Praktikum

Geometry Processing

1 Introduction

Ludwig-Maximilians-Universität München

Factual Error

Bad English

Lack of Explanation

Code Syntax Error

Tons of Typos

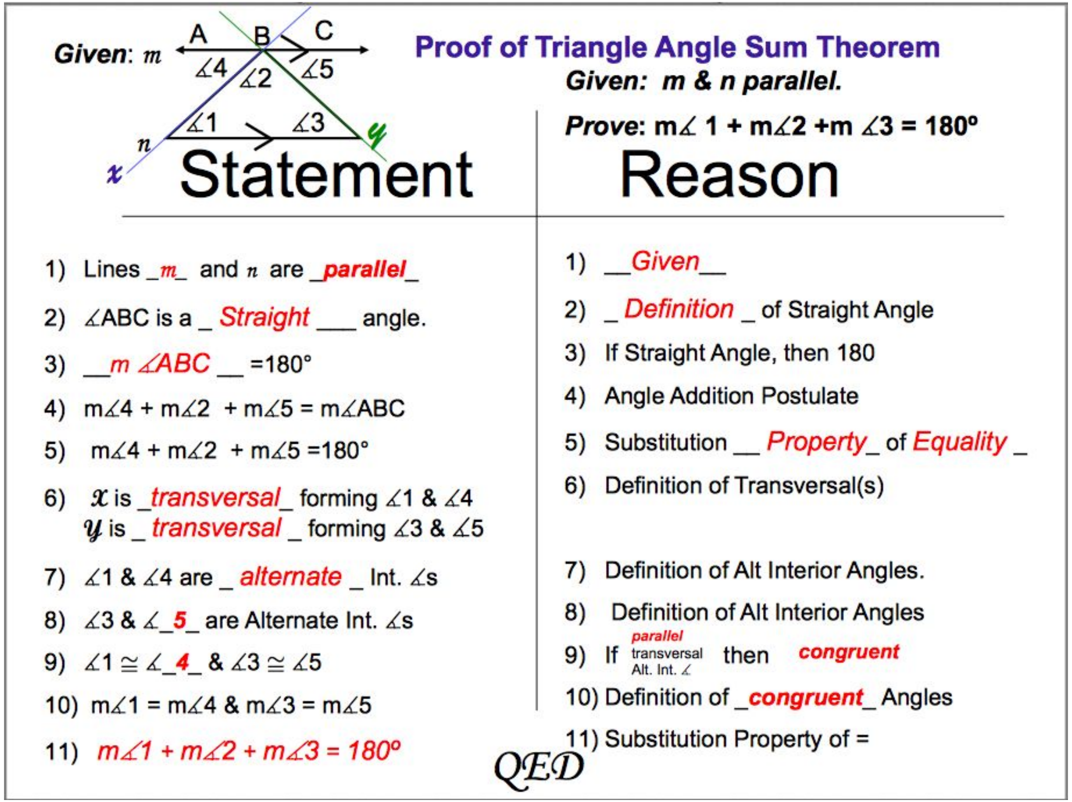
Incomprehensible

Wrong Information

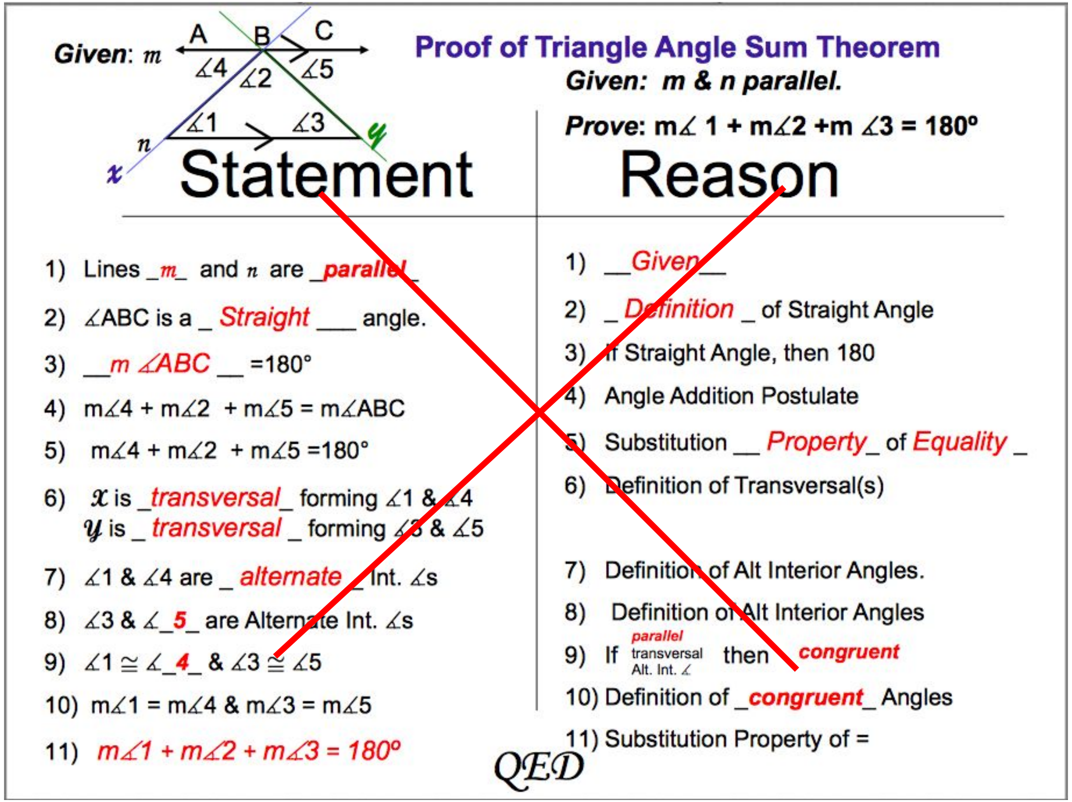
Session 1: Introduction

- Motivation
- Recap: Graphics Pipelines
- Geometry Representations
- Blender Basics
- Summary & Homework

This course is *not* about ...

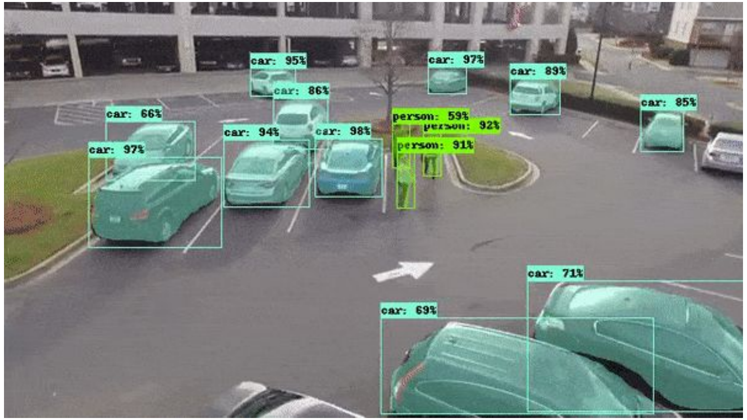


This course is *not* about ...

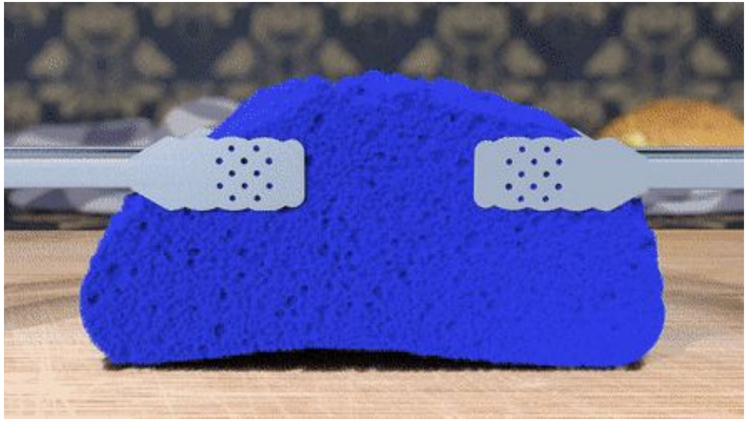


This course is *not* about ...

[Yan et al. 2015]

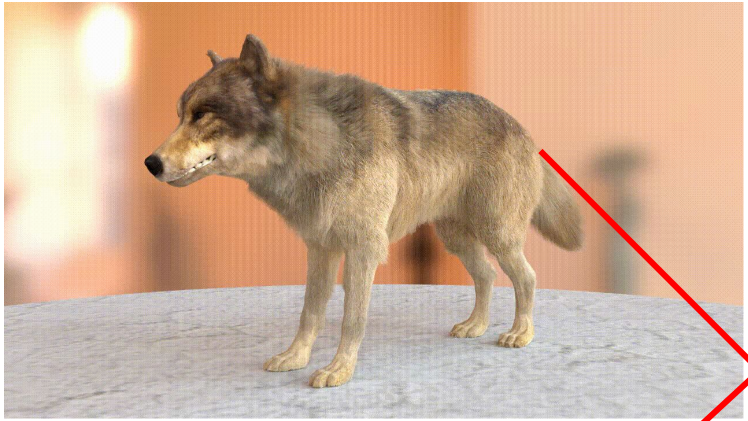


[He et al. 2018]

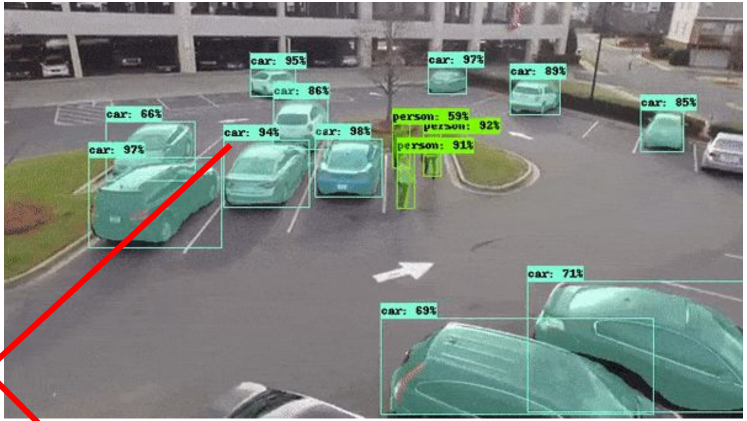


[Wolper et al. 2019]

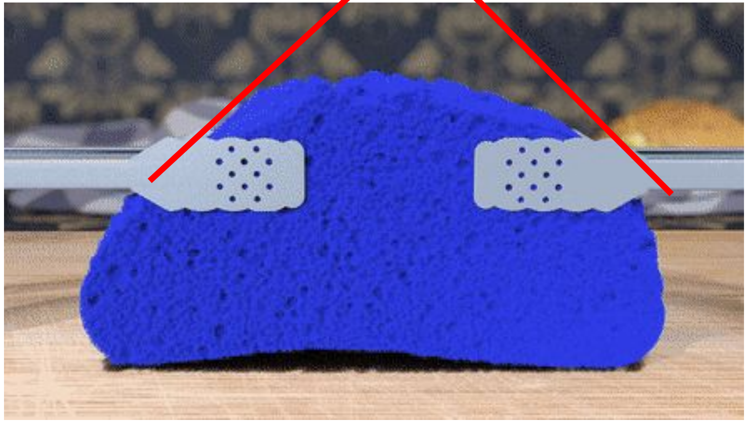
This course is *not* about ...



[Yan et al. 2015]

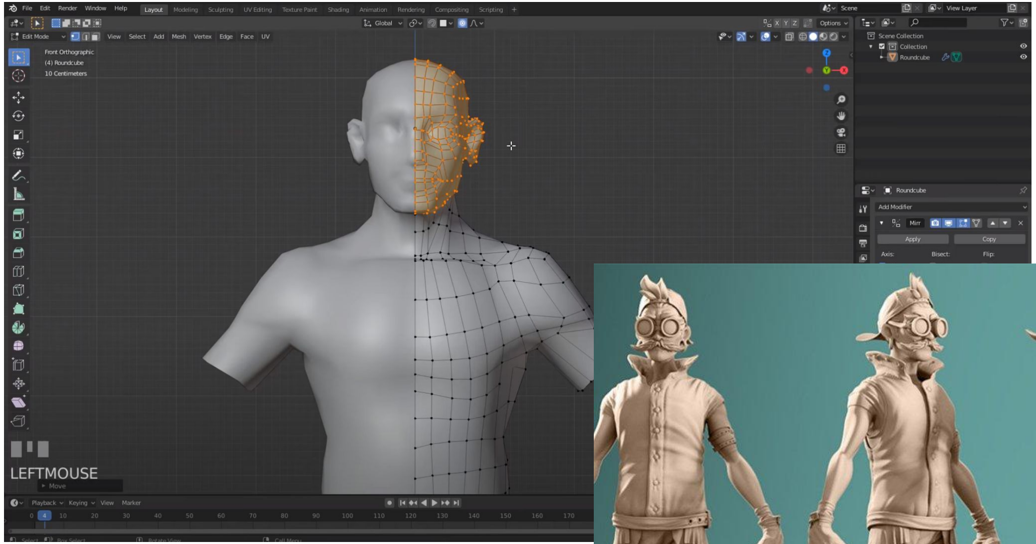


[He et al. 2018]

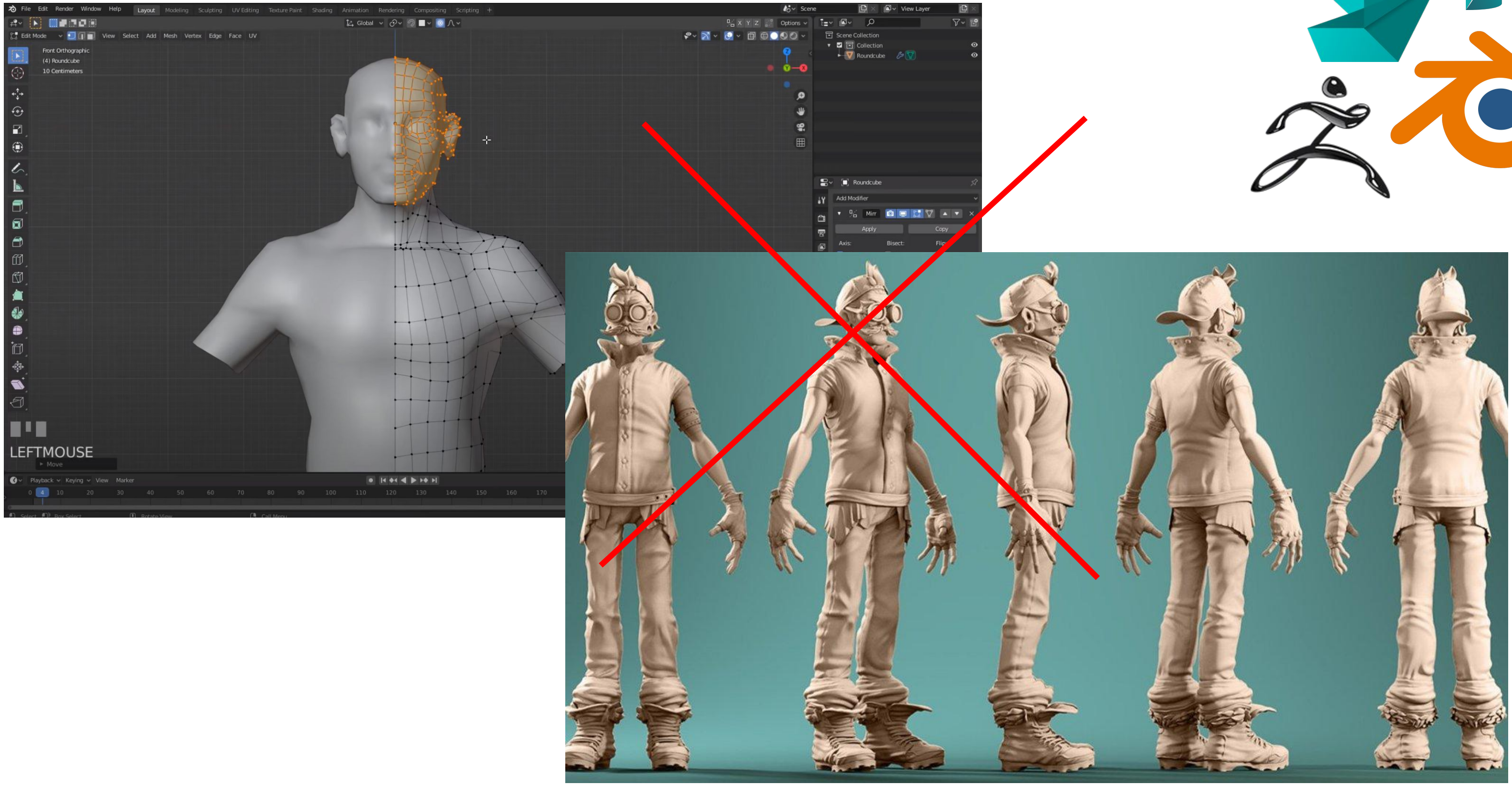


[Wolper et al. 2019]

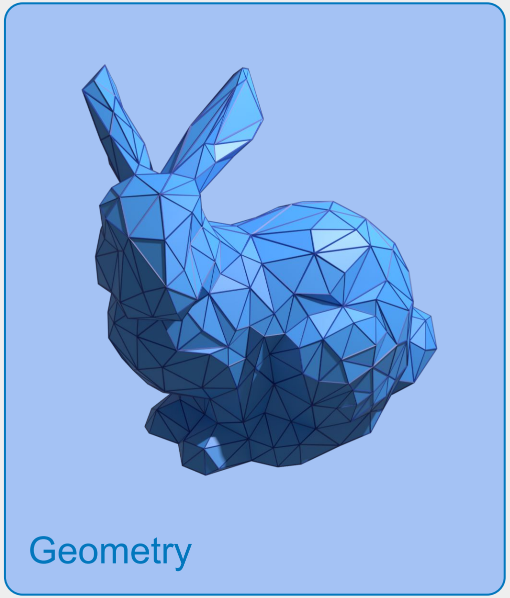
This course is also *not* about ...

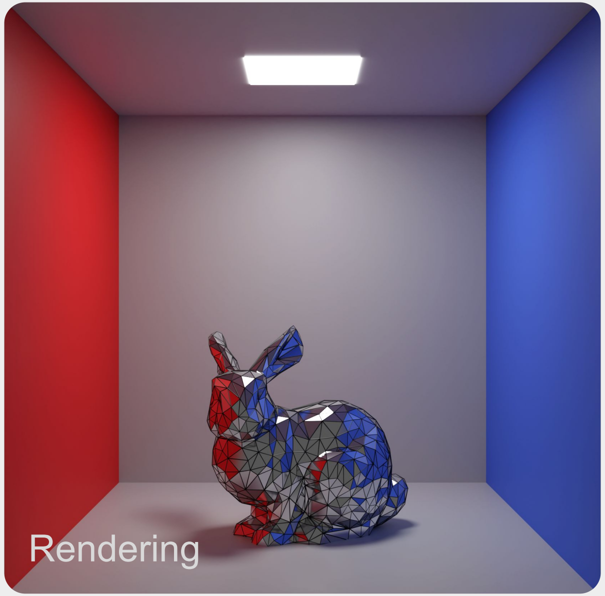


This course is also *not* about ...



This course is a direct extend to the CG1 for geometry





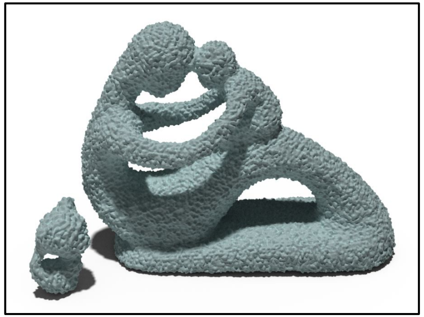
Computer Graphics

This course is a direct extend to the CG1 for geometry

We will Focus on How to Deal with 3D Geometries *Algorithmically*

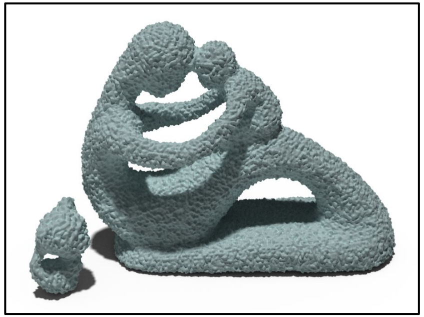
Curvature;

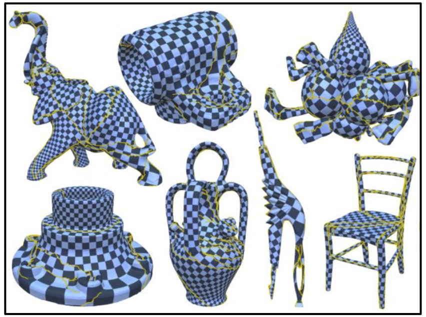
We will Focus on How to Deal with 3D Geometries *Algorithmically*



Curvature; Smoothing;

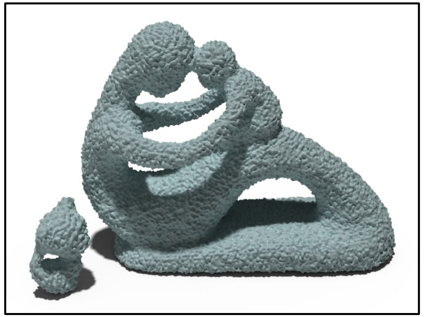
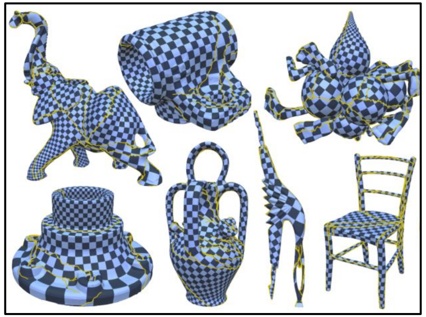
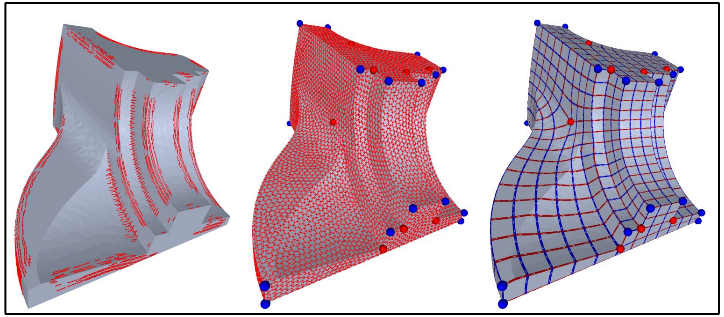
We will Focus on How to Deal with 3D Geometries *Algorithmically*





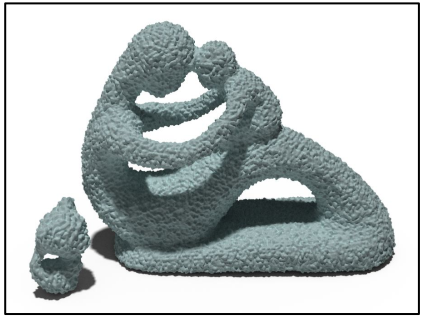
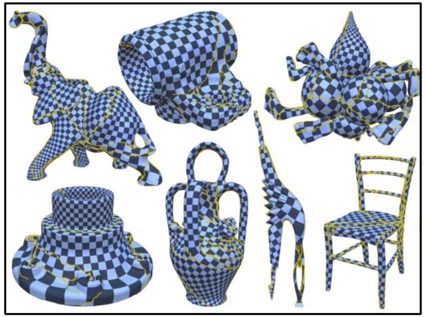
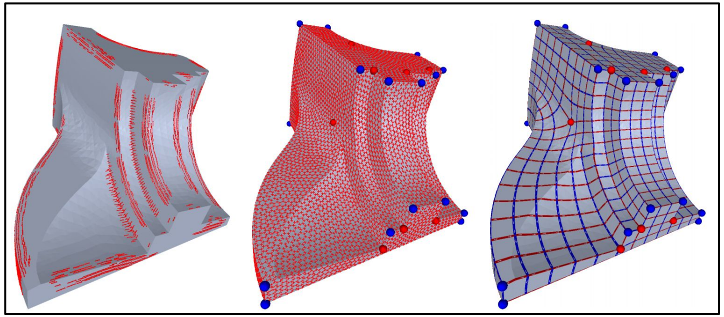
Curvature; Smoothing; Parameterization;

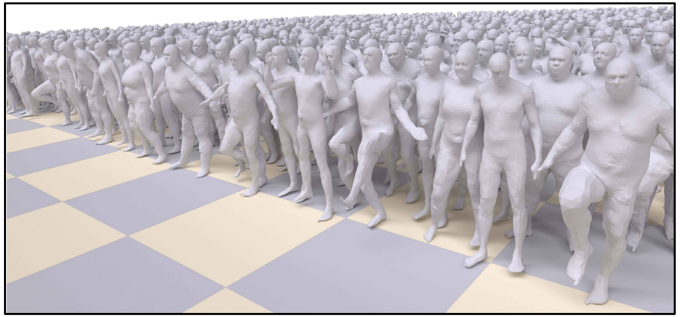
We will Focus on How to Deal with 3D Geometries *Algorithmically*


Curvature; Smoothing; Parameterization; Remeshing;

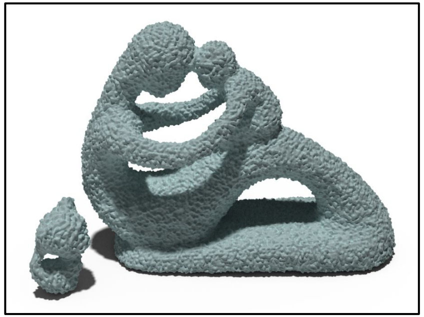
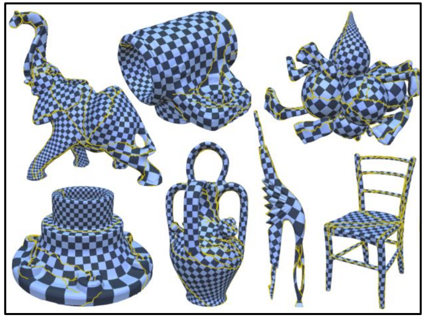
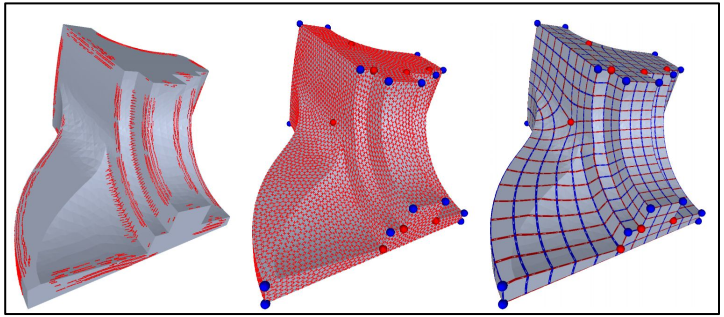
We will Focus on How to Deal with 3D Geometries *Algorithmically*

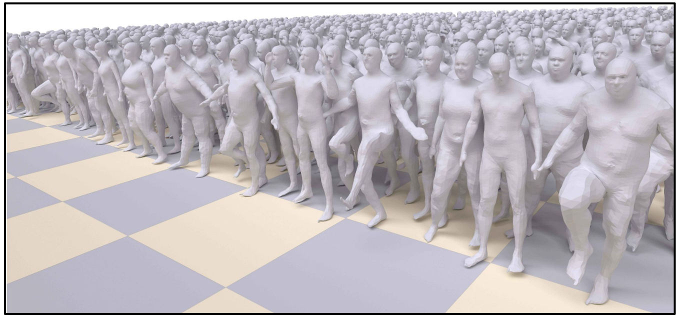
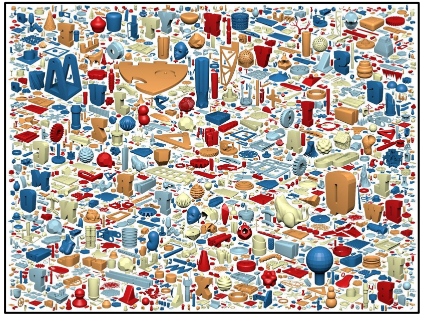







Curvature; Smoothing; Parameterization; Remeshing; Deformation;

We will Focus on How to Deal with 3D Geometries *Algorithmically*

......

Curvature; Smoothing; Parameterization; Remeshing; Deformation; Shape Analysis; ...

This Course remains Interdisciplinary ...

$$\mathcal{H}(\mathcal{M}, \mathcal{M}') = \sqrt{\frac{1}{|\mathcal{S}|} \iint_{v \in \mathcal{S}} d(p, \mathcal{S}')^2 \mathrm{d}\mathcal{S}}$$

Mathematics

Formal foundation: **Differential geometry**, numeric analysis, …
(back to 19 century)

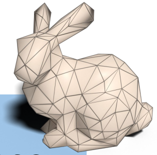
Geometry Processing

This Course remains Interdisciplinary ...

$$\mathcal{H}(\mathcal{M},\mathcal{M}') = \sqrt{\frac{1}{|\mathcal{S}|}\iint_{v\in\mathcal{S}} d(p,\mathcal{S}')^2 \mathrm{d}\mathcal{S}}$$

Mathematics

Formal foundation: Differential geometry, numeric analysis, …
(back to 19 century)

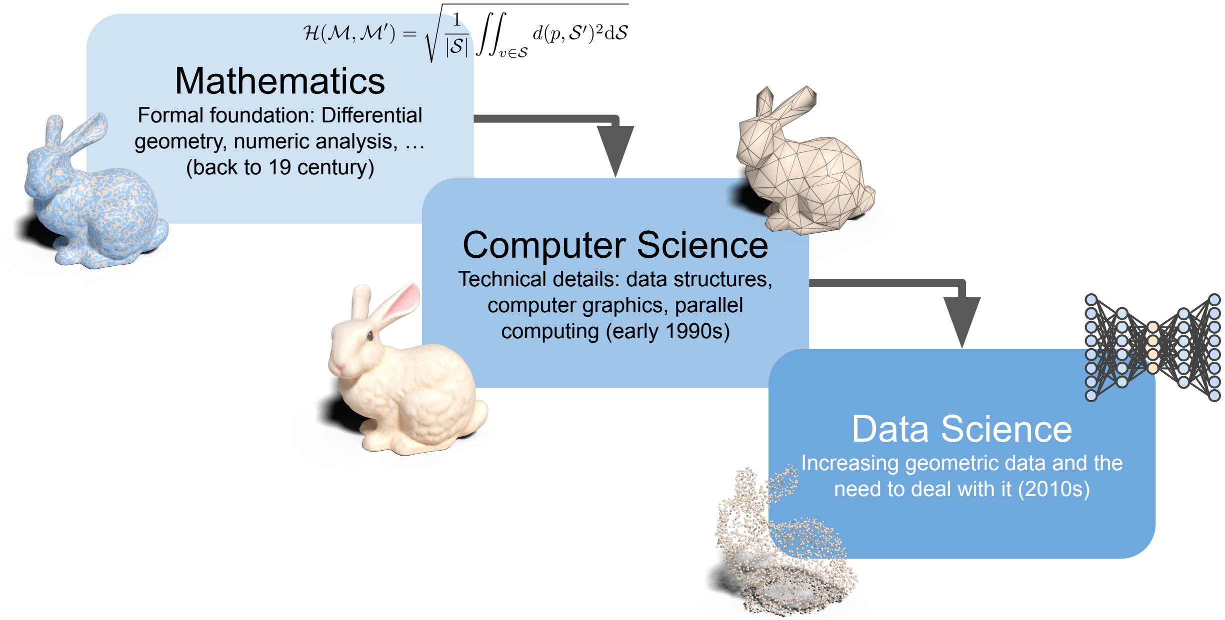


Computer Science

Technical details: data structures, computer graphics, parallel computing (early 1990s)

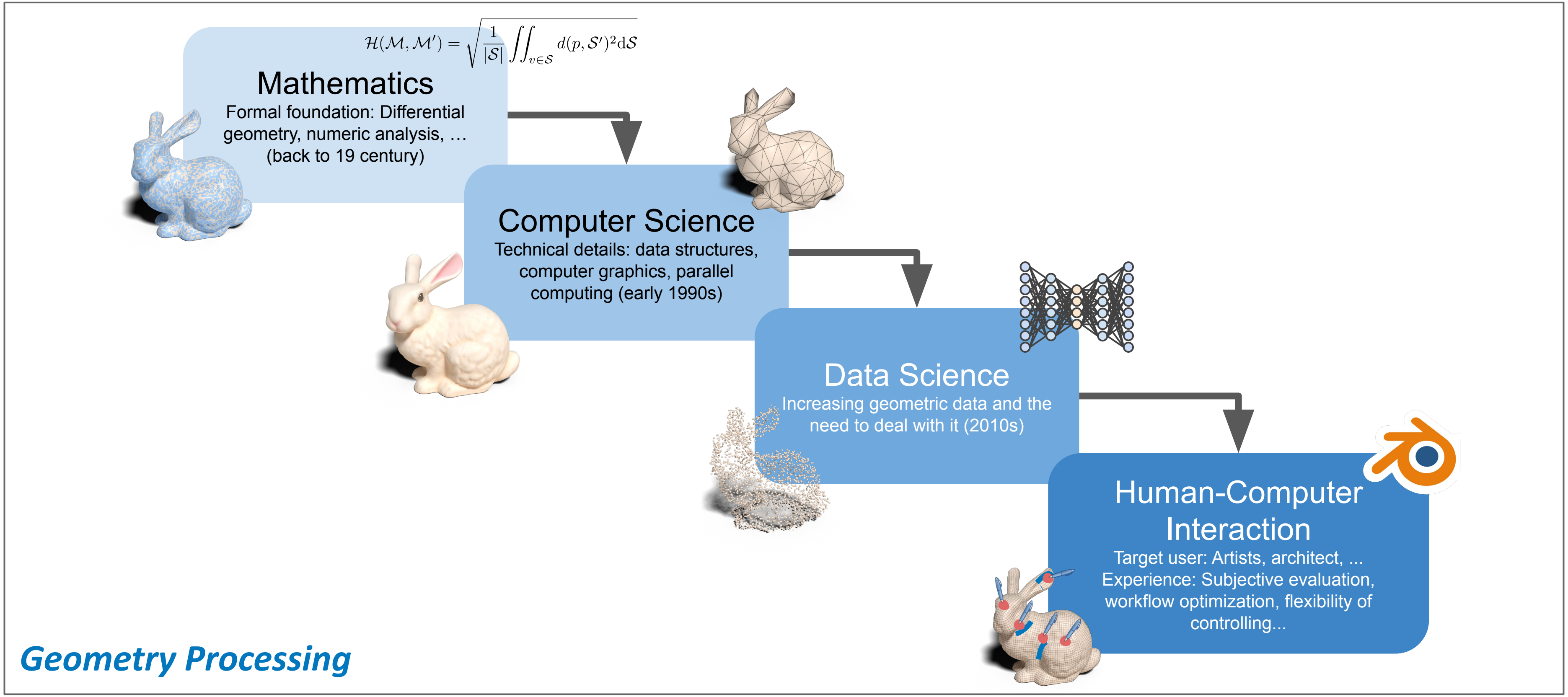
Geometry Processing

This Course remains Interdisciplinary ...



Geometry Processing

This Course remains Interdisciplinary ...

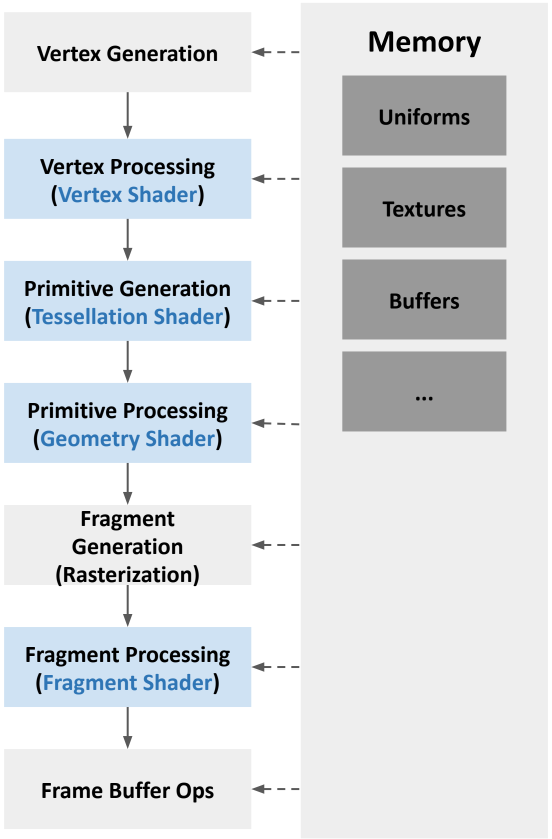


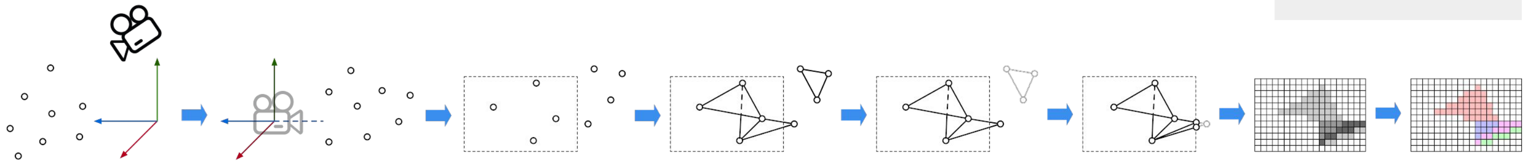
Session 1: Introduction

- Motivation
- Recap: Graphics Pipelines
- Geometry Representations
- Blender Basics
- Summary & Homework

Rasterization Pipeline

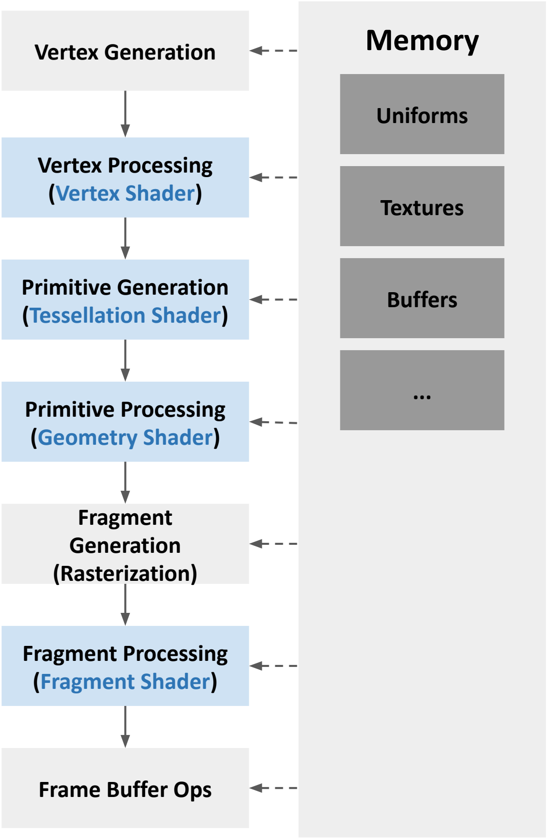
```
init frame buffer
init z buffer
for each triangle t in scene {
   tp = project(t)
   for each pixel p in frame buffer {
       if tp covers p {
           if z value at p is closer than z buffer at p {
               update z buffer and frame buffer
           }
       }
   }
}
flush frame buffer to monitor
```

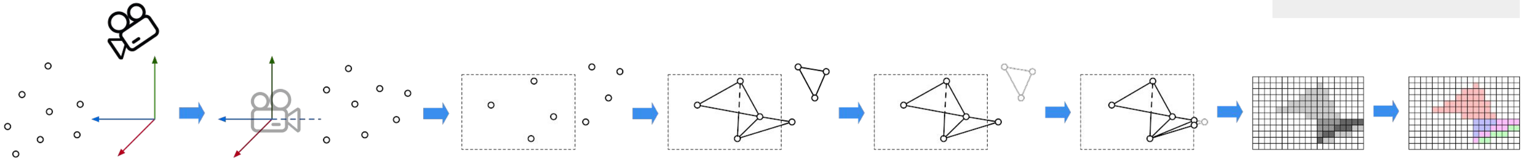




Rasterization Pipeline

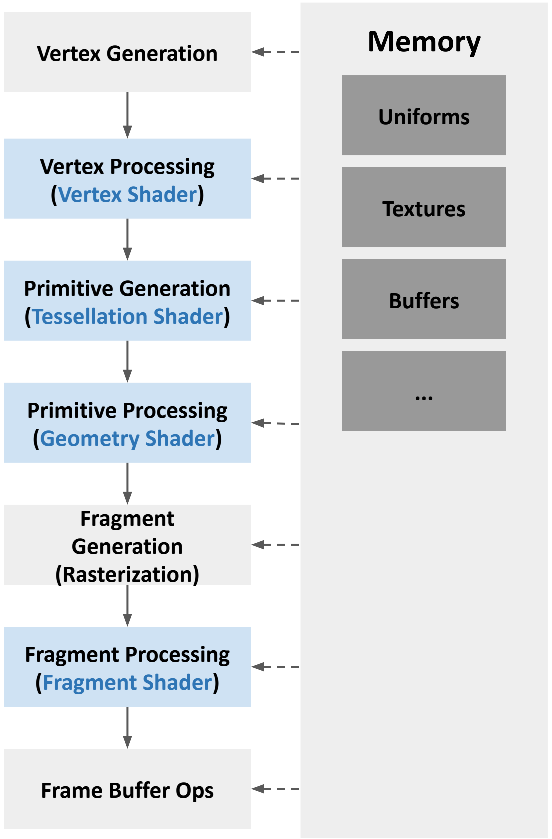
```
init frame buffer
init z buffer
for each triangle t in scene {
   tp = project(t) // MVP
   for each pixel p in frame buffer {
      if tp covers p {
         if z value at p is closer than z buffer at p {
            update z buffer and frame buffer
         }
      }
   }
}
flush frame buffer to monitor
```

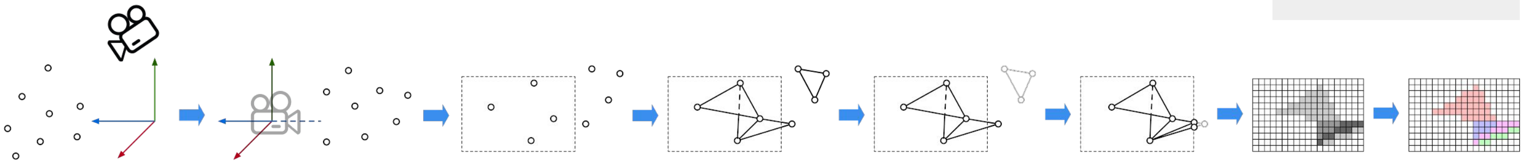




Rasterization Pipeline

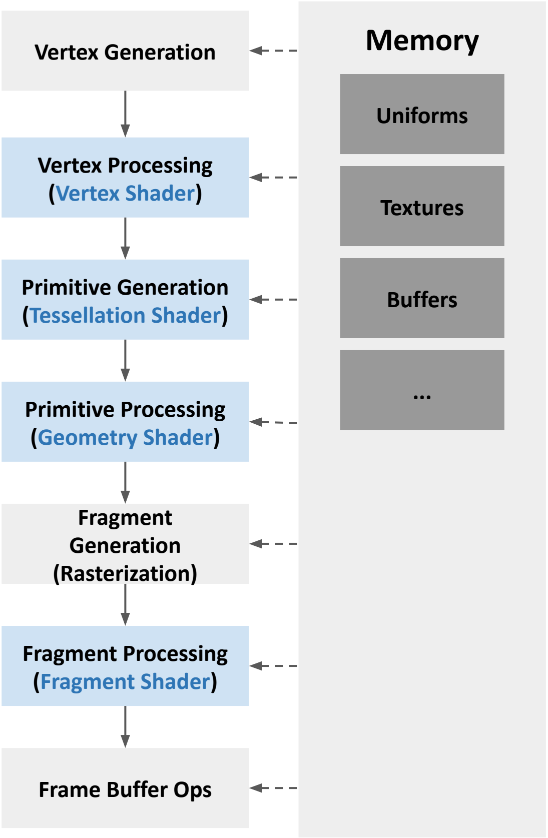
```
init frame buffer
init z buffer
for each triangle t in scene {
   tp = project(t) // MVP
   for each pixel p in frame buffer {
       if tp covers p { // culling
           if z value at p is closer than z buffer at p {
               update z buffer and frame buffer
           }
       }
   }
}
flush frame buffer to monitor
```

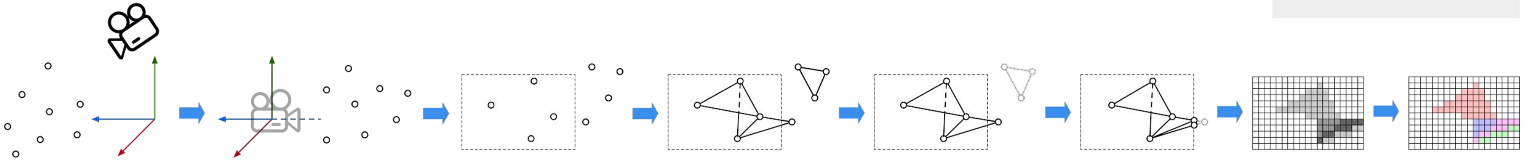




Rasterization Pipeline

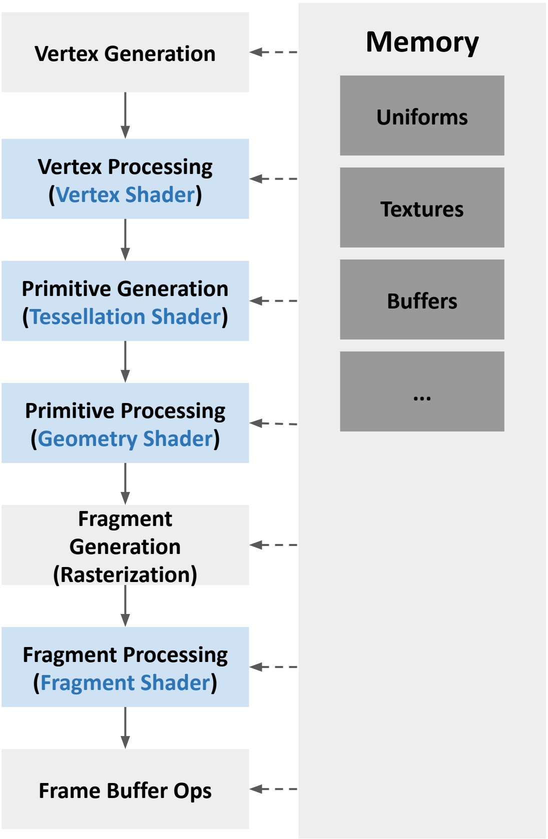
```
init frame buffer
init z buffer
for each triangle t in scene {
   tp = project(t) // MVP
   for each pixel p in frame buffer {
       if tp covers p { // culling
           if z value at p is closer than z buffer at p { // depth-test
               update z buffer and frame buffer
           }
       }
   }
}
flush frame buffer to monitor
```

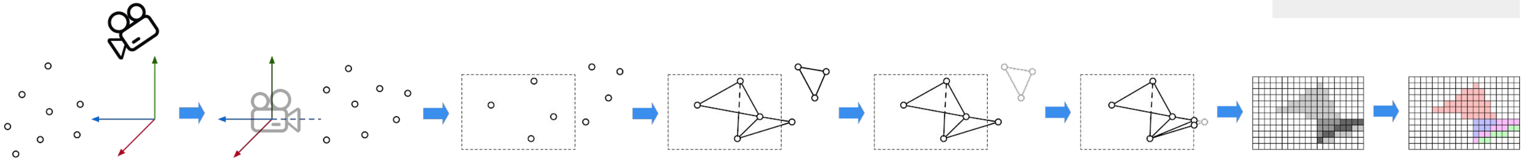




Rasterization Pipeline

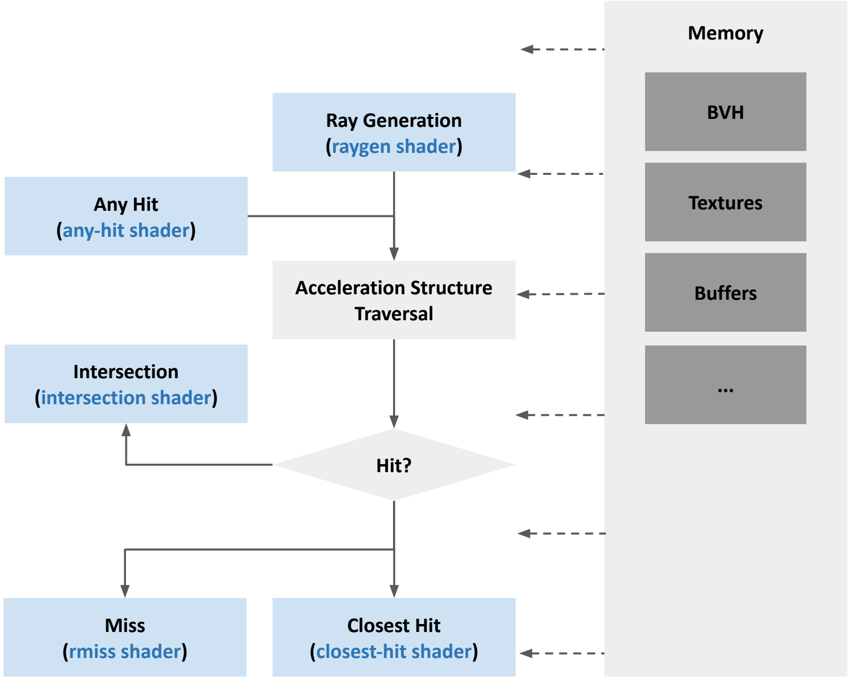
```
init frame buffer
init z buffer
for each triangle t in scene {
   tp = project(t) // MVP
   for each pixel p in frame buffer {
       if tp covers p { // culling
           if z value at p is closer than z buffer at p { // depth-test
               update z buffer and frame buffer // interpolation & update
           }
       }
   }
}
flush frame buffer to monitor
```

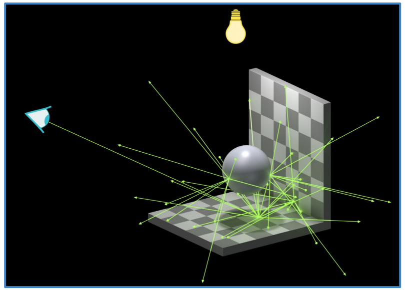


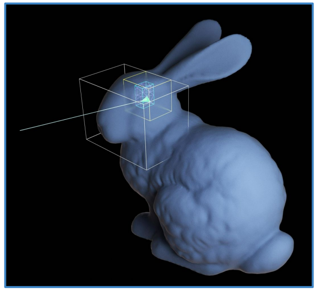


Ray Tracing Pipeline

```
init frame buffer
for each pixel p in frame buffer {
   construct a ray from p
   for ray bounces is not over {
       for each triangle t in the scene {
           if ray hit t at x {
               keep x if closest and update the ray
               break
           }
       }
   }
   update frame buffer
}
flush frame buffer to monitor
```

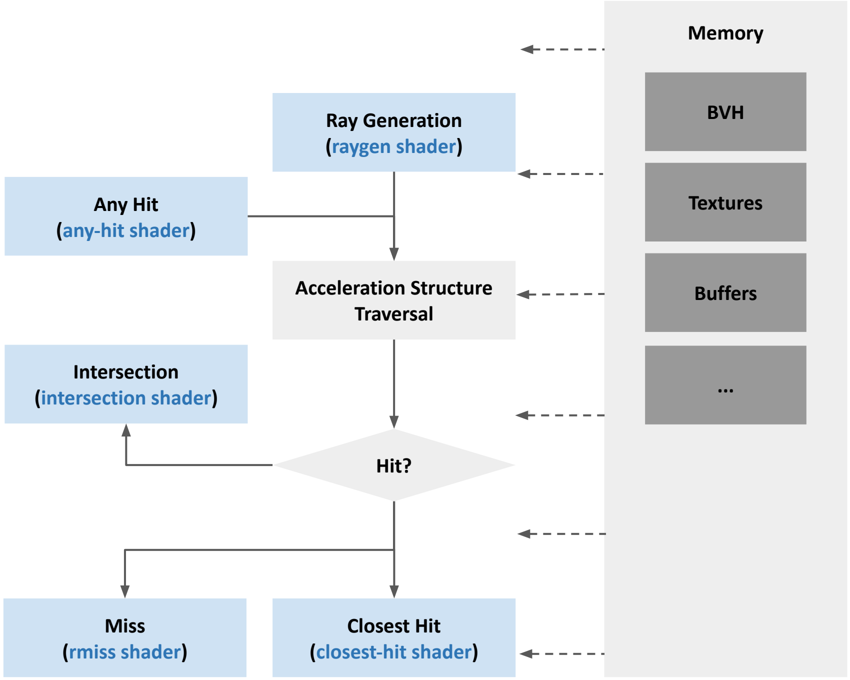


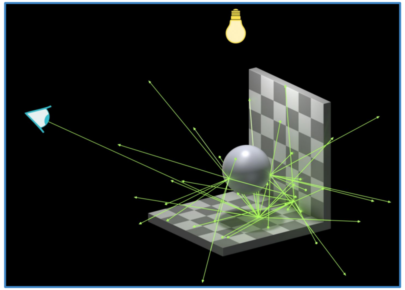


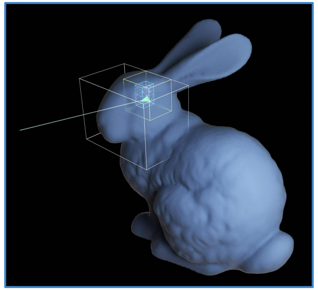


Ray Tracing Pipeline

```
init frame buffer
for each pixel p in frame buffer {
   construct a ray from p        // ray generation
   for ray bounces is not over {
       for each triangle t in the scene {
           if ray hit t at x {
               keep x if closest and update the ray
               break
           }
       }
   }
   update frame buffer
}
flush frame buffer to monitor
```

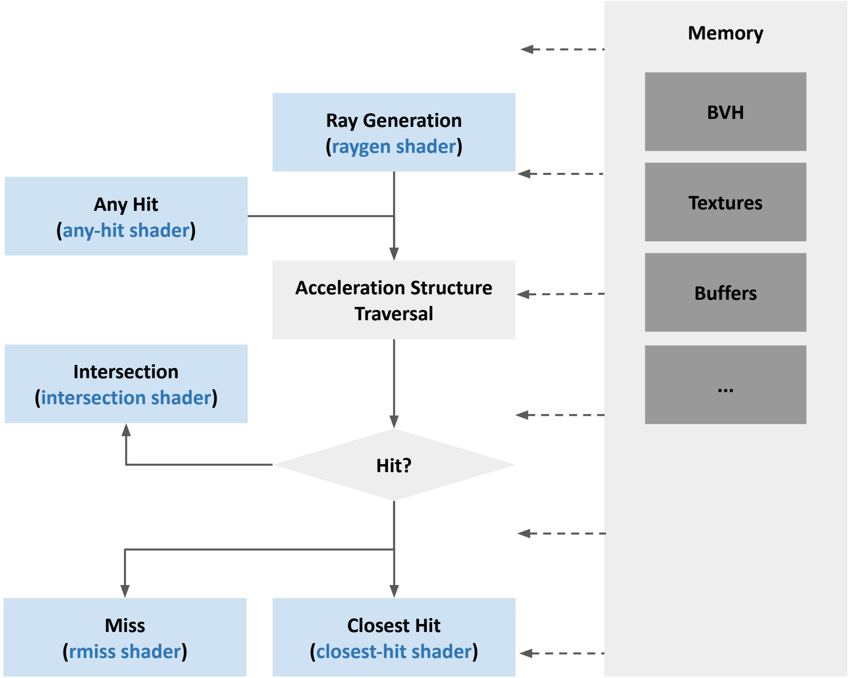


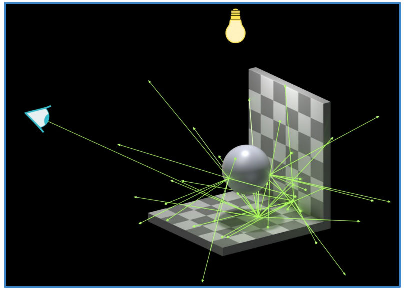


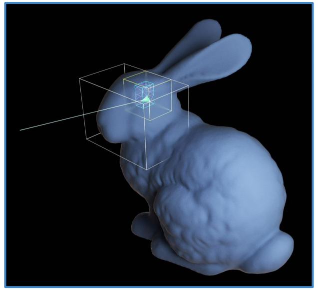


Ray Tracing Pipeline

```
init frame buffer
for each pixel p in frame buffer {
   construct a ray from p        // ray generation
   for ray bounces is not over { // russian roulette
       for each triangle t in the scene {
           if ray hit t at x {
               keep x if closest and update the ray
               break
           }
       }
   }
   update frame buffer
}
flush frame buffer to monitor
```

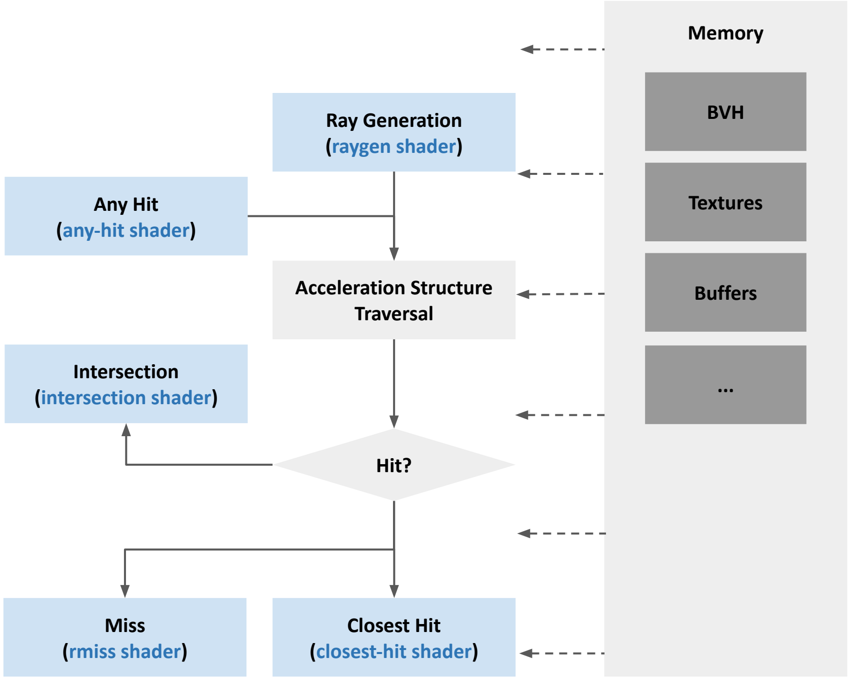


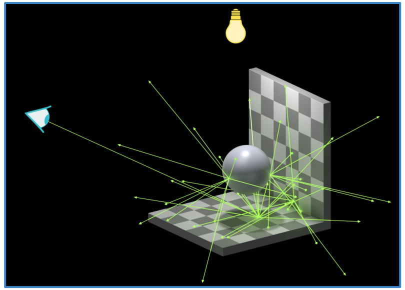


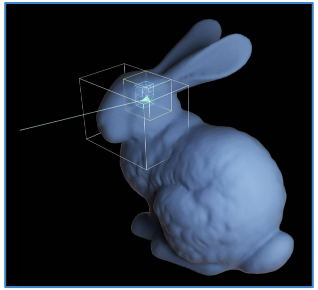


Ray Tracing Pipeline

```
init frame buffer
for each pixel p in frame buffer {
   construct a ray from p        // ray generation
   for ray bounces is not over { // russian roulette
       for each triangle t in the scene { // BVH
           if ray hit t at x {
               keep x if closest and update the ray
               break
           }
       }
   }
   update frame buffer
}
flush frame buffer to monitor
```

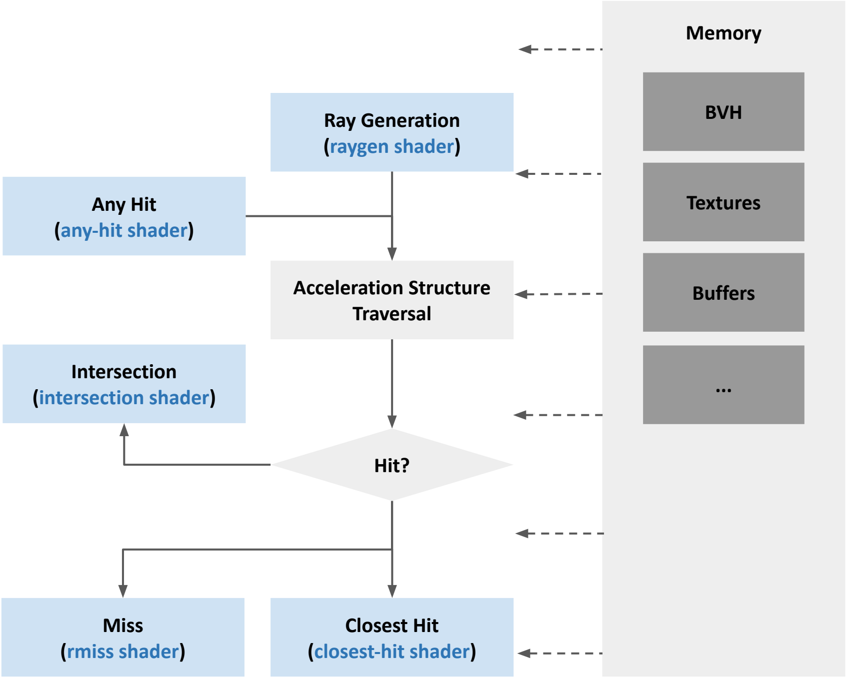


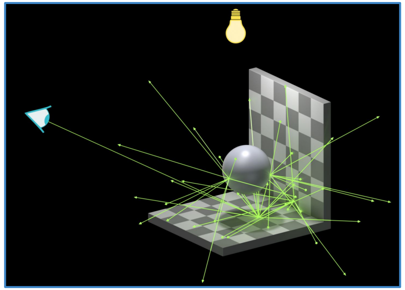


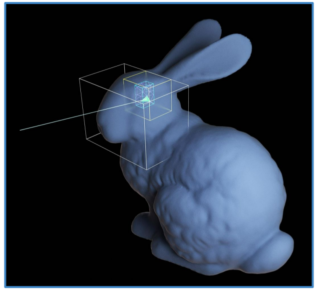


Ray Tracing Pipeline

```
init frame buffer
for each pixel p in frame buffer {
   construct a ray from p        // ray generation
   for ray bounces is not over { // russian roulette
       for each triangle t in the scene { // BVH
           if ray hit t at x {   // ray casting
               keep x if closest and update the ray
               break
           }
       }
   }
   update frame buffer
}
flush frame buffer to monitor
```







Unanswered Questions (in CG1)

- How geometric objects are created/loaded?

Unanswered Questions (in CG1)

- How geometric objects are created/loaded?
- How geometries are stored in file/memory?

Unanswered Questions (in CG1)

- How geometric objects are created/loaded?
- How geometries are stored in file/memory?
- How vertex normals/UVs are created/defined?

Unanswered Questions (in CG1)

- How geometric objects are created/loaded?
- How geometries are stored in file/memory?
- How vertex normals/UVs are created/defined?
- Why interpolation is done by barycentric coordinates instead of a different way?

Unanswered Questions (in CG1)

- How geometric objects are created/loaded?
- How geometries are stored in file/memory?
- How vertex normals/UVs are created/defined?
- Why interpolation is done by barycentric coordinates instead of a different way?
- How to deal with normals/uvs if a mesh is modified?
- …

Unanswered Questions (in CG1)

- How geometric objects are created/loaded?
- How geometries are stored in file/memory?
- How vertex normals/UVs are created/defined?
- Why interpolation is done by barycentric coordinates instead of a different way?
- How to deal with normals/uvs if a mesh is modified?
- …

Let's restart from the very beginning…

Geometry Processing Pipeline

Geometry Processing Pipeline

Scan

Processing

Geometry Processing Pipeline

Scan

Processing

Render

Print

Session 1: Introduction

- Motivation
- Recap: Graphics Pipelines
- Geometry Representations
- Blender Basics
- Summary & Homework

Representations of Geometry Objects

Point cloud

Voxels

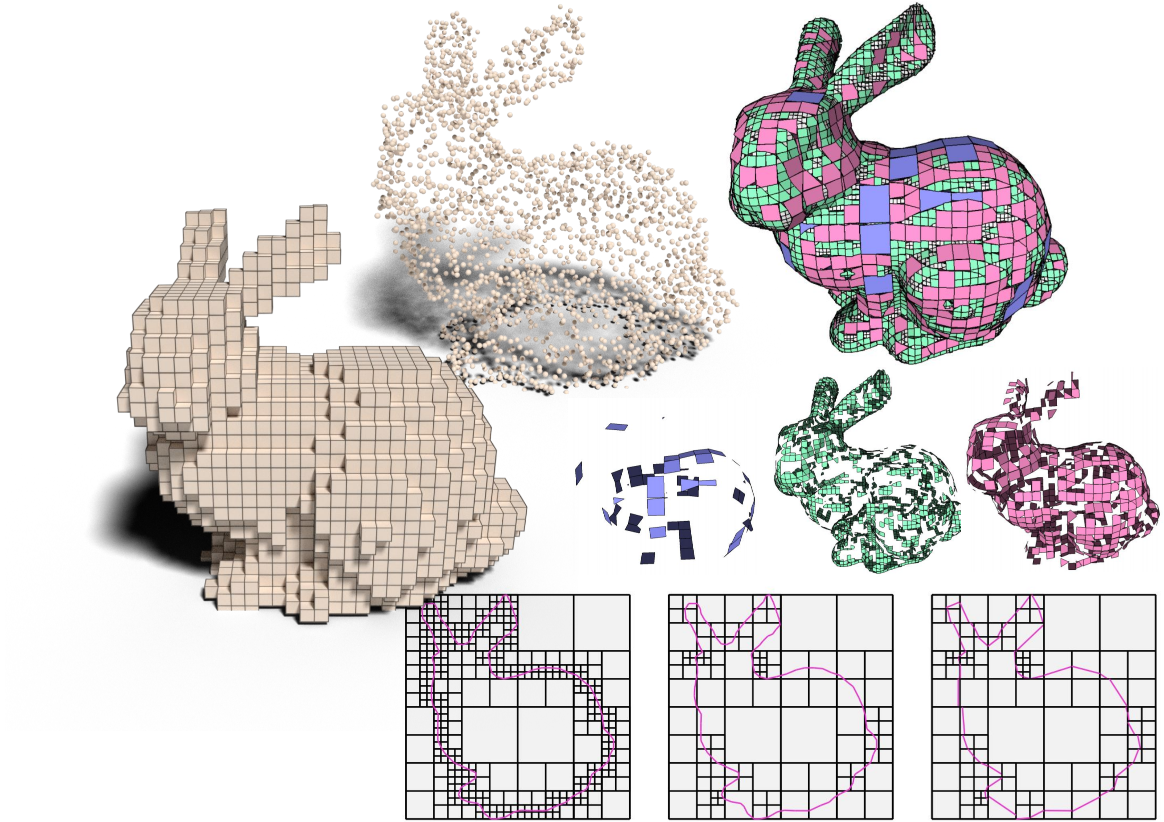
Patches

Implicit

Explicit

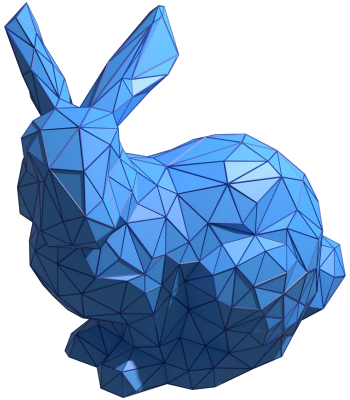
Parametric

...



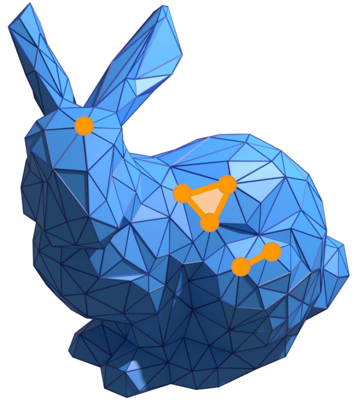
Polygonal Mesh

- A collection of polygons: a segment of a piecewise linear surface representation



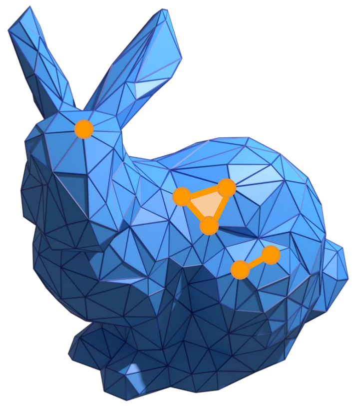
Polygonal Mesh

- A collection of polygons: a segment of a piecewise linear surface representation
- *Geometrical* component
    - Vertices $\mathcal{V} = \{v_1, v_2, ..., v_V\}, v_i \in \mathbb{R}^3$
- *Topological* components
    - Faces $\mathcal{F} = \{f_1, f_2, ..., f_F\}$
    - Edges $\mathcal{E} = \{e_1, e_2, ..., e_E\}$
- A polygonal mesh can be formulated as $\mathcal{M} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$



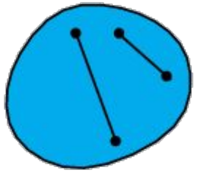
Polygonal Mesh

- A collection of polygons: a segment of a piecewise linear surface representation
- *Geometrical* component
  - Vertices $\mathcal{V} = \{v_1, v_2, ..., v_V\}, v_i \in \mathbb{R}^3$
- *Topological* components
  - Faces $\mathcal{F} = \{f_1, f_2, ..., f_F\}$
  - Edges $\mathcal{E} = \{e_1, e_2, ..., e_E\}$
- A polygonal mesh can be formulated as $\mathcal{M} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$

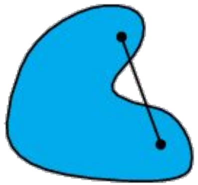


Q: How meshes are different from graphs?

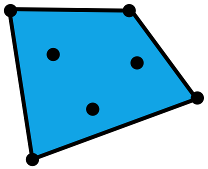
Terminologies



Convex



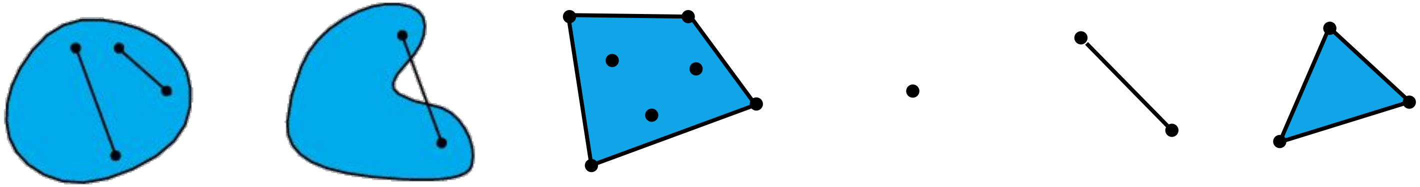
Non-convex



Convex Hull

- *Convex* and *Convex Hull*

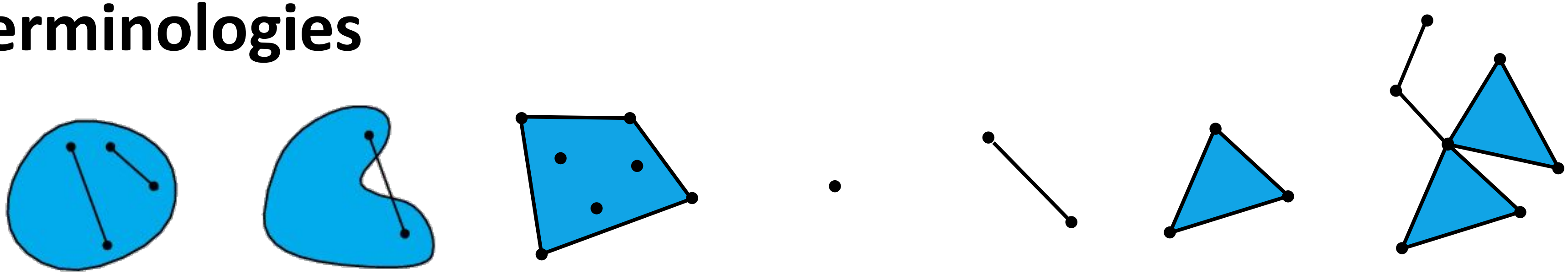
Terminologies



Convex Non-convex Convex Hull 0-simplex 1-simplex 2-simplex

- *Convex* and *Convex Hull*
- *k-Simplex*: the convex hull of k+1 affine-independent vertices
  - e.g. tetrahedra is a ***3-simplex***

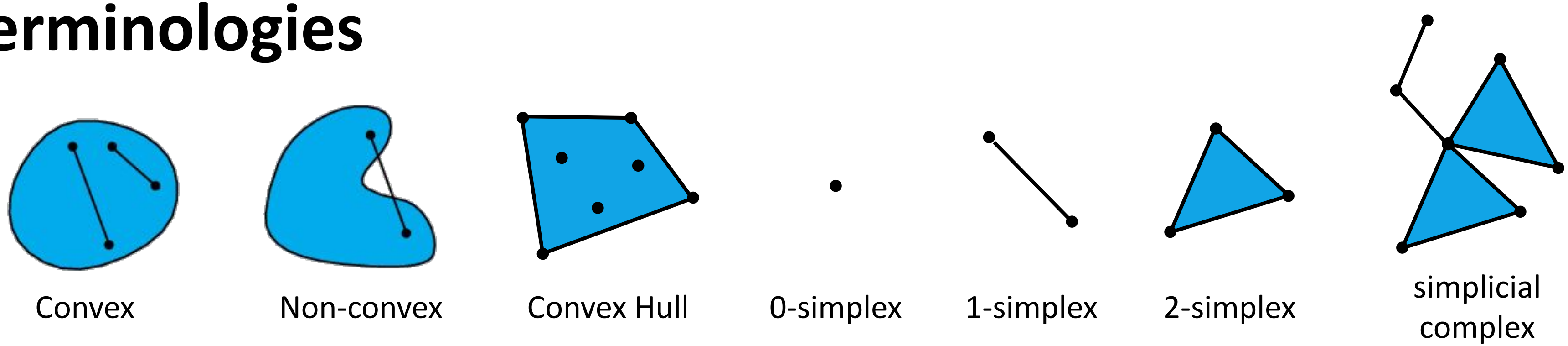
Terminologies



Convex | Non-convex | Convex Hull | 0-simplex | 1-simplex | 2-simplex | simplicial complex

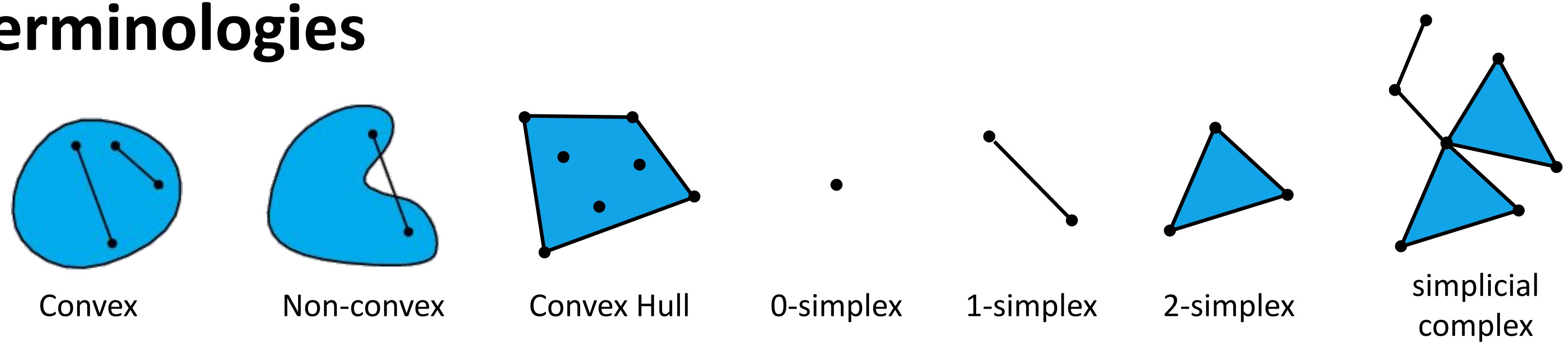
- *Convex* and *Convex Hull*
- *k-Simplex*: the convex hull of k+1 affine-independent vertices
  - e.g. tetrahedra is a ***3-simplex***
- *Face*: any simplices of a subset of vertices

Terminologies

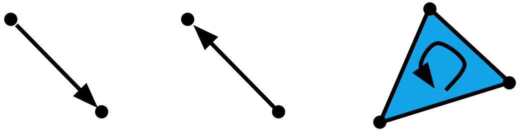


- *Convex* and *Convex Hull*
- *k-Simplex*: the convex hull of k+1 affine-independent vertices
  - e.g. tetrahedra is a ***3-simplex***
- *Face*: any simplices of a subset of vertices
- *Simplicial complex*: a collection of simplices
  - e.g. Graph is simplicial ***1-complexes,*** triangle meshes are simplicial ***2-complexes***

Terminologies

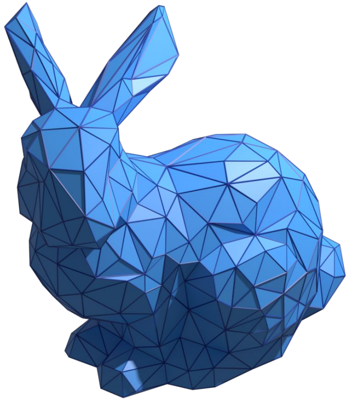


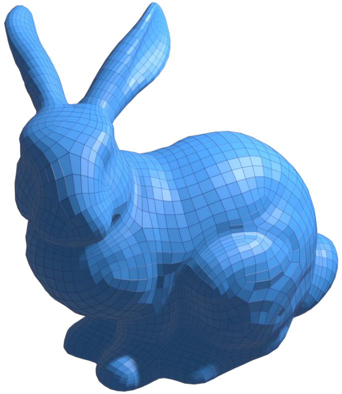
- *Convex* and *Convex Hull*
- *k-Simplex*: the convex hull of k+1 affine-independent vertices
    - e.g. tetrahedra is a ***3-simplex***
- *Face*: any simplices of a subset of vertices
- *Simplicial complex*: a collection of simplices
    - e.g. Graph is simplicial ***1-complexes**,* triangle meshes are simplicial ***2-complexes***
- Simplex can have ***orientation***



Types of Polygon Meshes

- Triangle Meshes
- Quadrilateral meshes
- …





Q: What are they in common?

Typological Invariant: *Euler-Poincaré Formula*

$$F - E + V = \boxed{2(1-g)}$$

***g*enus: #holes**

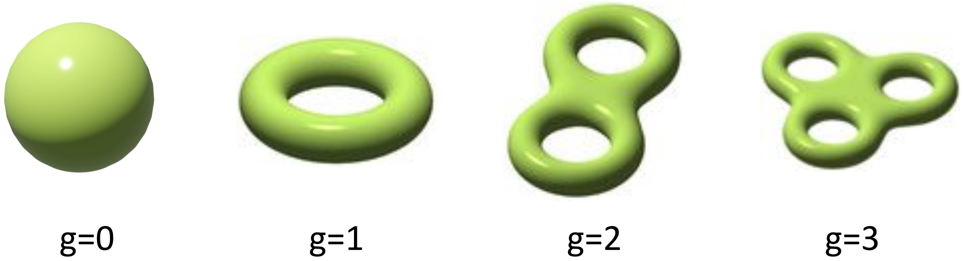
Euler characteristic

Euler characteristic allows you check topological property at constant time

e.g. the euler characteristic of a convex polyhedron is 2

Corollary (why?):

- Triangle Mesh
  - F ≈ 2V, E ≈ 3V
  - avg. vertex degree = 6
- Quad Mesh
  - F ≈ V, E ≈ 2V
  - avg. vertex degree = 4



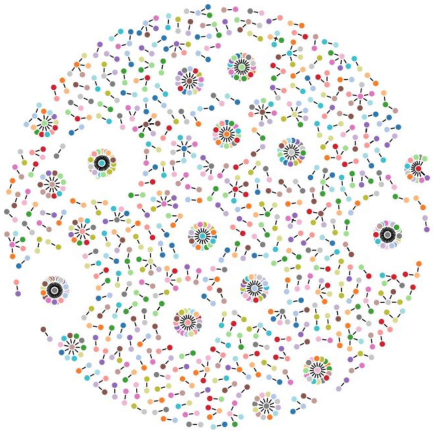
Why Polygons?

- Polygon mesh is a good surface compromise of "physical" solid
  - Think about approximation error (recall Taylor formula from calculus)
- Arbitrary topology
- Flexibility for piecewise smooth surfaces
- Flexibility for adaptive refinement (subdivision)
- Render efficiency
- ...

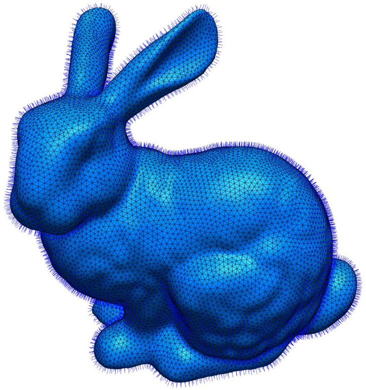
OK. Enough math. How do we actually store polygon meshes in a computer?

Mesh Data Structure: Critical Information

- **Position (x, y, z)**

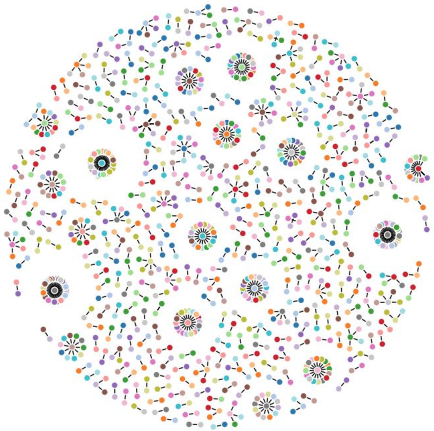


v.s.

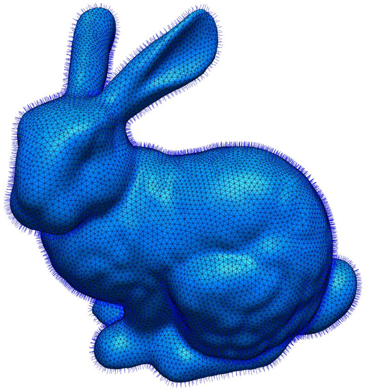


Mesh Data Structure: Critical Information

- **Position (x, y, z)**
- *Attributes, e.g. per-vertex/face normals, UV coordinates, per-vertex/face colors*

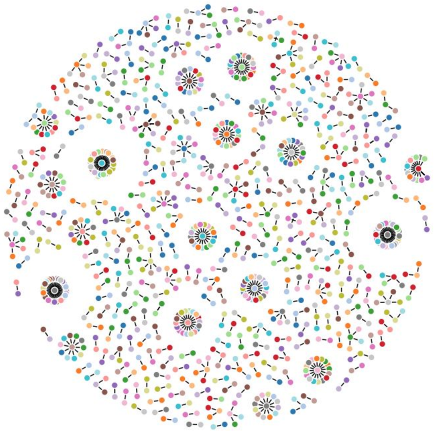


v.s.

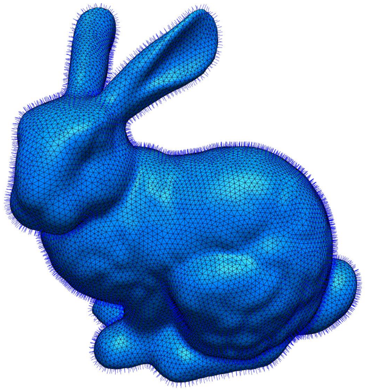


Mesh Data Structure: Critical Information

- **Position (x, y, z)**
- *Attributes, e.g. per-vertex/face normals, UV coordinates, per-vertex/face colors*
- ***Connectivity (later)***



v.s.

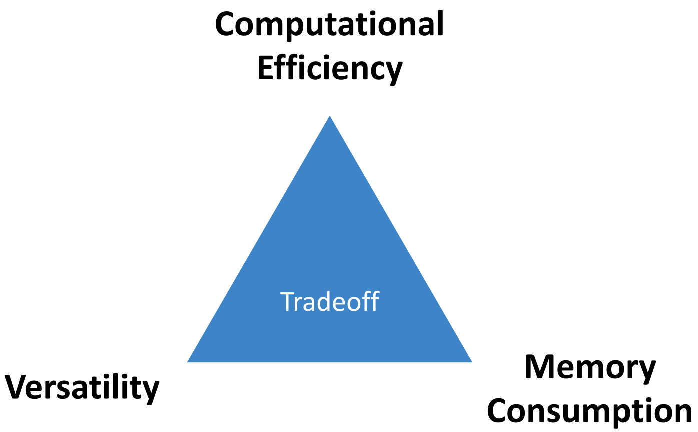


Mesh Data Structure: Optimize on-demand

- Optimize for storage, e.g. persistent on a cache/disk
- Optimize for runtime rendering tasks, e.g. OpenGL vertex buffer, BVH, etc.
- Optimize for geometry queries, e.g. What are the vertices and faces of a given face?
- Optimize for manipulation, e.g. Add/Remove/Collapse/Reconstruct a edge/face?
- ...

Mesh Data Structure: Evaluation Criteria

- Preprocessing time, e.g. How long does it take to convert to the structure from an existing (file) format?
- Query time, e.g. How long does it take to search all faces that connected to a given edge?
- Operation time, e.g. How long does it take to remove a edge/face from the structure?
- Space complexity (redundancy)



Why connectivity is important and how to represent it?

- Connectivity is critical in understanding ***local*** adjacency information of a vertex
- With connectivity information, one can avoid expensive searches
- Different types of connectivity
    - No connectivity: Face set
    - Vertex-based connectivity: Shared vertex
    - Face-based connectivity: Shared face
    - Edge-based connectivity: Shared edge
    - Halfedge-based connectivity

Why connectivity is important and how to represent it?

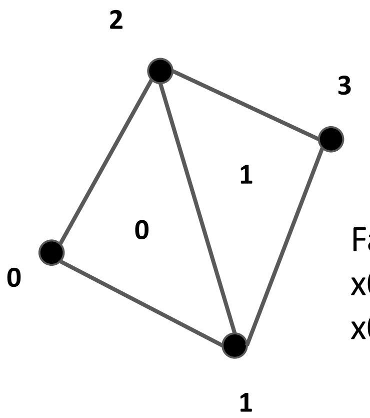
- Connectivity is critical in understanding ***local*** adjacency information of a vertex
- With connectivity information, one can avoid expensive searches
- Different types of connectivity
  - No connectivity: Face set
  - Vertex-based connectivity: Shared vertex
  - Face-based connectivity: Shared face
  - Edge-based connectivity: Shared edge
  - Halfedge-based connectivity
- We will revisit more about why local operations are so important later

Face Set (.stl format)

Basic Idea: each row stores the vertices of the face (***array***)

- Storage cost
  - 1 floating number = 4 bytes
  - 1 face = 4*9 = 36 bytes
  - 1 vertex ≈ 2*36 = 72 bytes (why?)
  - total: 72 bytes/vertex
- Pros
  - Very simple
- Cons
  - No connectivity
  - Redundancy (why?)

Triangles		
x11,y11,z11	x12,y12,z12	x13,y13,z13
x21,y21,z21	x22,y22,z22	x23,y23,z23
...	...	...
xn1,yn1,zn1	xn2,yn2,zn2	xn3,yn3,zn3



Face array:
x00,y00,z00;x01,y01,z01;x02,y02,z02
x01,y01,z01;x13,y13,z13;x02,y02,z02

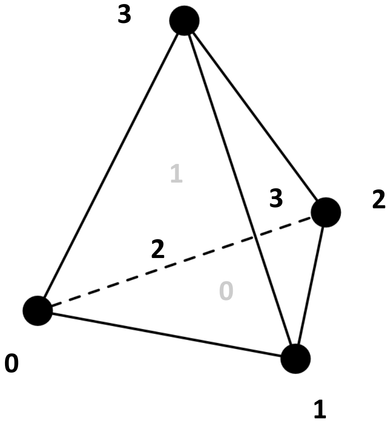
Shared Vertex (.obj, .off formats)

Basic Idea: isolate vertex positions, store an ***adjacency list*** for **vertices**

- Storage cost
  - 1 floating number = 4 bytes
  - 1 vertex = 4*3 = 12 bytes
  - 1 face = 4*2 = 12 bytes
  - total: #v * 12 + #f * 12 ≈ #v * 12 * 3 = 36 bytes/vertex
- Pros
  - Still simple, and with small redundancy
- Cons
  - No access to neighbors (why?)

Vertices
x1,y1,z1
x2,y2,z3
...
xv,yv,zv

Triangles
i11,i12,i13
i21,i22,i23
...
...
...
in1,in2,in3



Adjacency list for vertices:
0: 0 2 1
1: 0 3 2
2: 3 0 1
3: 3 1 2

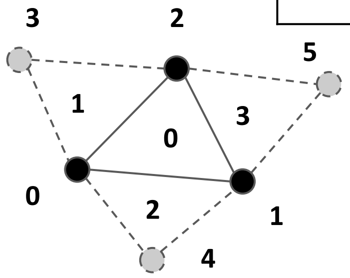
Face-based Connectivity

Basic Idea: store an ***adjacency list*** for **faces** and **neighboring faces**

- Vertex contains
  - Position
  - Associated face
- A face contains
  - Vertices
  - Face neighbors
- Pros
  - Constant time to access all neighboring faces
- Cons
  - No edge information

Vertices	
`x1,y1,z1`	`f1`
`x2,y2,z3`	`f2`
`...`	`...`
`xv,yv,zv`	`fn`

Triangles	
`i11,i12,i13`	`f11,f12,f13`
`i21,i22,i23`	`f21,f22,f23`
`...`	
`...`	
`...`	
`in1,in2,in3`	

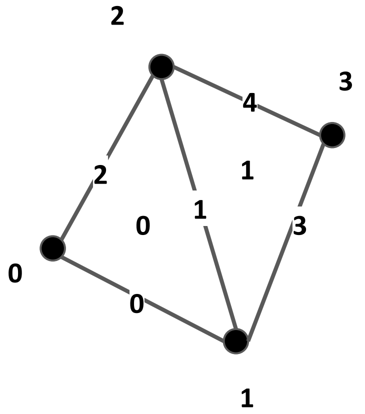


Adjacency List:
0: 0 1 2; 1 2 3
1: 0 2 3; 0
2: 0 4 1; 0
3: 1 5 2; 0

Edge-based Connectivity

Basic Idea: store an ***adjacency list*** for **edges**

- Vertex contains
  - Position
  - 1 adjacent edge index
- A edge contains
  - vertex indices
  - neighboring face indices
  - edges
- A face contains 1 edge index
- Pros: Constant time to access all neighboring faces and edges
- Cons: No edges orientation (why matters?)



Adjacency List:
0: 1 2
1: 0 3 2 4
2: 0 1
3: 4 1
4: 3 1

Incidence Matrix

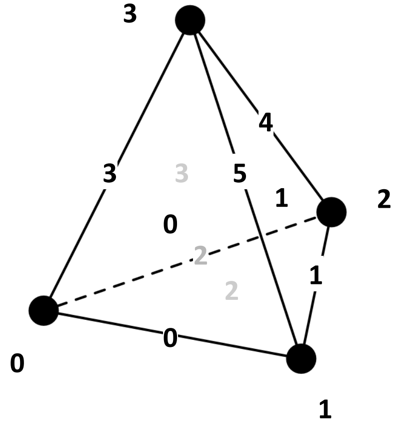
Basic idea: Store ***all*** neighbor informations via incidence matrices

Vertex-Edge Matrix

	0	1	2	3
0:	1	1	0	0
1:	0	1	1	0
2:	1	0	1	0
3:	1	0	0	1
4:	0	0	1	1
5:	0	1	0	1

Edge-Face Matrix

	0	1	2	3	4	5
0:	1	0	0	1	0	1
1:	0	1	0	0	1	1
2:	1	1	1	0	0	0
3:	0	0	1	1	1	0



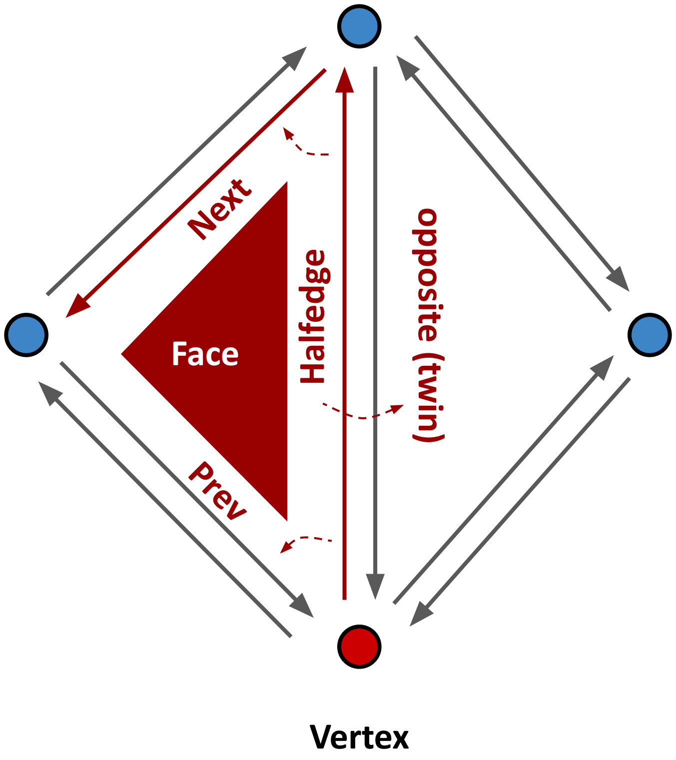
For large meshes, most of the elements will be zero

⇒ Use sparse matrix for tasks at scale

Data Structure: *Halfedge*

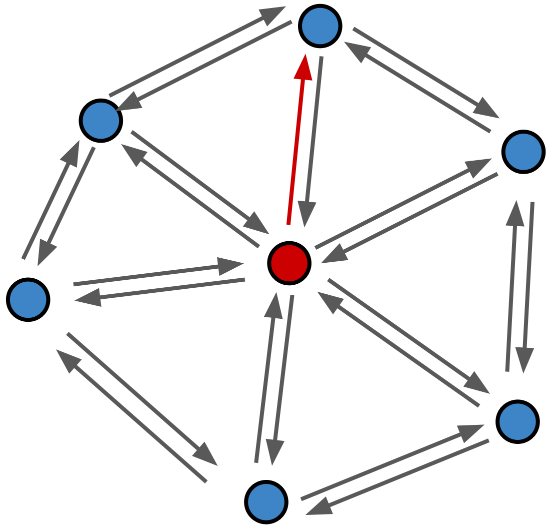
Basic idea: each edge gets split into two half edges

- Vertex
  - Position
  - 1 Halfedge
- Halfedge
  - 1 Vertex
  - 1 Face
  - Prev; Next; Opposite (Twin)
- Face
  - 1 Halfedge



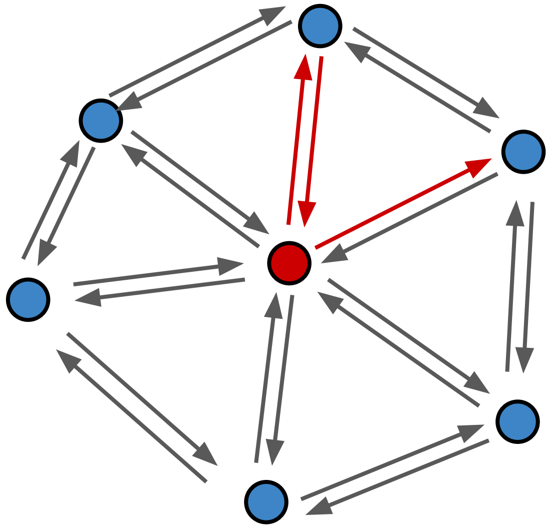
Example 1: Access All Adjacency Edges with Halfedge

```
/**
 * numAdjacentEdges returns the number of adjacency edges of the given vertex
 * @param vertex is an vertex from a haledge-based mesh
 */
function numAdjacentEdges(vertex) {
   const e0 = vertex.halfedge
   let edge_indices = [e0.index]
   for (let e = e0.opposite.next; e != e0; e = e.opposite.next) {
       edge_indices.push(e.index)
   }
   return edge_indices.length
}
```



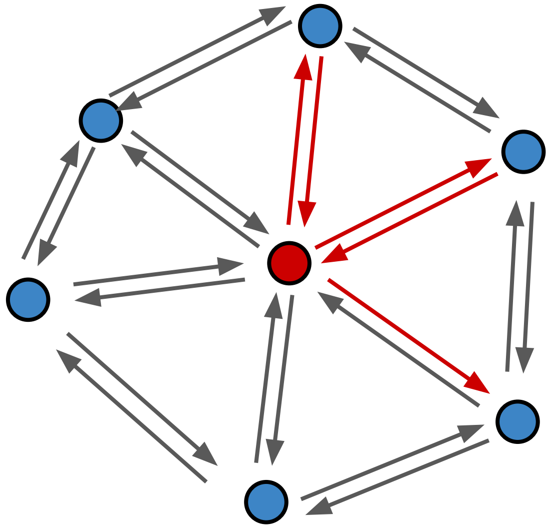
Example 1: Access All Adjacency Edges with Halfedge

```
/**
 * numAdjacentEdges returns the number of adjacency edges of the given vertex
 * @param vertex is an vertex from a haledge-based mesh
 */
function numAdjacentEdges(vertex) {
   const e0 = vertex.halfedge
   let edge_indices = [e0.index]
   for (let e = e0.opposite.next; e != e0; e = e.opposite.next) {
       edge_indices.push(e.index)
   }
   return edge_indices.length
}
```



Example 1: Access All Adjacency Edges with Halfedge

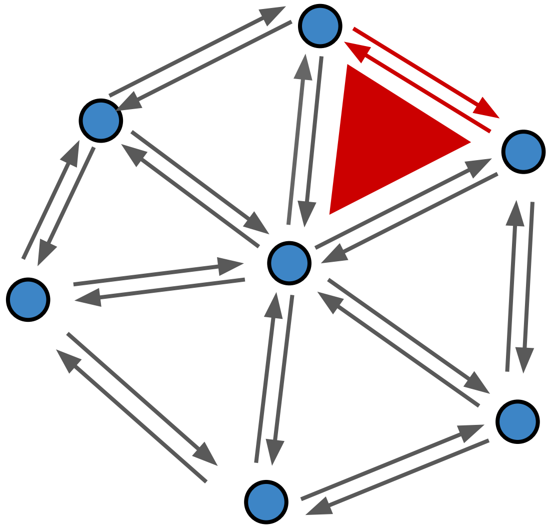
```
/**
 * numAdjacentEdges returns the number of adjacency edges of the given vertex
 * @param vertex is an vertex from a haledge-based mesh
 */
function numAdjacentEdges(vertex) {
   const e0 = vertex.halfedge
   let edge_indices = [e0.index]
   for (let e = e0.opposite.next; e != e0; e = e.opposite.next) {
       edge_indices.push(e.index)
   }
   return edge_indices.length
}
```



Example 2: Check if an edge is on the boundary of a mesh

```
class Halfedge {
  ...

  onBoundary() {
    let connectedFaces = 0
    if (this.face != null) {
      connectedFaces++
    }
    if (this.opposite.face != null) {
      connectedFaces++
    }
    if (connectedFaces != 1) {
      return false
    }
    return true
  }
}
```



See the benefits of halfedge?

- Key benefits: makes traversal easy
    - Easy for editing a mesh: constant time access of local neighbors
- Cons?

Manifold v.s. Non-Manifold

Manifold:

1. Each *edge is incident to one or two faces*, and

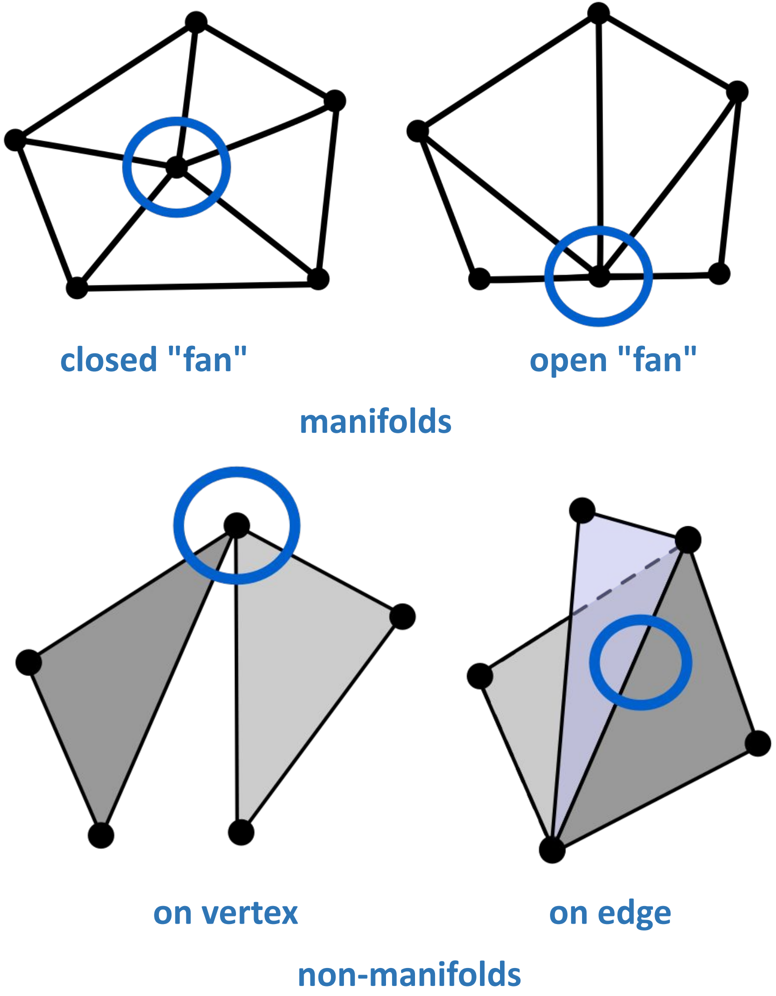
2. *faces incident to a vertex from a closed or open fan*.

Non-manifold:

1. every edges is constrained in only two polygons (no "fins")

2. the polygons containing each vertex makes a single "fan"

Q: Can halfedge structure deal with non-manifolds?



Alternatives to Halfedge

Winged edge

Corner table

Quadedge

…

Similar to halfedge and each stores local neighborhood information

General tradeoffs:

+ Convenient and better access time for individual elements, easy for traversal of locals

- Additional storage; cache incoherent; ...

More Sophisticated Mesh Data Structures

BMesh from Blender (see homework)

FbxMesh from Autodesk

FDynamicMesh from Unreal Engine

…

These data structures are not only for geometry processing purpose but also impacts the subsequent workflows, such as animation, rendering, etc.

Why Mesh Representation Is Still an Issue Today?

- Mesh is still the most *efficient/compact/structured* way to represent a solid geometric object
- Mesh structure has a ***fundamental impact*** on the way you implement an algorithm (why?)

Why Mesh Representation Is Still an Issue Today?

- Mesh is still the most *efficient/compact/structured* way to represent a solid geometric object
- Mesh structure has a ***fundamental impact*** on the way you implement an algorithm (why?)
- The increasing interests from the machine learning community, e.g.
  - How to input a mesh to a neural network?
  - How to export the output as a mesh from a neural network?

Why Mesh Representation Is Still an Issue Today?

- Mesh is still the most *efficient/compact/structured* way to represent a solid geometric object
- Mesh structure has a ***fundamental impact*** on the way you implement an algorithm (why?)
- The increasing interests from the machine learning community, e.g.
  - How to input a mesh to a neural network?
  - How to export the output as a mesh from a neural network?
- Approaching the end of Moore's law: The needs for a concurrent-safe mesh structure

Why Mesh Representation Is Still an Issue Today?

- Mesh is still the most *efficient/compact/structured* way to represent a solid geometric object
- Mesh structure has a ***fundamental impact*** on the way you implement an algorithm (why?)
- The increasing interests from the machine learning community, e.g.
  - How to input a mesh to a neural network?
  - How to export the output as a mesh from a neural network?
- Approaching the end of Moore's law: The needs for a concurrent-safe mesh structure

- Hopefully we will have time to revisit theses in the last session (7-data-driven-shape-analysis)
- Be careful & patient 😀
- Let's now go for some thing maybe more funny...

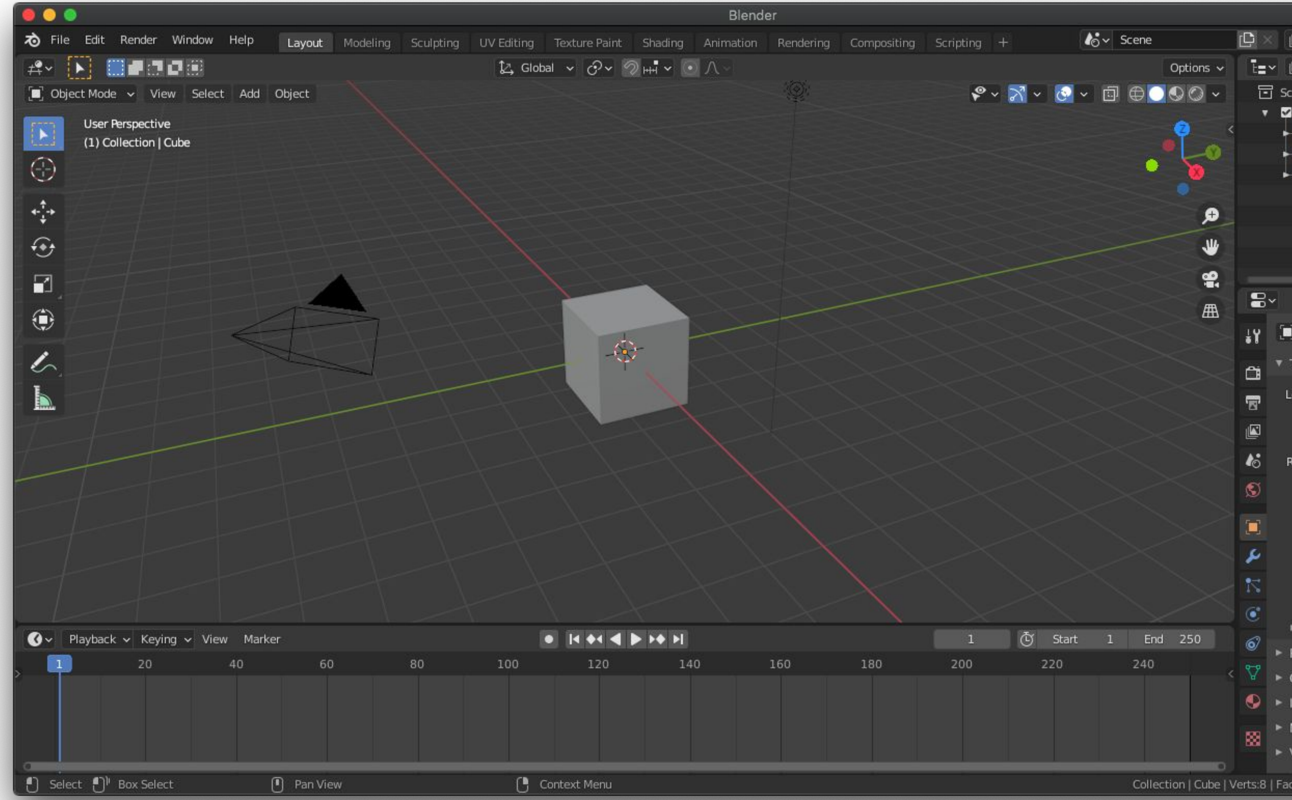
Session 1: Introduction

- Motivation
- Recap: Graphics Pipelines
- Geometry Representations
- Blender Basics
- Homework

Blender

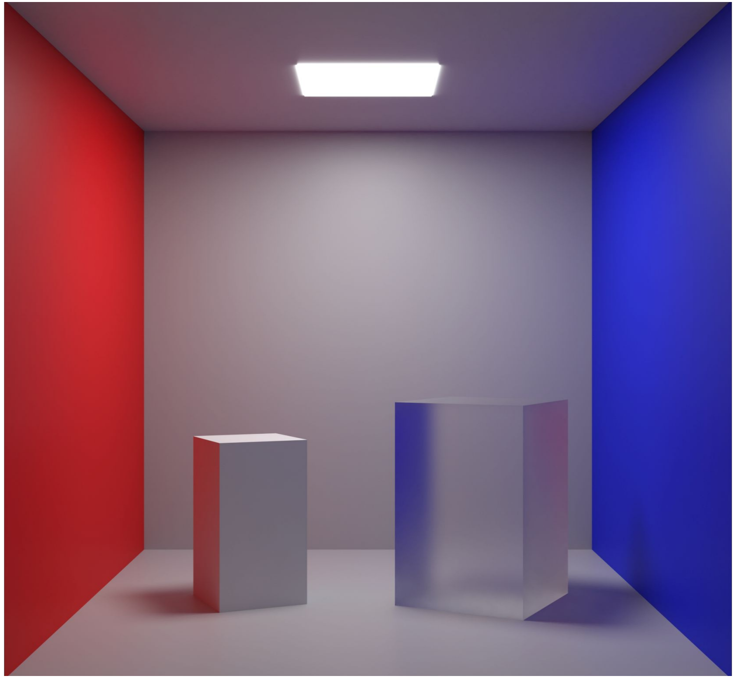
Why not XYZ?

Because of *open source*



Live Demo: Reproduce The Cornell Box in Blender

- What are the geometry objects do we need for the Cornell box?
- What are their properties?



Session 1: Introduction

- Motivation
- Recap: Graphics Pipelines
- Geometry Representations
- Blender Basics
- **Summary & Homework**

Summary

- Geometry representation has a long term impact on the processing pipeline
- Mesh is a good compromise and connectivity is critical for local operations
- No-free Lunch! ⇒ Think about your task and choose the right structure
- Understanding the modeling process helps understand more processing workflows

	Adjacency List Based Structure	Incident Matrices Based Structure	Halfedge Based Structure
Constant-time neighborhood access?	No	Yes	Yes
Easy to remove elements?	No	No	Yes
Deal with non-manifold geometry?	Yes	Yes	No

Homework 1 (recommended, no submission required)

1. Re-implement the rasterization and ray tracing pipeline
2. Getting started with Blender
3. Extend the rasterizer, *implement an .obj file loader that loads the model*
    - Can you implement all data structure that you learned today?
4. **Start think about your individual project**

See more details in https://github.com/mimuc/gp/tree/ws2021/homeworks/1-intro.

Thanks! What are your questions?

Next session: Discrete Differential Geometry