Basic Techniques in Computer Graphics

Winter 2017 / 2018

The slide comments are not guaranteed to be complete, they are no alternative to the lectures itself. So go to the lectures and write down your own comments!

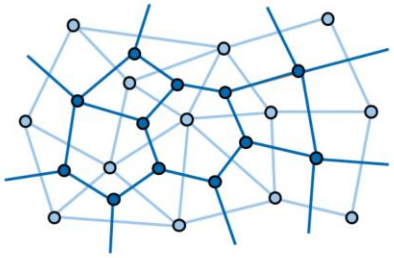
Geometric Shapes

- **Geometric Shapes**
  - **Polygonal meshes**
  - Constructive solid geometry
- Scene representations
  - Scene graphs
  - Culling
  - Optimization structures

After discussing the general concepts and basics we will now go into more detail of higher-level concepts. This chapter will look at geometry again, later we will look at advanced topics in rendering.

Polygonal Meshes

- Polygonal meshes consist of
  - **Geometry** (vertices)
  - **Topology** (edges, faces)
- Dual Meshes



The geometry defines the form, the positions while the topology defines the neighborhood, the connection of the geometry. The bright yellow mesh shows a polygon mesh, the darker orange the dual mesh defined as the connections of the centers of the triangles of the input mesh.A manifold surface has the potential to enclose a volume. Potential here means, that even a mesh with a small hole in it still defines a volume and is acceptable. On the other hand a model shaped like an 8 is not well defined as the surface has no well defined outside. For triangles meshes we define:
1) each edge is shared by one or more triangles
2) the edges/faces around a vertex form a half cycle
3) orient-ability (ex: a möbius strip fulfills 1&2 but is not orientable

has to hold so it is manifold.

Polygonal Meshes

- Mesh consistency
  - 2-manifold *(edge ordering → local disks)*
  - closed meshes *(no boundary edges)*
- Euler: **V - E + F + H = 2 (1-g)**
  - **V**ertices, **E**dges, **F**aces
  - **H**oles
  - **g**enus (separating cuts)
  - Proof ...
- Triangle meshes:
  - Magic numbers: „2“, „6“, „5“

A mesh is manifold if it has the potential to enclose a volume.

Polygonal Meshes

- Mesh consistency
  - 2-manifold *(edge ordering → local disks)*
  - closed meshes *(no boundary edges)*
- Euler: **V - E + F = 2 (1-g)**
  - **V**ertices, **E**dges, **F**aces
  - Holes
  - **g**enus (separating cuts)
  - Proof ...
- Triangle meshes:
  - Magic numbers: „2“, „6“, „5“

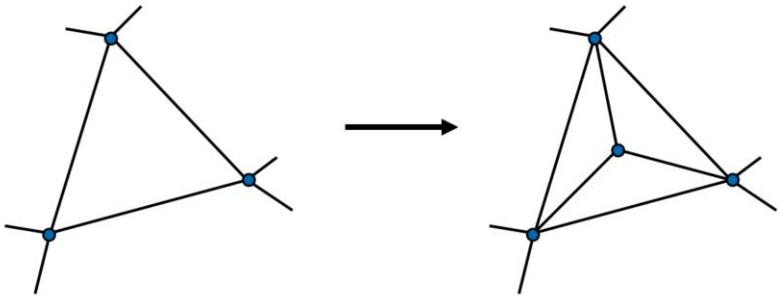
The simplified Euler formula ignoring holes.

Genus

- topological property
  (independent wrt. regular deformations)
- number of handles
- maximum number of closed cuts such that the surface still remains connected
- sphere, torus, double torus, pretzel, ...

Genus (a topological property):
eg: A sphere is a genus 0 object
eg: A torus is a genus 1 object
The genus tells us how many handles an object has.
The genus tells us how many cuts (maximal) we can make until the object falls into pieces.

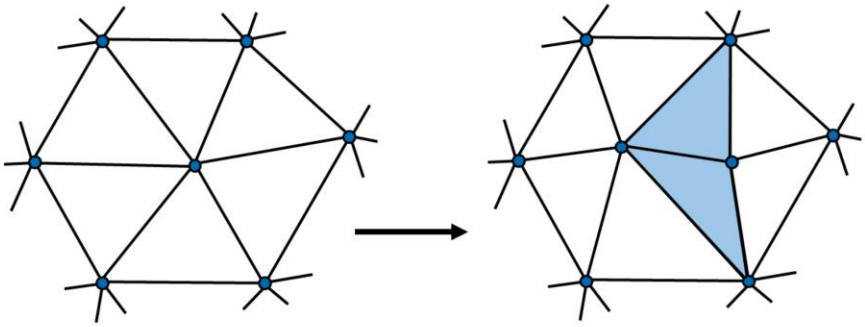
Operations on Polygonal Meshes



$$V - E + F = 2(1-g)$$

$$1 - 3 + 2 = 0$$

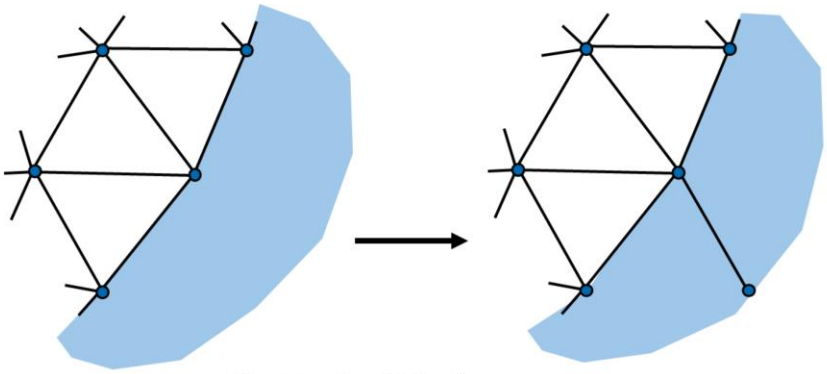
Operations on Polygonal Meshes



$$V - E + F = 2(1-g)$$

$$1 - 3 + 2 = 0$$

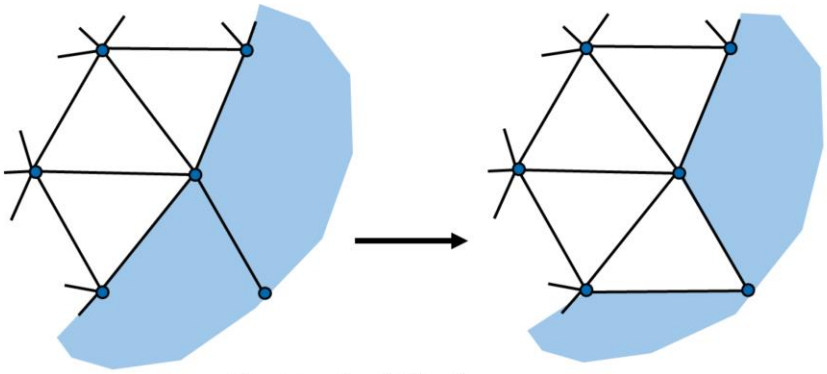
Operations on Polygonal Meshes



$$V - E + F = 2(1-g)$$

$$1 - 1 + 0 = 0$$

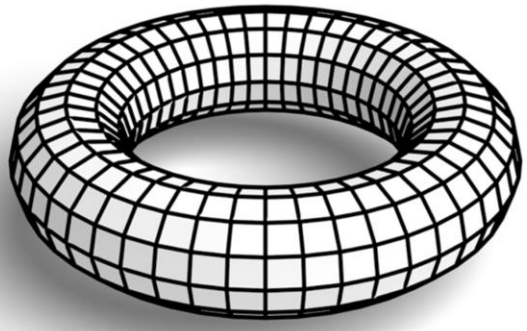
Operations on Polygonal Meshes



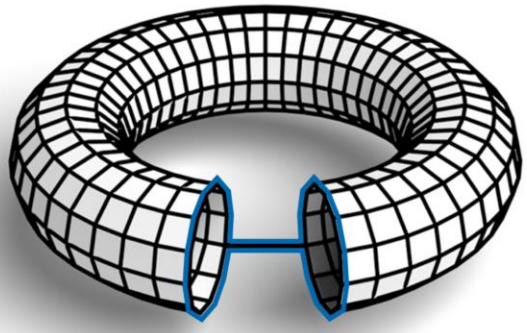
$$V - E + F = 2(1-g)$$

$$0 - 1 + 1 = 0$$

Operations on Polygonal Meshes



Operations on Polygonal Meshes



$V - E + F = 2(1-g)$

$0 - 1 - 1 = -2$

Platonic Solids

- *Platonic solid*
  - convex polyhedron
  - congruent convex regular faces
- Properties
  - equal vertex / face valence
  - equal angles
- *Schläfli*-Symbols:
  - {p, q} = p-gons, vertex valence q

The valence is the number of neighbors of a vertex (=number of edges joining in that vertex) or face (= number of edges around it).

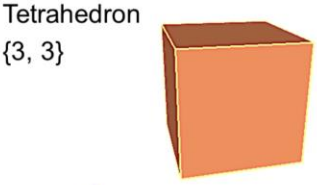
Platonic Solids

- V – E + F = 2 (1-g) = 2
- p ... HE = 2E = p F and p ≥ 3
- q ... HE = 2E = q V and q ≥ 3
- (2/q – 1 + 2/p) E = 2
- positive integer solutions: (2/q + 2/p) > 1
- {p,q} ∈ [{3,3}, {3,4}, {4,3}, {3,5}, {5,3}]

p = Face valence,
q = Vertex valence

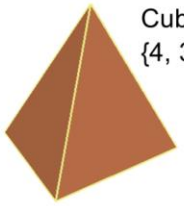
Platonic Solids

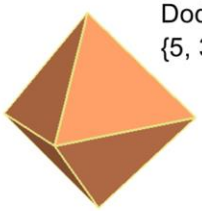
Tetrahedron
{3, 3}



Cube
{4, 3}

Dodecahedron
{5, 3}



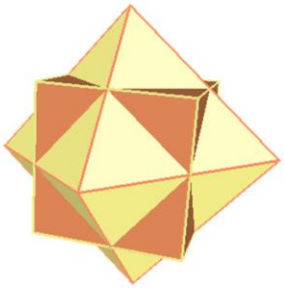


Octahedron
{3, 4}

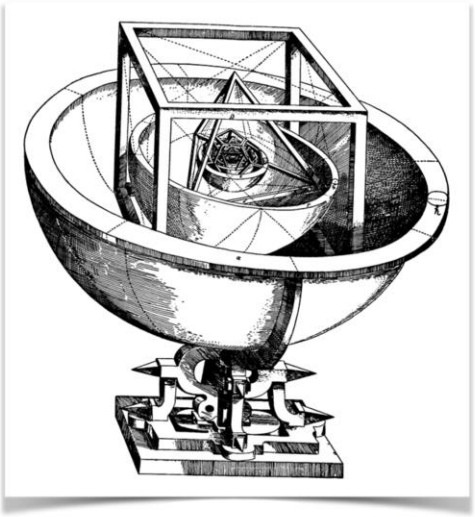
Icosahedron
{3, 5}

Platonic Solids

- Duality:
  - **Cube {4, 3}** ↔ Octahedron {3, 4}
  - Dodocahedron {5, 3} ↔ **Icosahedron {3, 5}**



Kepler‘s World View



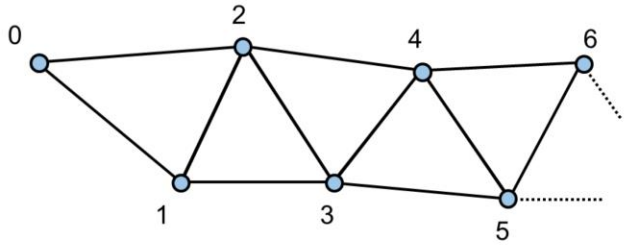
File Formats

- Face Set (e.g. STL, VRML)
  - faces: $[x_{3i}, y_{3i}, z_{3i}]$, $[x_{3i+1}, y_{3i+1}, z_{3i+1}]$, $[x_{3i+2}, y_{3i+2}, z_{3i+2}]$
- Shared vertex (e.g. OFF, OBJ, VRML)
  - vertices: $[x_i, y_i, z_i]$
  - face indices: [i, j, k]
- Triangle strips (efficient rendering)
  - two *transformed* vertices are cached
  - connectivity implicitly defined by vertex order
- Triangle fans
  - two *transformed* vertices are cached
  - connectivity implicitly defined by vertex order

Explicit connectivity:
Face Set: Define each face by all three vertices, simple but redundant.
Shared vertex: Define each face by indices into a vertex list, no redundancies for the vertices.
Implicit connectivity:
Triangle strip/fan

Triangle Strips

- cache transformed vertices
- triangles: {0,1,2}, {2,1,3}, {2,3,4}, ...
- larger caches !?



OpenGL can directly render triangle strips and triangle fans. Shared vertices often only have to be transformed once by the Vertex Shader (however, this in not guaranteed).

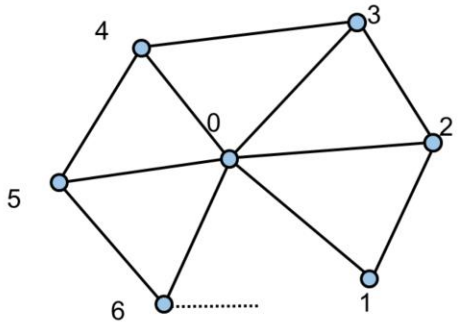
Triangle Strips

- regular triangle strips
  - 2+n vertices for n triangles
- generalized triangle strips
  - 2+n vertices plus n-1 bit for n triangles
- single strip triangulation
  - strip length depends on mesh regularity
  - slight modification of the mesh (ca. 2%)

A mesh can be modified slightly to be express-able completely as one triangle strip (generally 2% of the mesh has to be modified).
Alternatively, OpenGL provides a fast way to render multiple non-connected triangle strips in one render call (google ‚primitive restart‘).

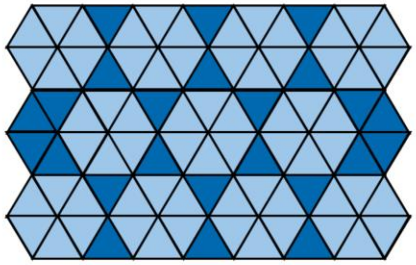
Triangle Fans

- cache transformed vertices
- triangles: {0,1,2}, {0,2,3}, {0,3,4}, ...



Triangle Fans

- expected length: 6 triangles
  - 2 + n vertices for n triangles
  - 8 vertices for 6 triangles
- average length: < 6 triangles
  - greedy generation of triangle fans leads to fragmentation

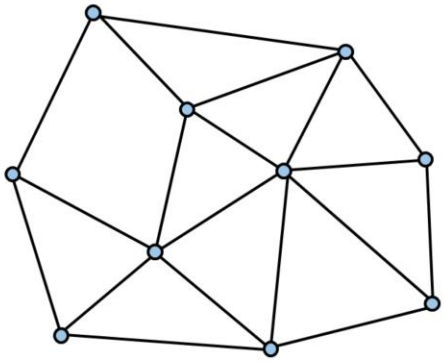


Mesh Data Structures

- How to store topology / connectivity?
- Geometric algorithms on meshes:
  - Identify time-critical operations
  - One-ring neighborhood
- Construct neighborhood relation

Mesh Data Structures

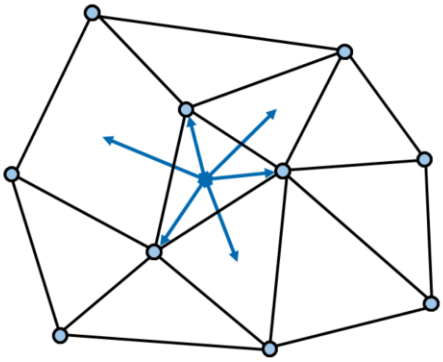
Store topology
- Face based
- Edge based
- Halfedge based



Mesh Data Structures

Store topology

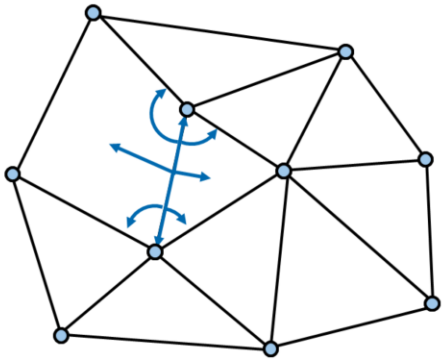
- Face based:
  - n face vertices
  - n neighboring faces
- Edge based
- Halfedge based



Face based: each Face gets saved (= all vertices that define the face) and each face saves the neighboring faces. Note: faces can have a varying number of vertices.

Mesh Data Structures

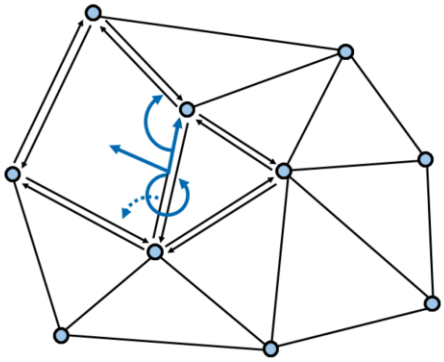
Store topology
- Face based
- Edge based:
  - 2 vertices
  - 2 incident faces
  - 4 edges
- Halfedge based



As each edge has exactly two vertices, these and the two faces can get stored together with neighboring edges.

Mesh Data Structures

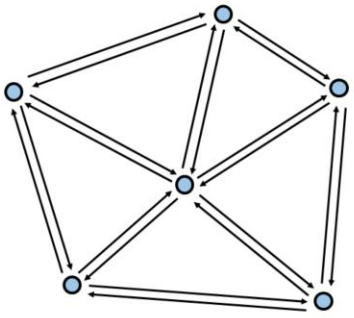
Store topology
- Face based
- Edge based
- Halfedge based:
  - 1 vertex
  - 1 face
  - Next halfedge
  - Opposite halfedge



Halfedge based mesh representation is more efficient in terms of memory consumption and also makes it easy to traverse the mesh.

One-Ring Neighborhood

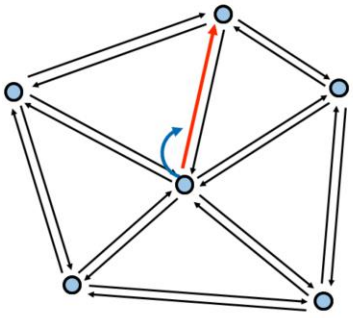
1. Start at vertex



Example: how to find the one-ring neighborhood of one vertex.

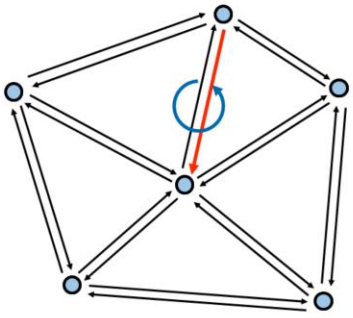
One-Ring Neighborhood

1. Start at vertex
2. Outgoing halfedge



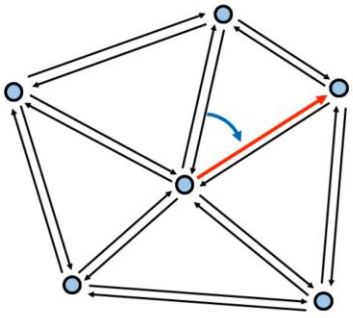
One-Ring Neighborhood

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge



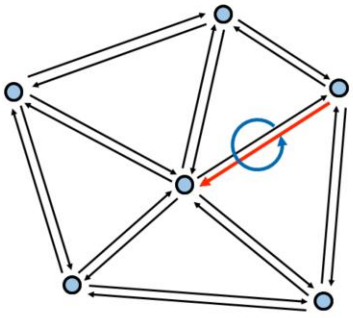
One-Ring Neighborhood

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge



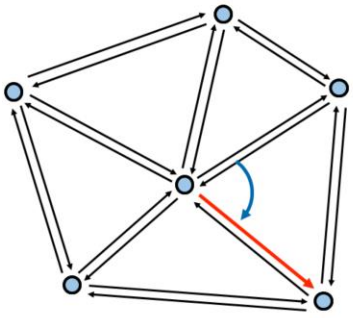
One-Ring Neighborhood

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge
5. Opposite



One-Ring Neighborhood

1. Start at vertex
2. Outgoing halfedge
3. Opposite halfedge
4. Next halfedge
5. Opposite
6. Next
7. ...



Geometric Shapes

- **Geometric Shapes**
  - Polygonal meshes
  - **Constructive solid geometry**
- Scene representations
  - Scene graphs
  - Culling
  - Optimization structures

After discussing the general concepts and basics we will now go into more detail of higher-level concepts. This chapter will look at geometry again, later we will look at advanced topics in rendering.

Basic Primitives

- Sphere
- Cylinder
- Cone

} *Quadrics :* [x y z 1] **Q** [x y z 1]^T = 0

- Torus

} *fourth order*

- Box (skew / rectangular)

$$a_i \leq n_i^T \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq b_i \quad , \quad i = 1, 2, 3$$

Sphere

$$\|\mathbf{p}-\mathbf{c}\|^2 - r^2 = \left\| \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} - \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} \right\|^2 - r^2 = \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}^{\mathrm{t}} \mathbf{Q} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ -c_x & -c_y & -c_z & q \end{pmatrix} \qquad q = c_x^2 + c_y^2 + c_z^2 - r^2$$

Parametric:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cos\phi\cos\theta \\ r\sin\phi\cos\theta \\ r\sin\phi \end{pmatrix} \qquad 0 \le \phi \le 2\pi, -\tfrac{1}{2}\pi \le \theta \le \tfrac{1}{2}\pi$$

Center of the sphere: (Mx,My,Mz)
Radius can be derived from P and the center

Cylinder

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & 0 \\ -c_x & -c_y & 0 & q \end{pmatrix} \quad q = c_x^2 + c_y^2 - r^2$$

Parametric:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cos\phi \\ r\sin\phi \\ h \end{pmatrix} \quad 0 \leq \phi \leq 2\pi, h_0 \leq h \leq h_1$$

Cone

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & -\tan^2\alpha & \tan^2\alpha \cdot c_z \\ -c_x & -c_y & \tan^2\alpha \cdot c_z & q \end{pmatrix}$$

$$q = c_x^2 + c_y^2 - \tan^2\alpha \cdot c_z^2$$

Parametric:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} c_x + h\tan\alpha \cdot \cos\psi \\ c_y + h\tan\alpha \cdot \sin\psi \\ c_z + h \end{pmatrix} \quad 0 \le \psi \le 2\pi, h_0 \le h \le h_1$$

Ray Intersection

- ray in parametric form: $p + \lambda r$
- object in implicit form: $v^T Q v = 0$

Ray Intersection

- ray in parametric form: $p + \lambda r$
- object in implicit form: $v^T Q v = 0$
- $(p + \lambda r)^T Q (p + \lambda r) = 0$
- $p^T Q p + 2 \lambda p^T Q r + \lambda^2 r^T Q r = 0$

Transformation of Quadrics

- Quadric Q : $v^T Q v = 0$
- Affine transformation M : $Q \rightarrow Q'$
- Apply inverse transform M^{-1} to points

- $v^T Q' v = 0 \quad \Leftrightarrow \quad (M^{-1}v)^T Q (M^{-1}v) = 0$
  $\Leftrightarrow \quad Q' = (M^{-1})^T Q M^{-1}$

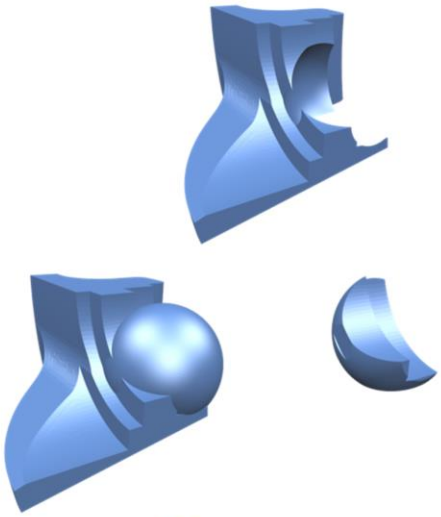
Torus

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos\varphi \ (R + r\cos\theta) \\ \sin\varphi \ (R + r\cos\theta) \\ r \ \sin\theta \end{pmatrix}$$

$$0 \le \phi \le 2\pi, \quad 0 \le \theta \le 2\pi$$

Constructive Solid Geometry

- Combine basic primitives
- Boolean Operations
  - Intersection
  - Union
  - Subtraction



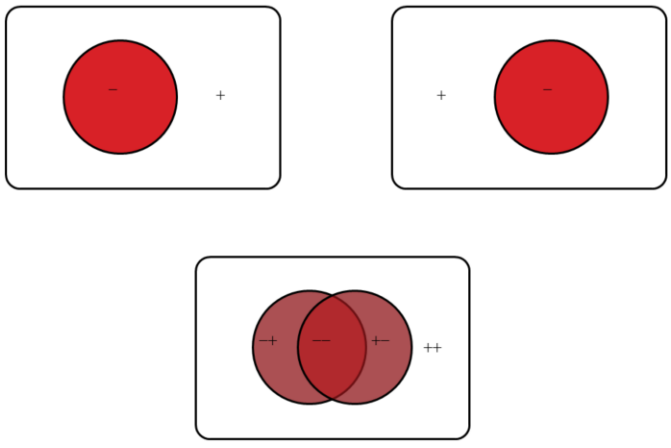
Boolean operations can be used to combine primitives to create more complex objects.

Implicit Representation

- Surface: $F(x,y,z) = 0$
- Interior: $F(x,y,z) < 0$
- Exterior: $F(x,y,z) > 0$
- Caution: in general, F(x,y,z) does ***not*** measure the distance to the surface

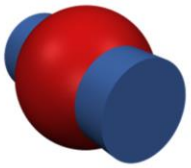
  ($\rightarrow$ *signed distance function*)

Boolean Operations



Boolean Operations

Union(F,G) = min(F,G)

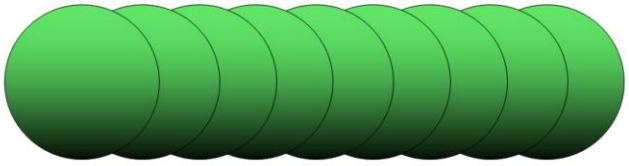


Intersect(F,G) = max(F,G)

Subtract(F,G) = max(F,-G)

Union, intersection and subtraction can be defined as min/max functions.

Example: Milling Simulation

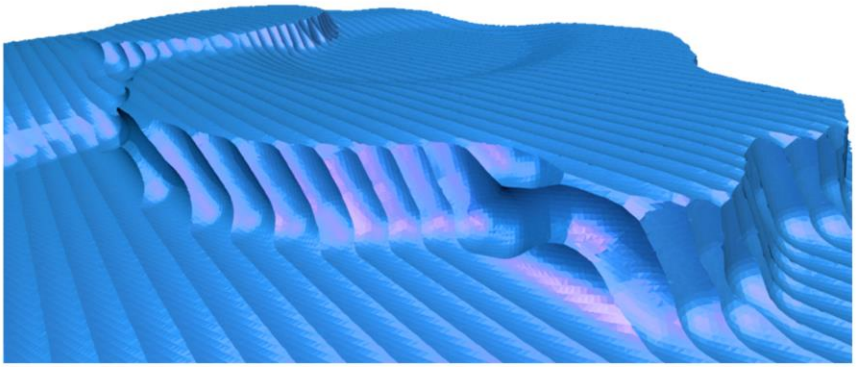


The milling head can be simulated by a sphere.

Example: Milling Simulation

The part that gets milled away can be created as the union of two spheres and a cylinder.

Example: Milling Simulation



An example image of a milling simulation.

Geometric Shapes

- Geometric Shapes
  - Polygonal meshes
  - Constructive solid geometry
- **Scene representations**
  - **Scene graphs**
  - Culling
  - Optimization structures

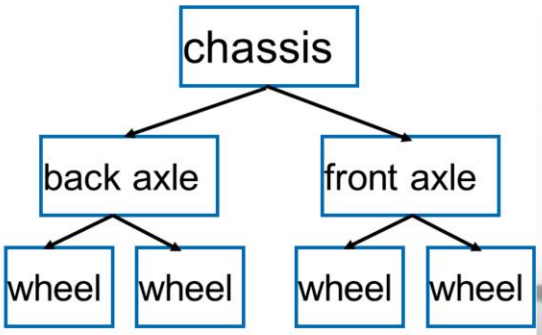
Scene Graphs

- So far: How to handle a single object
  - geometric representation, material, rendering
  - …
- Scenes usually consist of multiple objects
- **Scene graphs** model dependencies between objects

A scene can be seen as one large, complex object or just as a lose collection of individual objects, but this is neither an intuitive, not an effective way to handle a large scene.

Example

- Car consists of multiple parts like chassis, wheels, ...
- Subparts move relative to chassis of car



Humans think of scenes in a hierarchical way: A rooms has tables and chairs, all chairs are the same, each chair has 4 legs etc... If a chair gets moved, all chair-legs get moved as well etc.

Scene Graphs

- Model scene as graph structure, e.g. tree, DAG
- Nodes represent objects, store information
  - appearance, how to render
  - …
- Edges represent relations between objects
  - relative transformations, spatial proximity
  - ...

DAG: directed acyclic graph
With this hierarchical view of the scene, instead of storing absolute transformations for all objects we only need relative ones which is more intuitive and easier to update if a father-object gets moved.

Scene Graphs

- Render the scene:
  - traverse scene graph
  - accumulate relative transformations
  - render objects in nodes
- Optimizations for faster rendering & traversal
  - **culling**
  - **state sorting**

For rendering the absolute transformations are needed again, so while traversing the graph the relative transformations get accumulated.
Now we have the complete information of the scene, so optimizations are easier.

Culling

- Idea: Avoid rendering objects that will not be visible
  - view frustum culling
  - back-face culling
  - occlusion culling
  - …
- Hierarchical culling allows early termination of traversal
- More details on the next slides

State Sorting

- State changes in the rendering pipeline:
  - enabling / disabling lights
  - activating textures, materials
- State changes are expensive
- Find rendering order that minimizes state changes

State changes can be very expensive, so sorting the objects in a way to minimize state changes can speed up rendering times significantly.

State Sorting

- Store state information in each node
- Define sorting function on states
  - Some changes are more expensive than others
  - Cost might depend on hardware !
- Sort objects to minimize state change cost

Switching models can be expensive as well as can switching the shader and even changing uniforms. So for what should be optimize? Sadly, that depends on the system/GPU/driver and can change over time, so sadly no general, absolute truth can be given thats guaranteed to hold for the next years...
nice short tutorial :
http://opengl.j3d.org/tutorials/statesorting.html

Modern Scene Graphs

- Modern scene graphs:
- Perform both culling and state sorting
- Offer Level-of-Detail (LoD) control
- Platform independence
  - Data loading
  - Optimization of data (geometry, …) for rendering
  - Encapsulate rendering commands
- ...

Culling and state sorting both require different scene graph structures. For culling, objects in spatial proximity should be grouped together for efficient view frustum culling. For state sorting, objects with similar states should be grouped together. Thus it might be necessary to build multiple scene graphs, each with a special purpose and combine them. For example by first determining the visible objects and then a suitable rendering order for those objects only.
Nice overview over various properties of scene graphs:
http://www.realityprime.com/articles/scenegraphs-past-present-and-future
Many other features: multi-threading / distributed rendering on multiple GPUs, easy to extend with e.g. physics engines, ...

Modern Scene Graphs

Some (freely) available scene graphs:

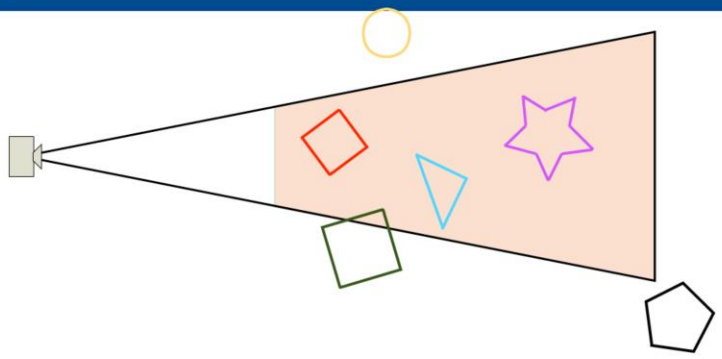
- OpenSceneGraph
- NVidia SceniX
- OpenSG

Nice overview over various properties of scene graphs:
http://www.realityprime.com/articles/scenegraphs-past-present-and-future

Geometric Shapes

- Geometric Shapes
  - Polygonal meshes
  - Constructive solid geometry
- **Scene representations**
  - Scene graphs
  - **Culling**
  - Optimization structures

Culling

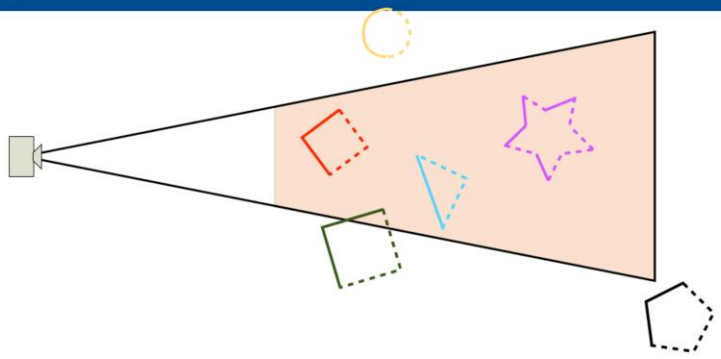


- Back-face culling
- (Hierarchical) View frustum culling
- Portal culling
- Occlusion culling

Culling is basically the process of finding out what can‘t be seen and to don‘t render that at all.

Different culling techniques, ordered by increasing computational cost.

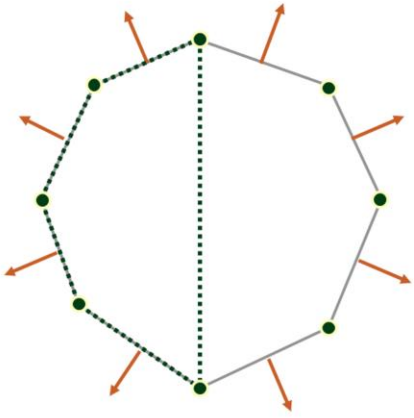
Culling



- **Back-face culling**
- (Hierarchical) View frustum culling
- Portal culling
- Occlusion culling

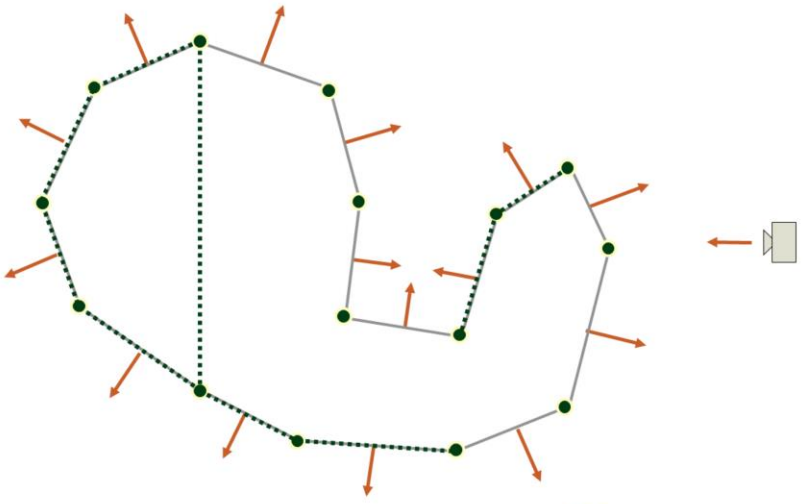
If all objects are closed meshes, the back-faces should never be visible so GPUs have the option to throw those polygons away before rasterization.

Back-Face Culling



Dashed lines are back-faces.

Back-Face Culling



Not all front-faces objects will be visible in the end, so a visibility test is still needed. Also: additional culling techniques can still find other objects that can be thrown away.

Back-Face Culling

44782
back-facing fragments

91030
front-facing fragments

135812
fragments shaded

In this example the fragment shader got called 44782 times for back-faces, these results got overdrawn by front-facing fragments. Depending on the rendering order of the triangles back-facing fragments will get discarded by the z-test, that‘s why the back side of the bunny (left) is incomplete.

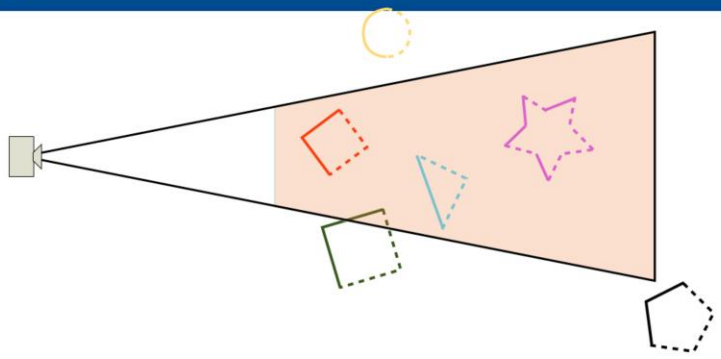
Back-Face Culling

91030 fragments

86650 fragments

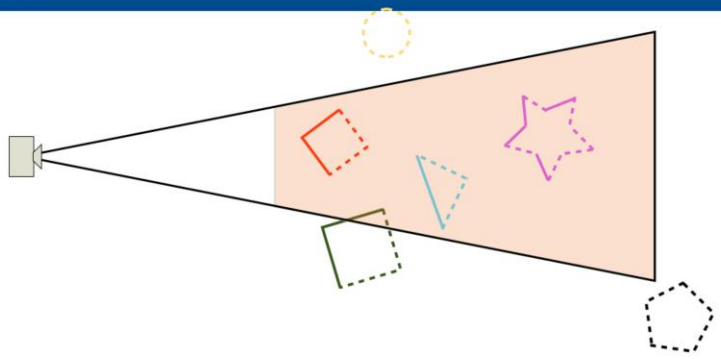
Note that the lighting assumes to get applied to front-facing polygons and does not create useful colors on back-faces (right image). The holes in the right bunny are holes in the model.

Culling



- Back-face culling
- (Hierarchical) View frustum culling
- Portal culling
- Occlusion culling

Culling



- Back-face culling
- **(Hierarchical) View frustum culling**
- Portal culling
- Occlusion culling

Don‘t render anything that‘s outside the field of view (or behind the far-plane!)

View Frustum Culling

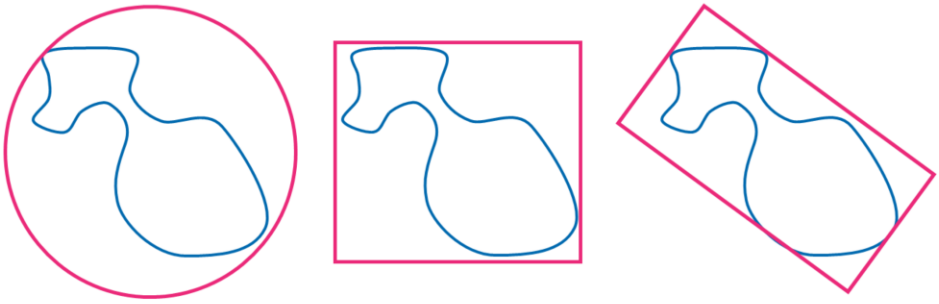
- Objects outside the viewing frustum are invisible
  - Avoid sending them through rendering pipeline!
- Test object against viewing frustum, if ...
  - outside → do not render
  - inside → render
  - intersecting → render

expensive for complex objects

- Test **bounding volume** instead of object itself

Testing the object itself could be too expensive to justify the test, so testing a proxy geometry that‘s simpler than the real object will be much faster. The downside is, that there could be false-positives and objects will get rendered that are in fact not visible but the larger bounding volume intersects slightly with the frustum. The bounding volume should be as small and as simple to test as possible, but also has to be larger than the real object.

Bounding Volumes



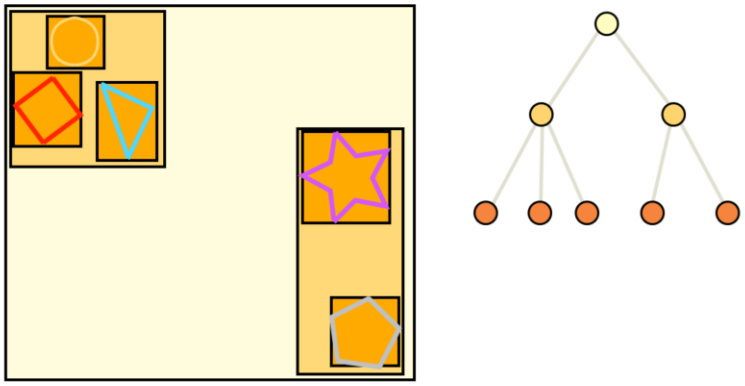
Bounding Sphere

Axis-Aligned Bounding Box (AABB)

Oriented Bounding Box (OBB)

The three most commonly used bounding volumes are bounding spheres, axis-aligned bounding boxes and oriented bounding boxes. They differ in how well they enclose the object, the complexity of their construction and the complexity of their intersection tests. While both the construction and the intersection test for bounding spheres is very simple, they tend to enclose a lot of empty space. AABBs surround their objects more closely, but their intersection tests are more complex. OBBs offer the best fitting volume of the three methods, but also have the most complex intersection test.
See http://www.realtimerendering.com/intersections.html for reference about different bounding volume intersection tests.

Bounding Box Hierarchies



Bounding boxes of bounding boxes in a hierarchy.

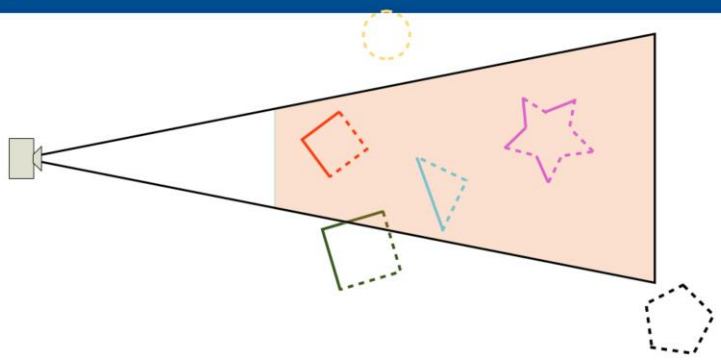
If a higher-level box is completely outside of the view frustum, all children will also be outside. If it is completely inside, so are all children. If however the box intersects the frustum, the child nodes should get tested individually.

Hierarchical View Frustum Culling

- Test node against viewing frustum, if ...
  - completly outside → do not render, stop traversal
  - completly inside → render all leafs, stop traversal
  - intersecting → check next level

- heuristics:
  - test only frustum planes intersecting parent node
  - **frame-to-frame** consistency: remember planes for which object is outside in current frame, test those first in next frame.

Culling

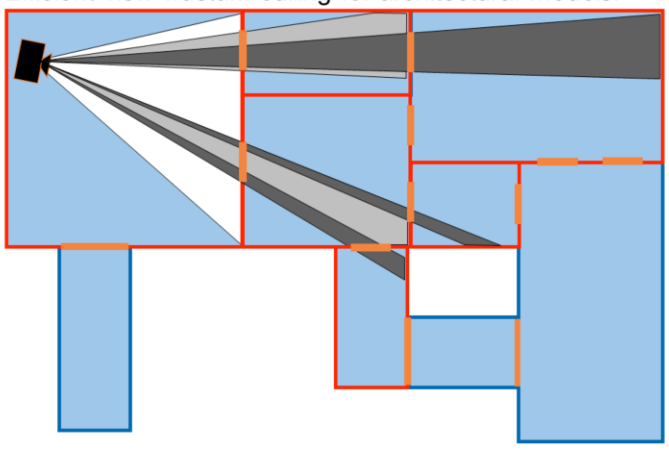


- Back-face culling
- (Hierarchical) View frustum culling
- **Portal culling**
- Occlusion culling

Using high level knowledge of the scene to determine visibility.

Portal Culling

Efficient view frustum culling for architectural models:



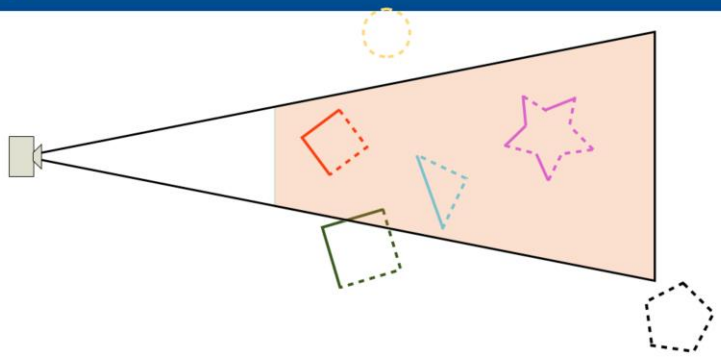
Usually, we can naturally subdivided architectural scenes into cells, e.g. room, corridors, ..., which are connected by portals. Given such a decomposition we can use this structure to accelerate rendering by culling away all cells that are not visible. We start with the cell in which the camera is located and the viewing frustum of the camera. After rendering the objects in this cell (using frustum culling), we detect which portals are visible in the viewing frustum. We use these portals to generate a smaller version of the viewing frustum which we use to recursively render the cells visible through that portal.

Portal Culling

David P. Luebke and Chris Georges. *Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets.* I3D, 1995

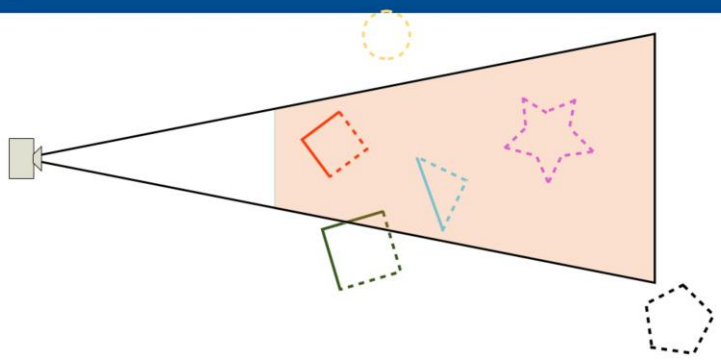
Reflections like mirrors make the portal approach more complicated.

Culling



- Back-face culling
- (Hierarchical) View frustum culling
- Portal culling
- Occlusion culling

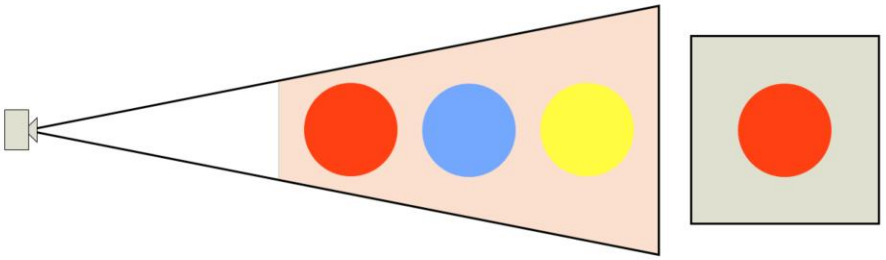
Culling



- Back-face culling
- (Hierarchical) View frustum culling
- Portal culling
- **Occlusion culling**

Finding out which objects are occluded by other objects in the scene.

Occlusion Culling



- Avoid rendering objects that are
  - inside the viewing frustum
  - and occluded by other objects.
- Also known as **Occlusion Query**

Here, only the red ball is visible, the blue and yellow one are not visible despite they being completely in the view frustum and half of them are front-facing. Occlusion queries can help here to find this kind of occluded objects.

Occlusion Culling

- Algorithm outline:
  - for every object *o*
    - if *o* is not occluded
      - render *o*
      - add o to list of occluders
- Rendering order matters! Render **front-to-back** (inverse Painter's Algorithm)!

If the scene gets rendered back-to-front, no occluders will get found. If the scene gets rendered front-to-back all occluded objects will get detected.

Occlusion Query on GPU

- Count fragments generated by rendering object (fragments passing z-test)
- Render object if any / ≥ n fragments generated
- Depending on n, use lower level of detail (LOD)

- **Important**: Occlusion query should be **asynchronous**, CPU & GPU should not have to wait on test result

Whenever a rendering call gets called, the call will return immediately. The CPU does not have to wait for the rendering to get finished, instead the call gets added to a rendering list the GPU will evaluate in order. This also means, that after a simple rendering command gets called (e.g. just a simple bounding box), waiting for that rendering to finish could take some time because the GPU could still be busy with older and more complex calls.
Requesting the result of an occlusion query will force the GPU to catch up with the CPU so this will destroy the asynchronous workflow. That‘s why it‘s better to wait some time before requesting the OQ result - just to be sure the GPU will not force us to wait.

Occlusion Query

- Use low detail geometry (low LOD, bounding volume, ...) to speed up occlusion test

- **Temporal coherence**: Use results of occlusion query in the next frame.
  - Avoids long waiting time for query results.
  - Delay of one frame no problem at high framerates.

Note: in Direct3D the driver is allowed to collect rendering calls of up to three frames, so waiting only one frame for the results may not be enough to be sure the results are available. But the concept of temporal coherence and waiting n frames stays the same.

Occlusion Query: One Frame

- get query results from last frame
  - update visibility list
- draw visible objects
- deactivate writing to color/z-buffer
  - activate query 0
  - ‚render' bounding volume 0
  - activate query 1
  - ‚render' bounding volume 1
  - ...

One frame in a rendering loop using occlusion queries:
First the results of the last frame gets queried to update the list of visible and non-visible objects.
Based on that the visible ones get rendered which also fills the z-buffer
Then the color and z-buffer are set read-only to not change the final frame
For each invisible object an occlusion query gets activated, the object gets rendered (it will go thru the pipeline but not change any framebuffer) and the query gets deactivated (omitted here for brevity).
The results however will not get queried yet!
Additional queries can get used while rendering the visible objects to determine which of those are not visible any more!

Geometric Shapes

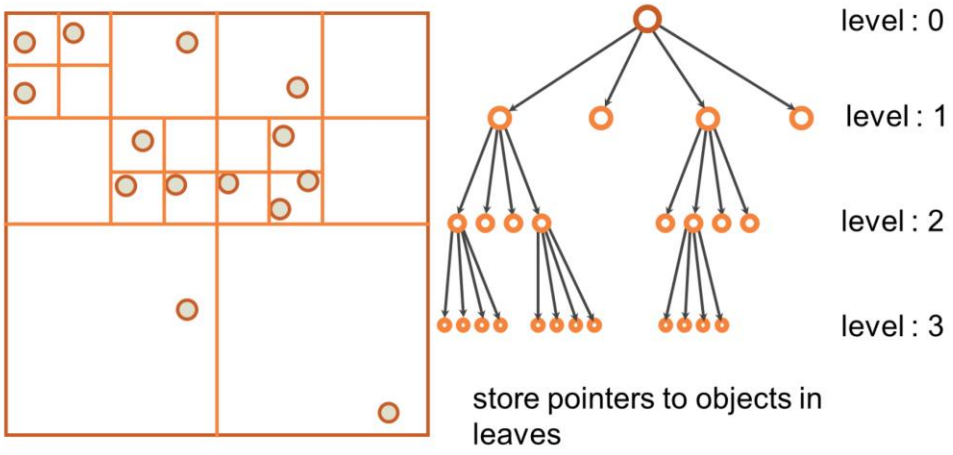
- Geometric Shapes
  - Polygonal meshes
  - Constructive solid geometry
- **Scene representations**
  - Scene graphs
  - Culling
  - **Optimization structures**

Optimization Structures

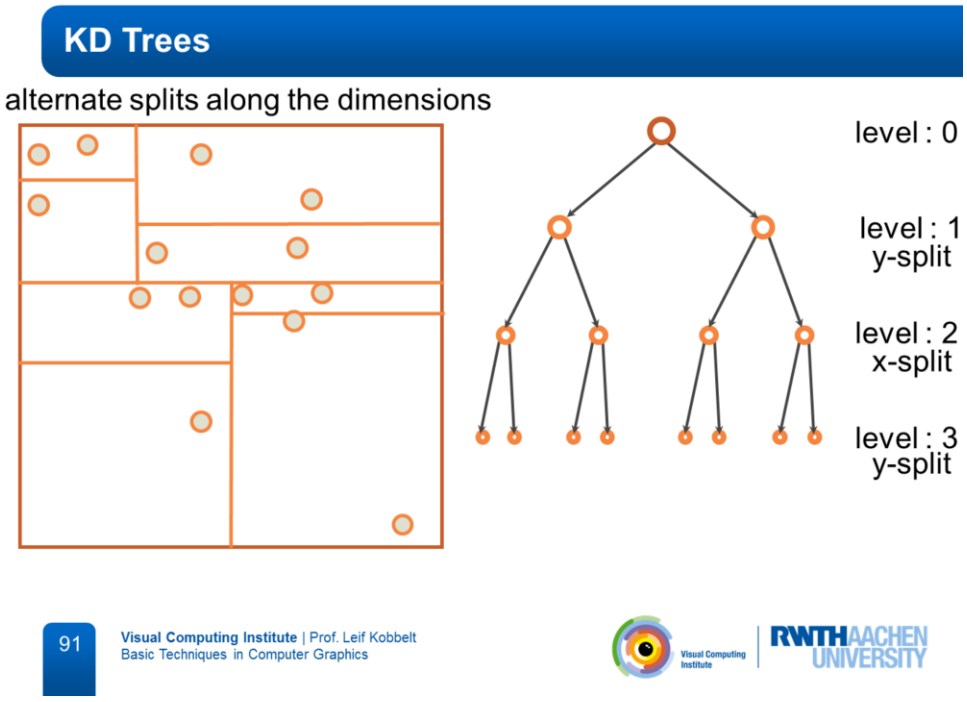
- Idea: Accelerate rendering / application
- Exploit spatial configuration of data / objects
- Choice of structure application dependent
- Common optimization structures:
  - **Octrees**
  - **KD Trees**
  - **BSP**

Often, the rendering or other parts of an application can be speed up very much by using optimization structures. We will present some of the most common optimization structures and motivate them by the example of the Painter‘s Algorithm.

Quadtree / Octree



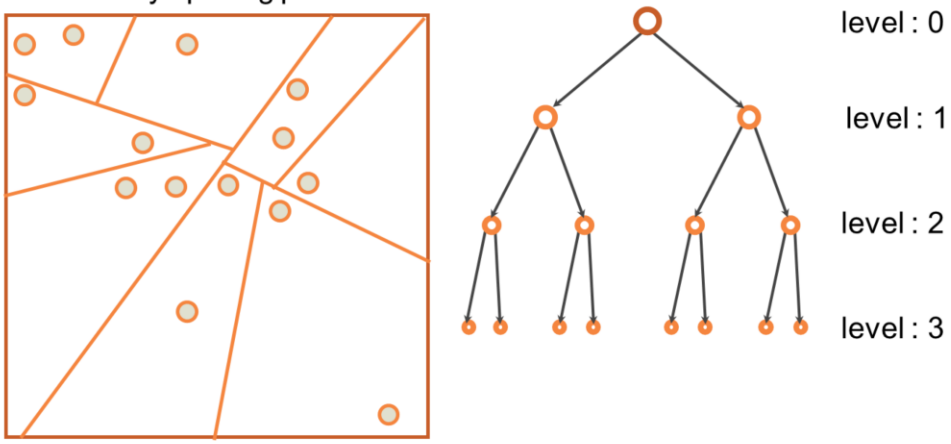
We iteratively subdivide the space into equal sized cells until each cell contains at most n objects or a maximal refinement level is reached. In the leaf cells we then store indices to the enclosed objects. In 2D we divide one cell into 4 cells while in 3D we divide one cell into 8 cells. Therefore, the resulting trees are called Quadtrees and Octrees.



Similar to Octrees, KD Trees also recursively subdivide the space. Octrees are scene independent since the subdivision scheme does not depend on the positions of the objects. KD Trees achieve the subdivision by alternatingly selecting splitting planes along the main directions of the coordinate system. These planes split the set of objects into two subsets, thus the resulting trees are strictly binary. Ideally the resulting trees are as balanced as possible. To achieve this, the splitting plane is usually chosen in a way that splits the set of objects into two subsets of (nearly) the same size. The resulting subdivision schemes are thus scene dependent since the placement of the splitting planes depends on the positions of the objects.

Binary Space Partitioning

arbitrary splitting planes

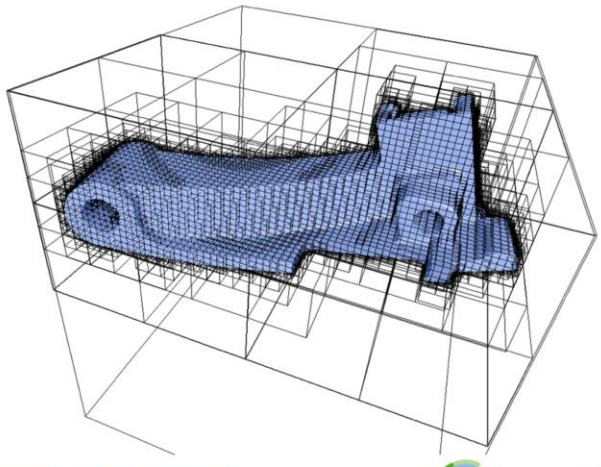


Binary Space Partitioning (BSP) further generalizes KD Trees by allowing the splitting planes to be placed arbitrarily. The resulting BSP Trees are again strictly binary trees. Similar to KD Trees, the splitting planes for BSP Trees should be chosen in a way such that the resulting tree is as balanced as possible.

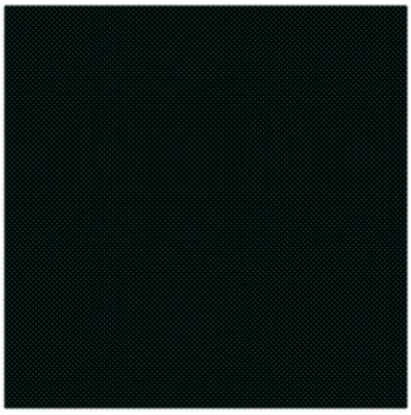
Adaptive Refinement

- Voxel grid $\rightarrow$ $O(h^{-3})$
- Adaptive octree
  - Three color octree $\rightarrow$ $O(h^{-2})$
  - Adaptively sampled distance fields $\rightarrow$ $O(h^{-1})$
- kD-Tree
- Binary space partition
  - Partition the space along arbitrary planes
  - Piecewise **linear** C^{-1} approximation
  - Align cells to surface

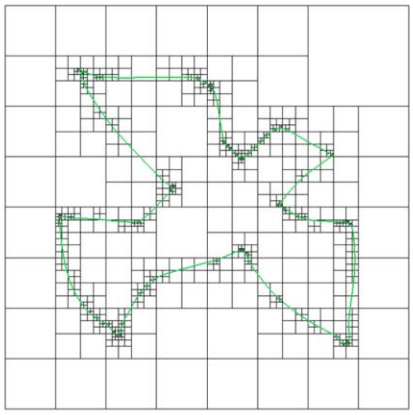
Three Color Octree



Adaptively Sampled Distance Fields



1048576 cells



1294 cells

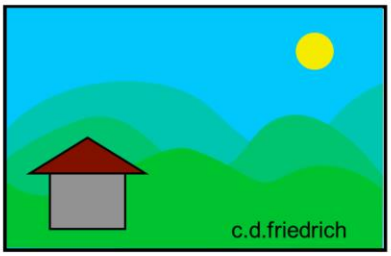
Comparison

- Octree
  - \+ easy & fast subdivision
  - – no balancing → slow traversal
- KD Tree
  - \+ better balancing than Octrees
  - \+ fast traversal
  - – tree construction more complex
- BSP Tree
  - \+ best balancing due to arbitrary splitting planes
  - \+ fast traversal
  - – most complex tree construction

Painter's Algorithm

Remember: Painter's Algorithm:

- Paint polygons from back to front
- Sorting is needed each frame...

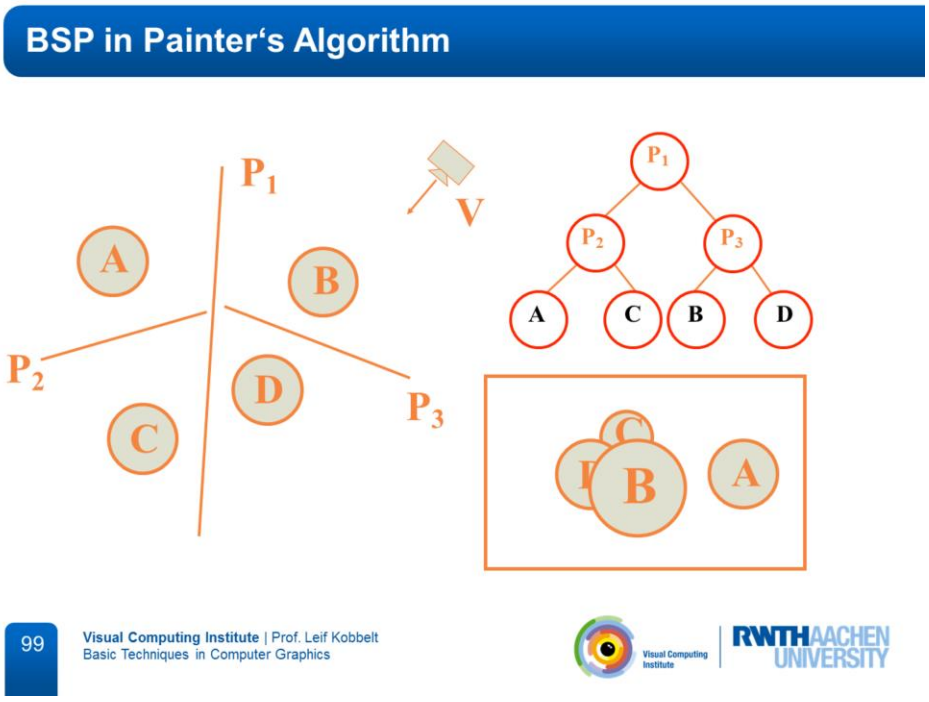


We already talked about the painters algorithm where sorting from back to front was needed. The following optimization structures can be used to speed this sorting up as it has to be done for every frame from different positions in space.
If a z-buffer is used, sorting can also be useful by sorting everything from front-to-back to reduce overdraw and enable efficient occlusion queries.
Rendering transparent objects also needs sorting....

Space Partitioning

- Sorting becomes dominant when rendering a scene from *many* viewpoints

- Pre-sort the scene to exploit clustering
  → *tree traversal instead of sorting*
  - Octree (scene independent)
  - KD Tree (scene dependent)
  - Binary Space Partitioning (scene dependent)



We can use BSP Trees (or Octrees or KD Trees) to determine the ordering in which the objects are drawn by Painter's Algorithm. Given a BSP tree and a camera position, we start at the tree traversal at the root node and do a depth first search. For any splitting plane, we first visit the subtree corresponding to the half-space in which the camera does not lie. If we visit a leaf node we draw its objects onto the scene. Similar traversal schemes can be defined for KD Trees and Octrees.
So rendering a sorted scene becomes a traversal of the tree and so reduces the complexity from O(n log(n)) to O(n)!