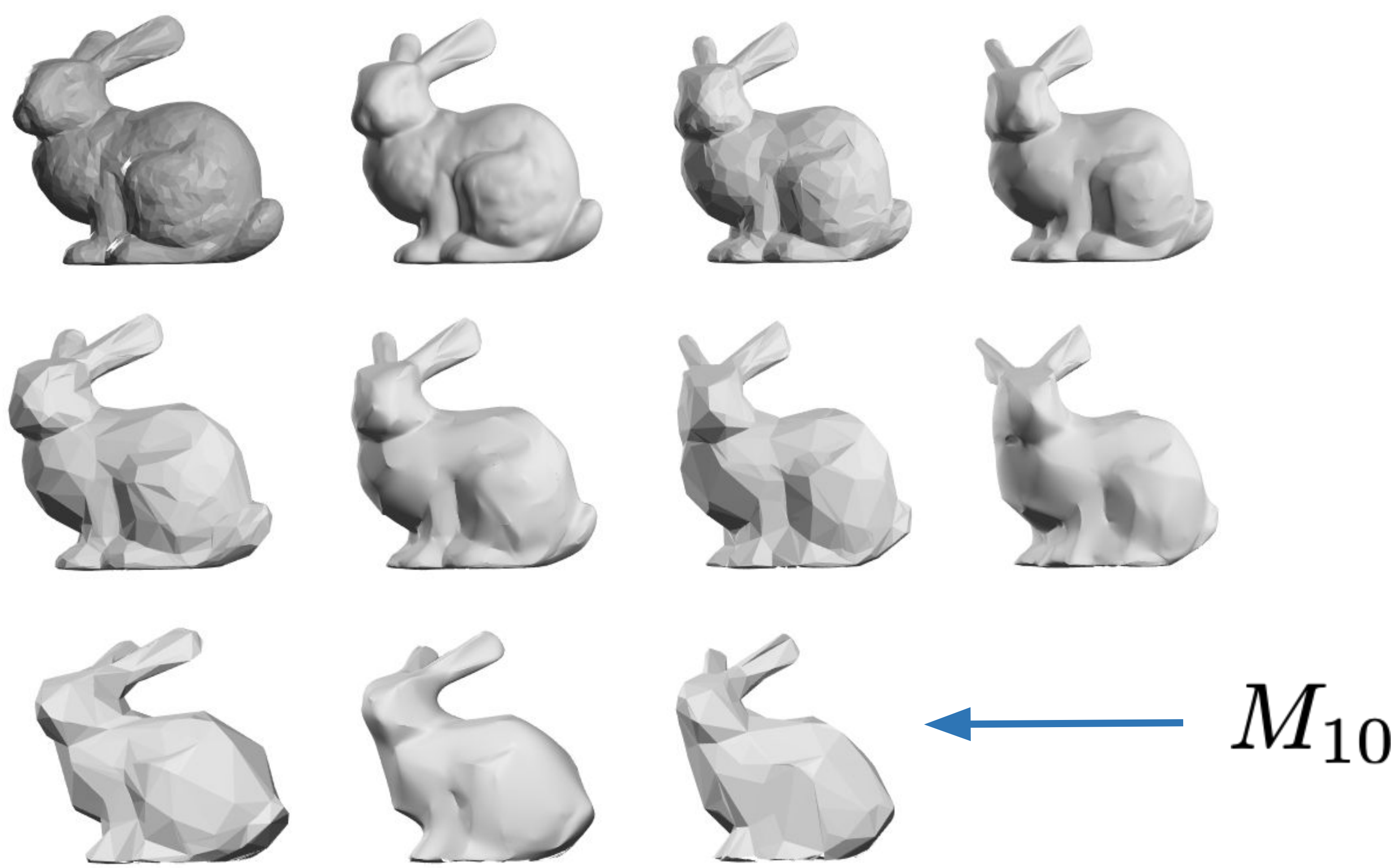
$$M_0 \xrightarrow{\text{subdivision}} M_1 \xrightarrow{\text{simplification}} M_2 \xrightarrow{\text{subdivision}} M_3 \xrightarrow{\text{simplification}} M_4 \longrightarrow M_n$$

Task 3 e)

```
export default class Bunny extends Renderer {
 constructor() {
   super()
   this.scene.add(new AmbientLight(0x333333))
   const light = new PointLight(0xffffff, 0.8, 1000);
   light.position.copy(new Vector3(100, 50, 100))
   this.scene.add(light)

   const loader = new GLTFLoader()
   loader.load('assets/bunny.glb', model => {
     const simplifier = new SimplifyModifier()
     const subdivision = new SubdivisionModifier(2)
     const reduceRatio = 0.95
     const N = 10

     // TODO: Implement repetitive subdivision and simplification.
     const addBunny = (g, i) => {
       const bunny = new Mesh(g, new MeshStandardMaterial())
       bunny.rotateX(Math.PI/2)
       bunny.scale.copy(new Vector3(40, 40, 40))
       bunny.translateX(8*i)
       this.scene.add(bunny)
     }
     // original model
     const original = model.scene.children[0]
     original.scale.copy(new Vector3(40, 40, 40))
     this.scene.add(original.clone())

     let g = new Geometry().fromBufferGeometry(model.scene.children[0].geometry)
     g.mergeVertices()
     for (let i = 1; i <= N; i += 2 ) {
       g = subdivision.modify(g)
       addBunny(g, i)
       g = simplifier.modify(g, Math.floor(g.vertices.length*reduceRatio))
       g = (new Geometry()).fromBufferGeometry(g)
       addBunny(g, i+1)
     }
   })
 }
}
```

Task 3 e)

If subdivision number = 2, reduction ratio of number of vertices = **95%**:

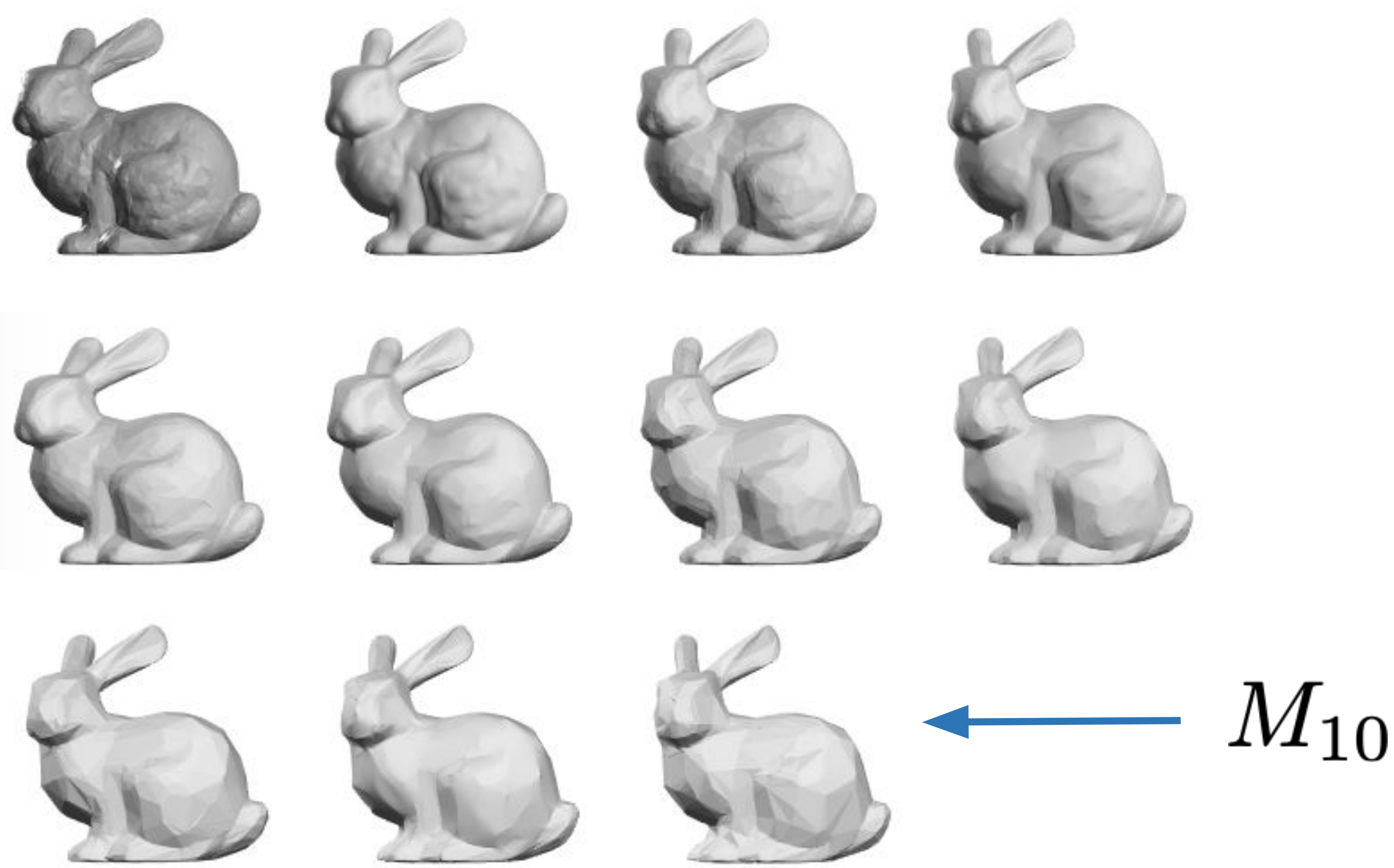


Iteration	Vertices	Faces
0	**2503**	**4968**
1 (subdivision)	39826	79488
2 (simplification)	1990	3816
3 (subdivision)	30853	61056
4 (simplification)	1387	2592
5 (subdivision)	21061	41472
6 (simplification)	789	1438
7 (subdivision)	11763	23008
8 (simplification)	537	978
9 (subdivision)	7962	15648
10 (simplification)	**370**	**616**

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

Task 3 e)

If subdivision number = 2, reduction ratio of number of vertices = **90%**:



Iteration	Vertices	Faces
0	**2503**	**4968**
1 (subdivision)	39826	79488
2 (simplification)	3981	7798
3 (subdivision)	62715	124768
4 (simplification)	6075	11545
5 (subdivision)	93357	184720
6 (simplification)	3561	6710
7 (subdivision)	54051	107360
8 (simplification)	2658	5009
9 (subdivision)	40176	80144
10 (simplification)	**2666**	**5002**

Neither number of vertices nor faces were below the original model;

Observation: Shape is still not exactly preserved.

More about mesh sampling

Other possibilities:

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

We encourage you to explore and verify by yourself :)

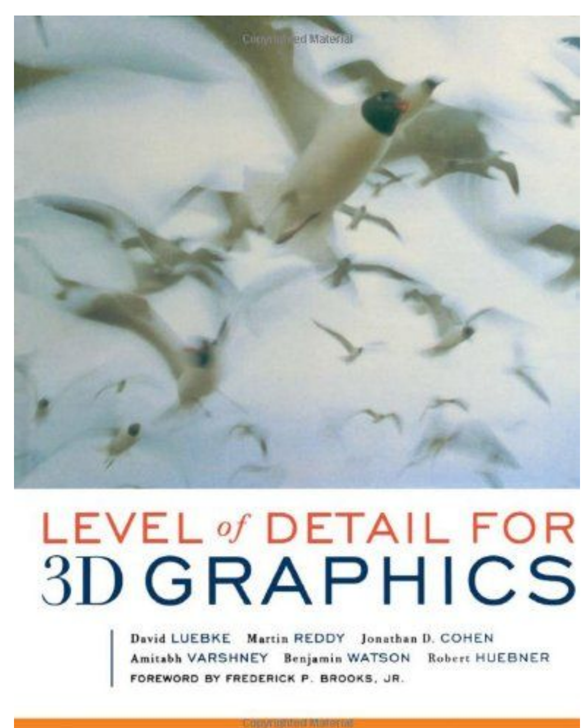
Task 3 f) Mesh *Aliasing*

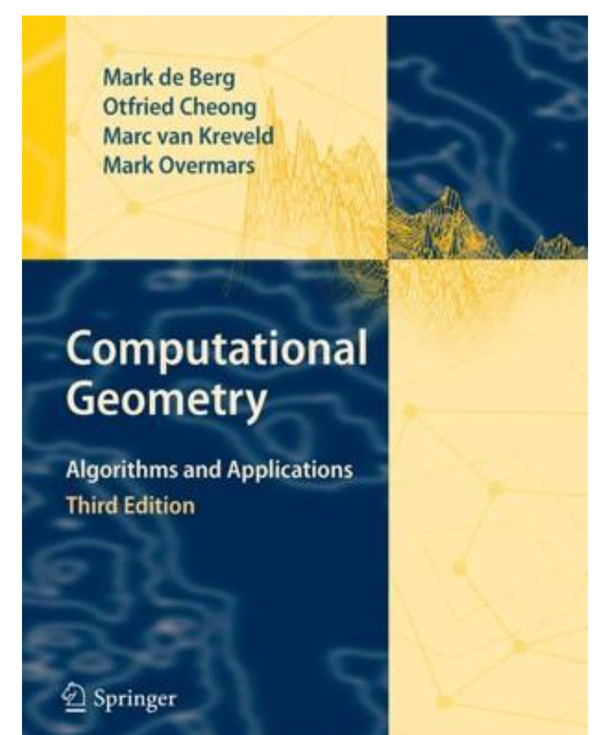
- The method for upsampling or downsampling is not an inverse to one another

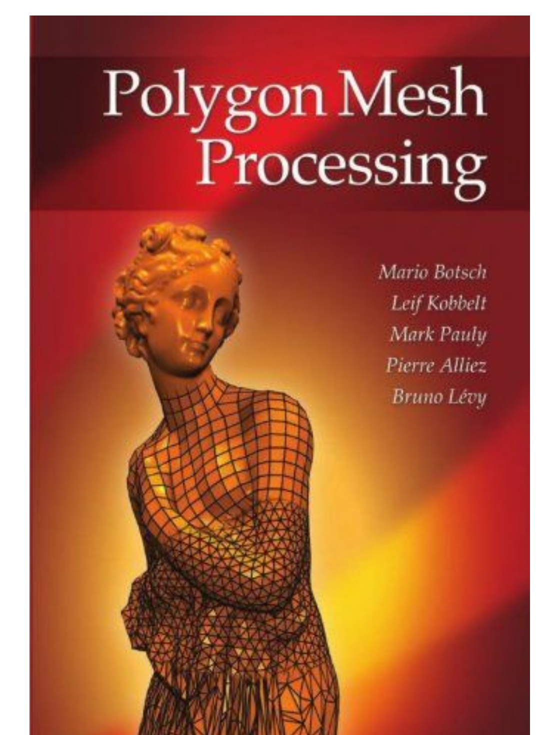
⇒ Aliasing errors can occur if the sampling pattern is not perfectly aligned to features in the original geometry

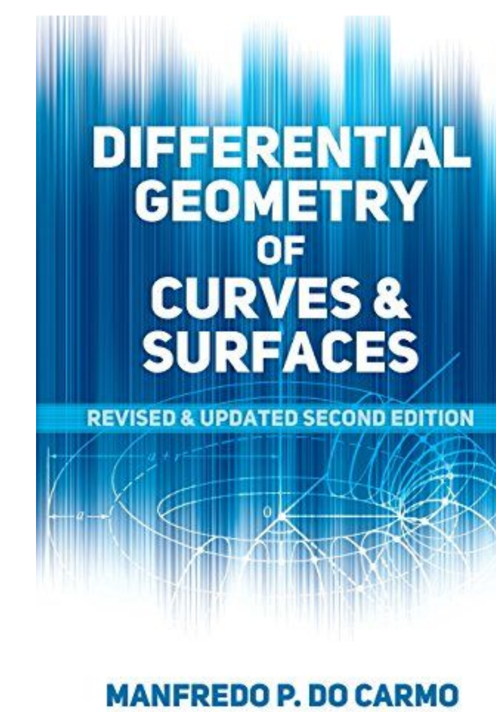
Take Away

- A lot of open problems in geometry remains unsolved, and they are utterly hard
- If you are interested in practical 3D modeling, now you have enough basic knowledge. Check out the **Blender** (an amazing free and open source software), find a tutorial that fits your taste then get started.
- If you are more interested in technical geometric analysis, check out these fascinating books, and enjoy :)









Thanks!
What are your questions?

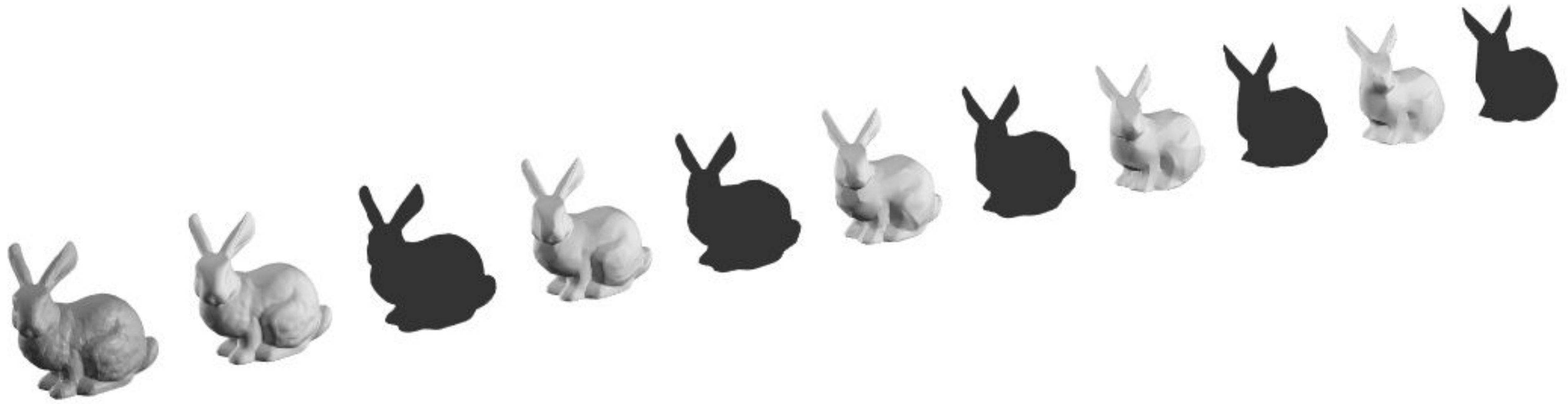
Appendix

If you met this issue… 😄

`SimplifyModifier` does not compute vertex normals, this means your simplified model will not be shaded unless you use flat shading. Two possible fixes:

1. manually compute vertex normals:

```
const addBunny = (g, i) => {
        g.computeVertexNormals()
        const bunny = new Mesh(g,
          new MeshStandardMaterial(),
        )
        bunny.rotateX(Math.PI/2)
        bunny.scale.copy(new Vector3(40, 40, 40))
        bunny.translateX(8*i)
        this.scene.add(bunny)
}
```



2. Or create a Geometry from a BufferGeometry (used in the provided solution):

```
for (let i = 1; i <= N; i += 2 ) {
        g = subdivision.modify(g)
        addBunny(g, i)
        g = simplifier.modify(g, Math.floor(g.vertices.length*reduceRatio))
        g = new Geometry().fromBufferGeometry(g)
        addBunny(g, i+1)
}
```

Midterm Survey

Submit your feedback before 08.06.2020, the results will be available to you later when the evaluation is done.
Link: https://forms.gle/XqWC5cctM56GBvZV9