第十四章 三维场景的组织与优化

王长波 教授

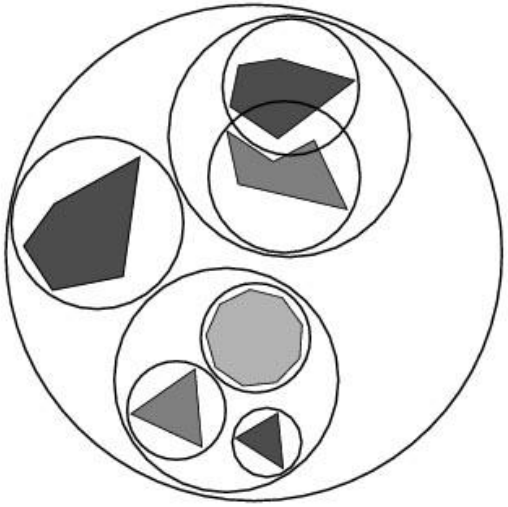
三维场景的组织与管理

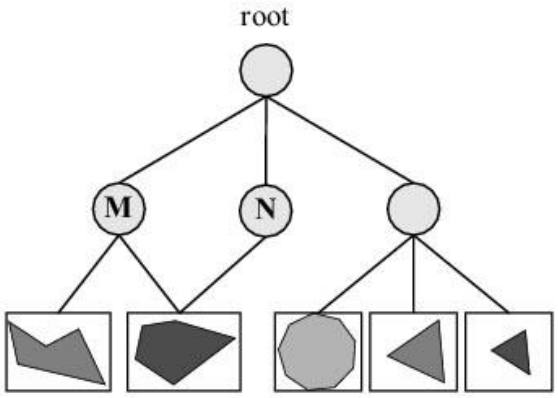
基于场景图的表达与管理

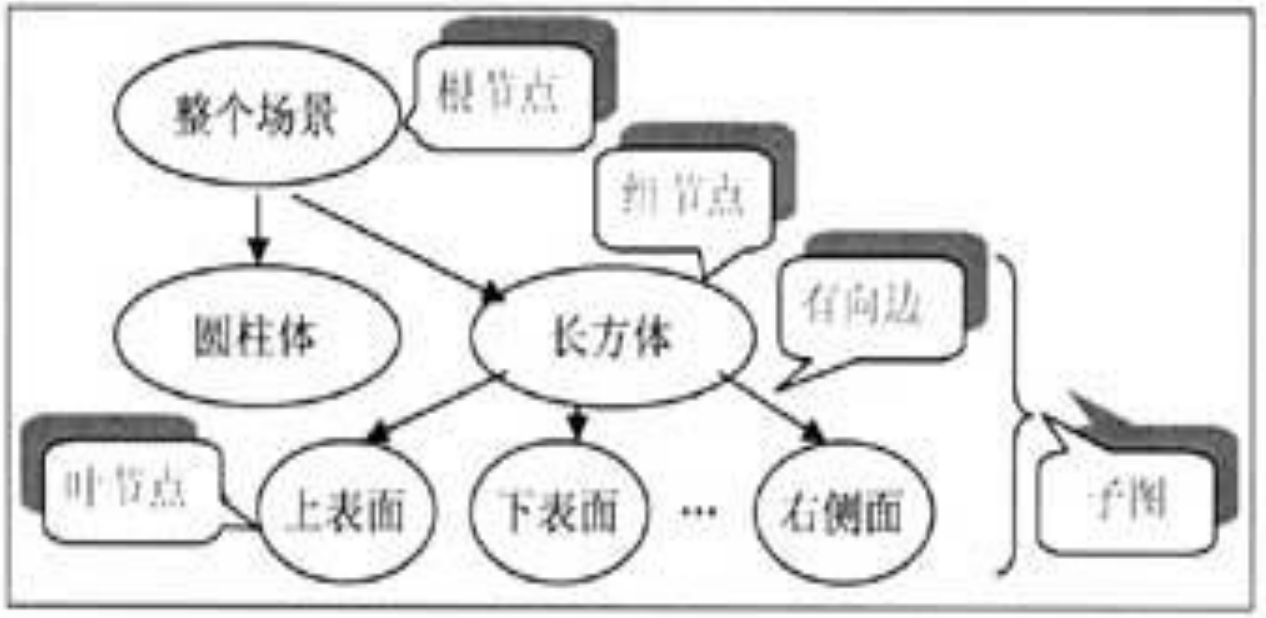
- 场景图是一种将场景中的各种数据以图的形式组织在一起的场景数据管理方式。
- 它是一个K-树,场景图中的每一个节点都是数据的存储结构,父结点会影响子结点 。
- 每个节点信息包括：（1）场景的组织结构信息，如父节点或子节点的句柄；（2）支持绘制流程的各种信息，如节点在当前帧中是否绘制的标识，节点的包围体等；（3）描述表现自身所需的各类特征属性信息，如位置坐标、变换矩阵以及颜色、材质等。

Scene graphs

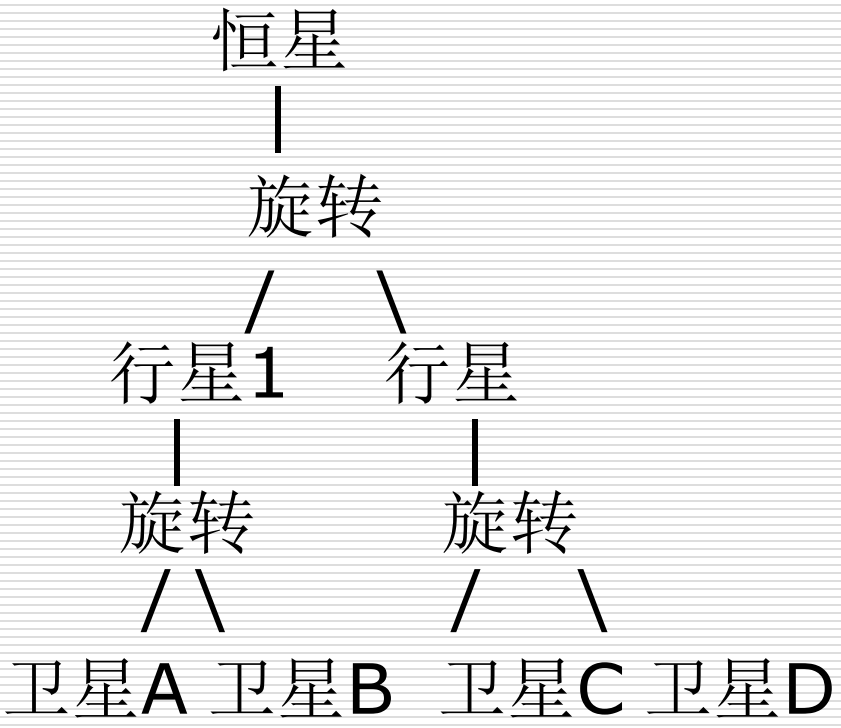
- BVH is the data structure that is used most often
  - Simple to understand
  - Simple code
- However, it stores just geometry
  - Rendering is more than geometry
- The scene graph is an extended BVH with:
  - Lights
  - Textures
  - Transforms
  - And more





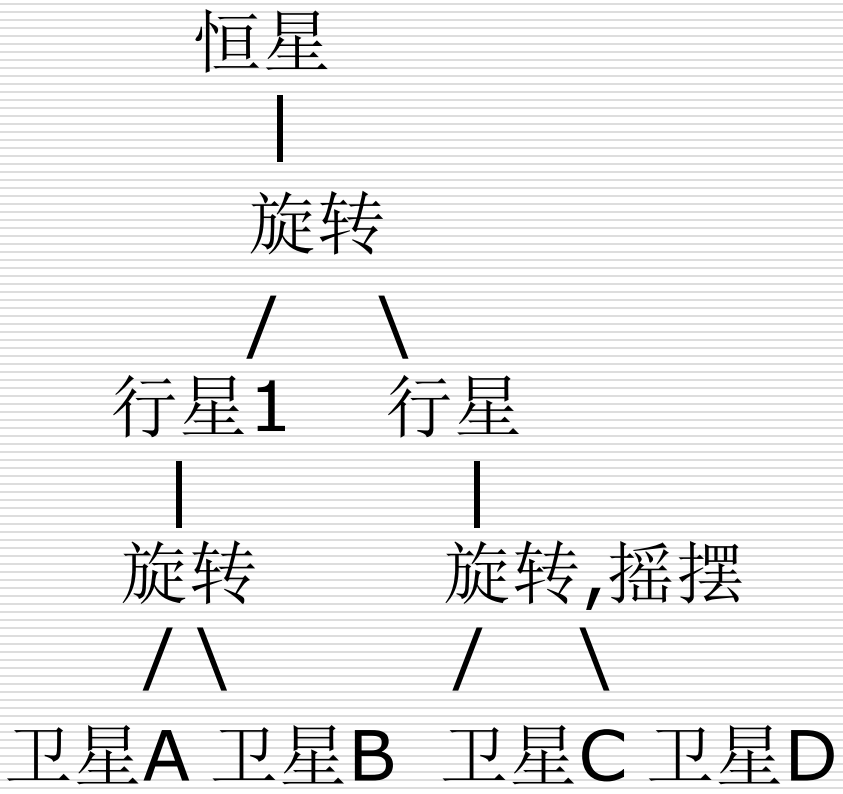

整个场景
根节点
组节点
圆柱体
长方体
有向边
叶节点
上表面
下表面
…
右侧面
子图

例:

- 模拟一个星系
  - 

- ·渲染恒星
- ·保存当前矩阵
  - 应用第一个旋转
  - 渲染行星1
  - 保存当前矩阵
    - 运用第二个旋转
    - 渲染卫星A
    - 渲染卫星B
  - 恢复我们保存的矩阵
  - 渲染行星2
  - 保存当前矩阵
    - ·运用行星2的旋转
    - 渲染卫星C
    - 渲染卫星D
  - 恢复我们保存的矩阵
- ·恢复我们保存的矩阵

行星A摇晃



- 渲染恒星
- 保存当前矩阵
  - 应用旋转
  - 保存当前矩阵
    - 运用摇晃
      - 渲染行星A
      - 保存当前矩阵
        - 运用旋转
        - 渲染卫星A
        - 渲染卫星B
        - 恢复矩阵
      - 恢复矩阵
  - 恢复矩阵
- 恢复矩阵
- 运用旋转
- 渲染行星2
  - 保存当前矩阵
    - 运用行星2的旋转
    - 渲染卫星C
    - 渲染卫星D
- 恢复我们保存的矩阵
- 恢复我们保存的矩阵

- 场景绘制过程
  - 根据游戏的需要,更新场景中的部分结构,从下到上;
  - 场景图的剔除绘制过程:
  - 要修改一个行星的位置,只需修改星结点的属性,不更改任何子节点的属性.

- 场景图一般包含下列节点：
  - 几何节点
  - 变换节点
    - 平移，旋转，缩放
  - 开关节点

    通过当前状态对子节点进行选择的节点

基于绘制状态的场景管理

- 把场景中的物体按照绘制状态分类，对相同状态的物体设置一次状态；
- 状态切换是一个比较耗时的运算；
- 绘制状态包括：
  - 纹理映射的参数设置
  - 材质参数，包括泛光、漫射光等
  - 各类其他渲染模式：多边形插值、融合等
- 半条命

- 建立状态树
  - 按状态集进行排序
  - 遵照“尽量使状态转换最少”
- 按深度优先遍历状态树，依次绘制

基于景物包围体的场景组织

- 由于可见性检测、求交、碰撞等都可归结为空间关系的计算。
- 对每个物体建立包围体；
- 对场景建立包围体层次树；
- 快速判断一个点是否在物体的凸包围体中。

- 常用的场景物体包围体
  - 包围球
  - AABB包围盒
  - OBB包围盒
  - 平行六边包围体
  - K对平行六面包围体

优化场景绘制的几何剖分技术

- BSP树
  - 应用于深度排序，碰撞检测，绘制，节点裁减和可见性判断，加速三维场景的漫游；
  - 空间中的任意平面把空间分成两部分：一份为二地空间剖分方法；
  - 一直递归下去，结束的条件：
    - 空间中没有物体了；
    - 剖分的深度达到了指定的数值就停下

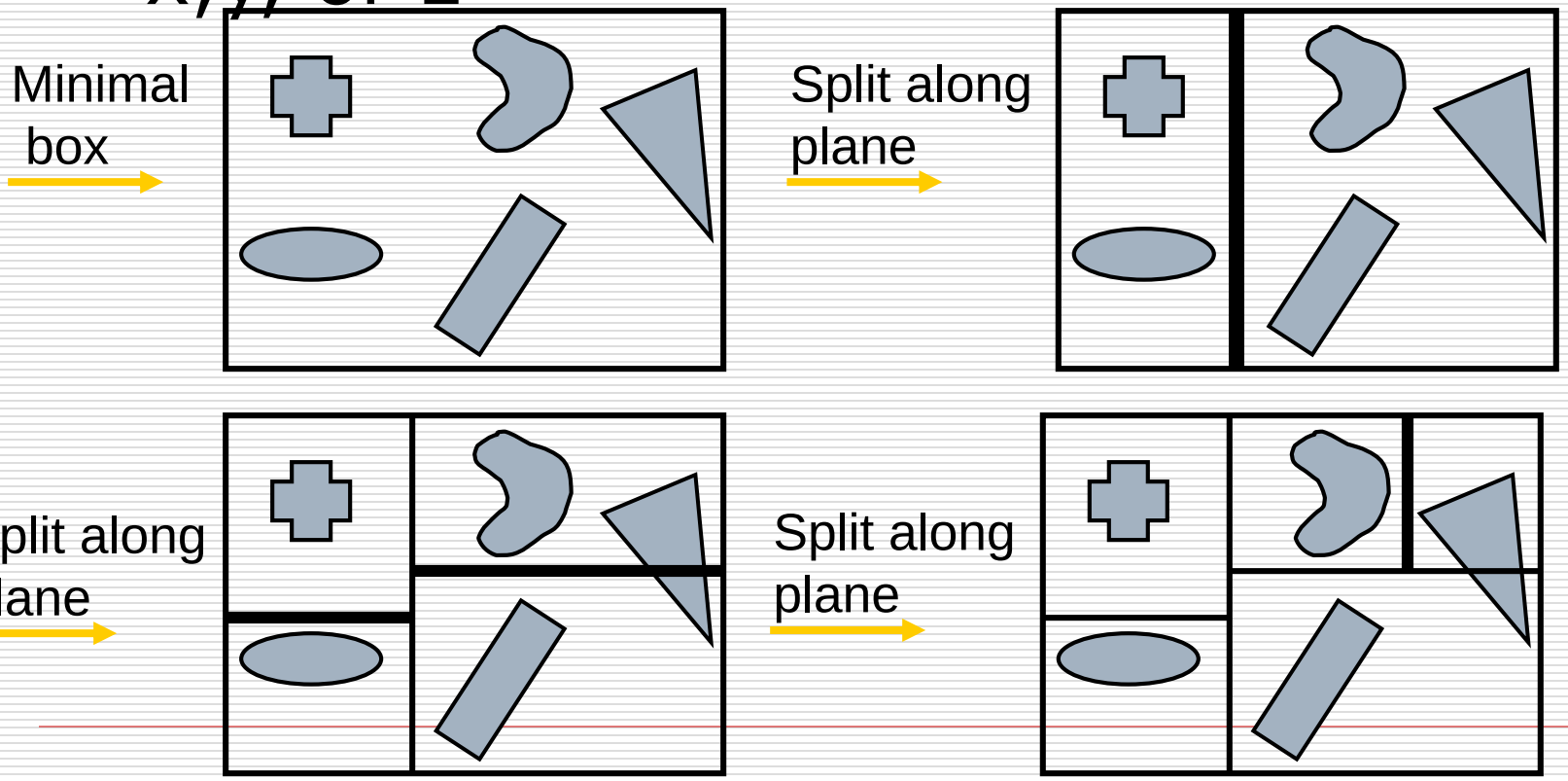
- BSP树的建立
  - 先对物体建立包围体结构，然后以包围体为单位建立场景的BSP树；
  - 剖分方式：
    - 均匀剖分：适于场景中物体分布均匀
    - 平行坐标轴剖分：室内游戏
    - 选取场景中面积最大和遮挡物体最多的面：判断物体与剖分面的关系稍负责

Binary Space Partitioning (BSP) Trees

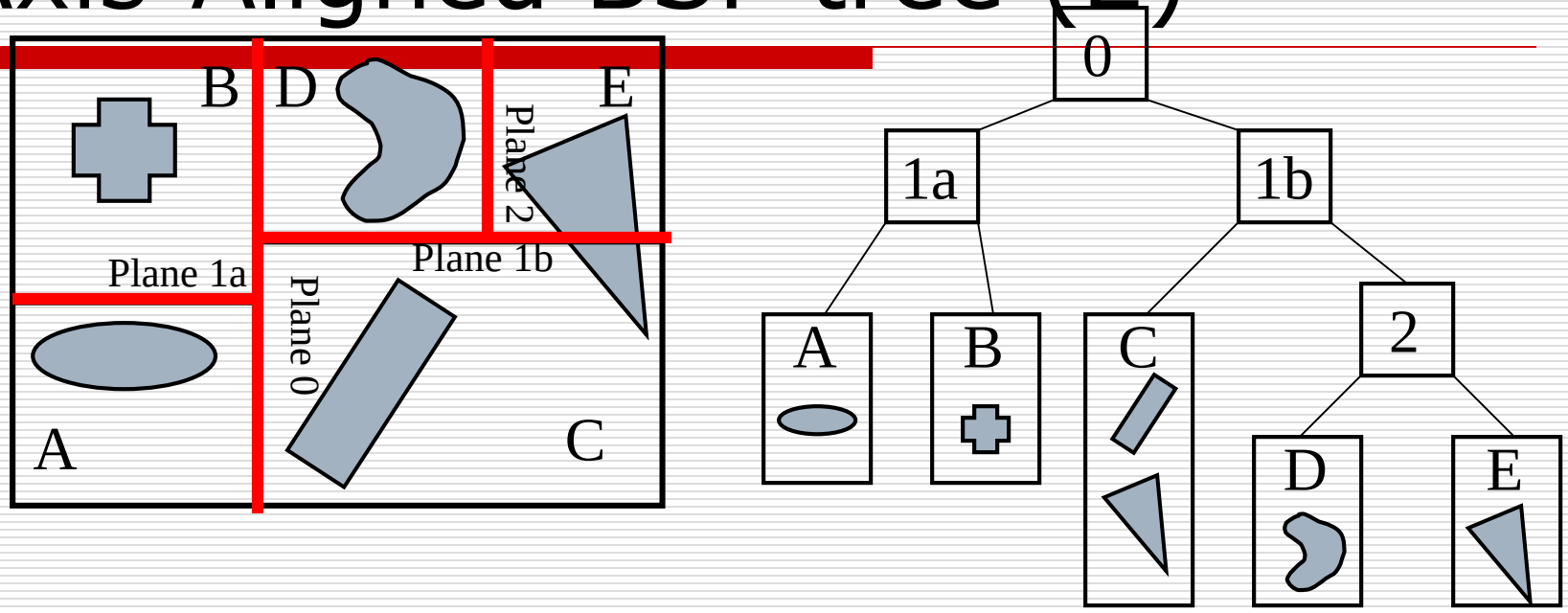
- Two different types:
  - Axis-aligned
  - Polygon-aligned (treated in other graphics course, and in handout text)
- General idea:
  - Divide space with a plane
  - Sort geometry into the space it belongs
  - Done recursively
- If traversed in a certain way, we can get the geometry sorted along an axis
  - Exact for polygon-aligned
  - Approximately for axis-aligned

Axis-Aligned BSP tree (1)

- Can only make a splitting plane along x,y, or z



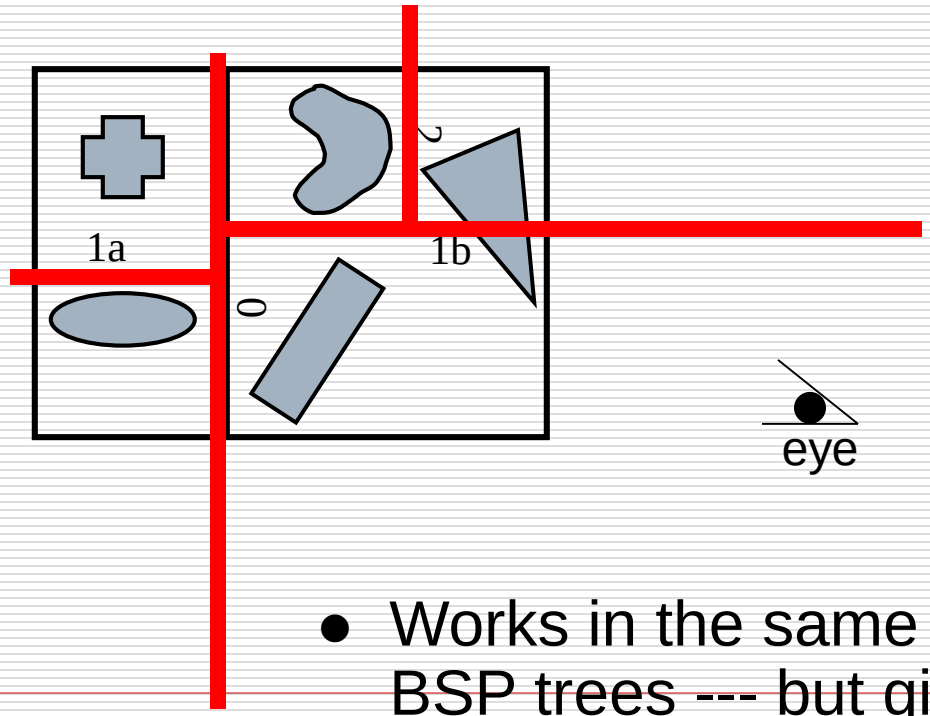
Axis-Aligned BSP tree (2)

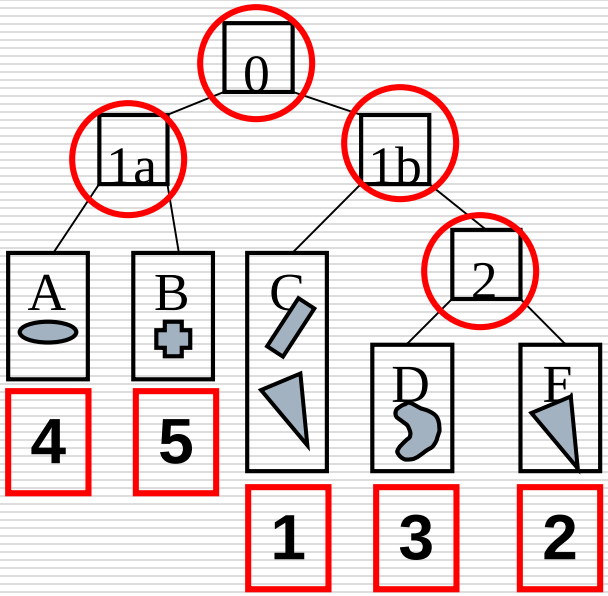


- Each internal node holds a divider plane
- Leaves hold geometry
- Differences compared to BVH
  - Encloses entire space and provides sorting
  - The BV hierarchy can be constructed in any way (no sort)
  - BVHs can use any desirable type of BV

Axis-aligned BSP tree Rough sorting

- Test the planes against the point of view
- Test recursively from root
- Continue on the ”hither” side to sort front to back

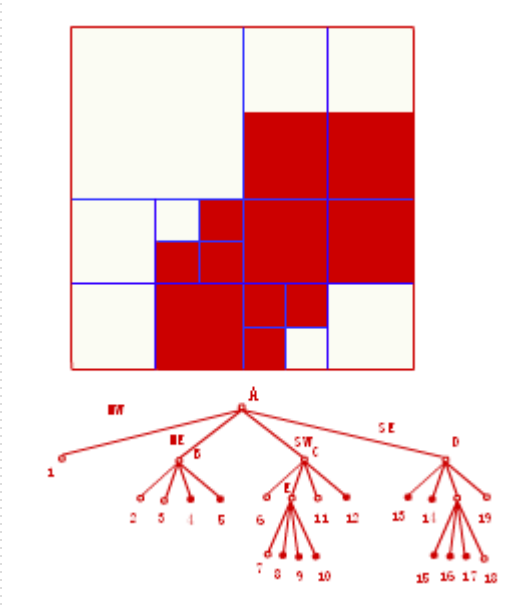




- Works in the same way for polygon-aligned BSP trees --- but gives exact sorting

- BSP树的遍历：
  - 深度优先
  - 广度优先
- BSP树局限性：
  - 不太适合动态场景
  - 构造时间长

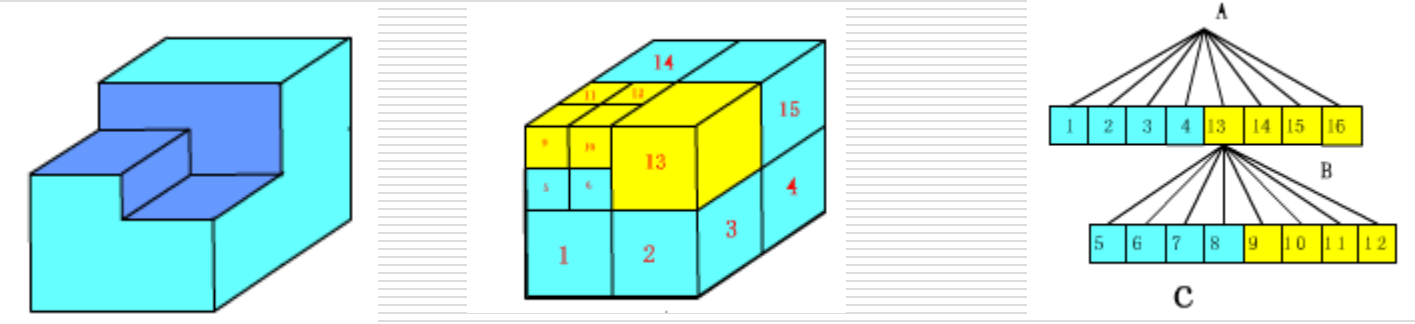
- 四叉树
  - 常用于地形绘制；
  - 以包围四边形逼近场景，然后迭代地一分为四。
    - 均匀剖分：适于场景中物体分布均匀
    - 平行坐标轴剖分：室内游戏
    - 选取场景中面积最大和遮挡物体最多的面：判断物体与剖分面的关系稍负责
  - 最大的优点：层次剔除



八叉树

- 长方体递归地剖分为八个长方体；
- 构建时间比BSP树短，容易使用。

- 常用于视域裁剪、碰撞检测。



比较

技术名称	适用场景	构建复杂度	实用性
二叉树	尺寸不大的室内场景	复杂	大部分游戏引擎
四叉树	室外地形	一般	仅用于地形绘制
八叉树	大规模室内空间场景	一般	复杂游戏引擎
均匀八叉树	分布均匀的三维场景	简单	少量三维游戏引擎

游戏场景的几何优化

- Spatial data structures are used to speed up rendering and different queries

- Why more speed?
- Graphics hardware 2x faster in 6 months!
- Wait… then it will be fast enough!
- NOT!
- We will never be satisfied
  - Screen resolution: 3000x1500
  - Realism: global illumination
  - Geometrical complexity: no upper limit!

- 层次细节技术（LOD）
  - 在不影响画面视觉效果情况下，逐步简化景物表面的细节来减少场景的几何复杂性；
  - 四类：
    - 简单取舍型
      - 设置阈值，小于则不绘
    - 平滑过渡型
      - 大于上限，正常绘制；小于，不绘；中间，线性过渡
    - 静态LOD型
      - 预计算同一物体的多个不同精度的版本，根据不同距离切换使用不同的物体
      - 跳跃：增加雾化效果

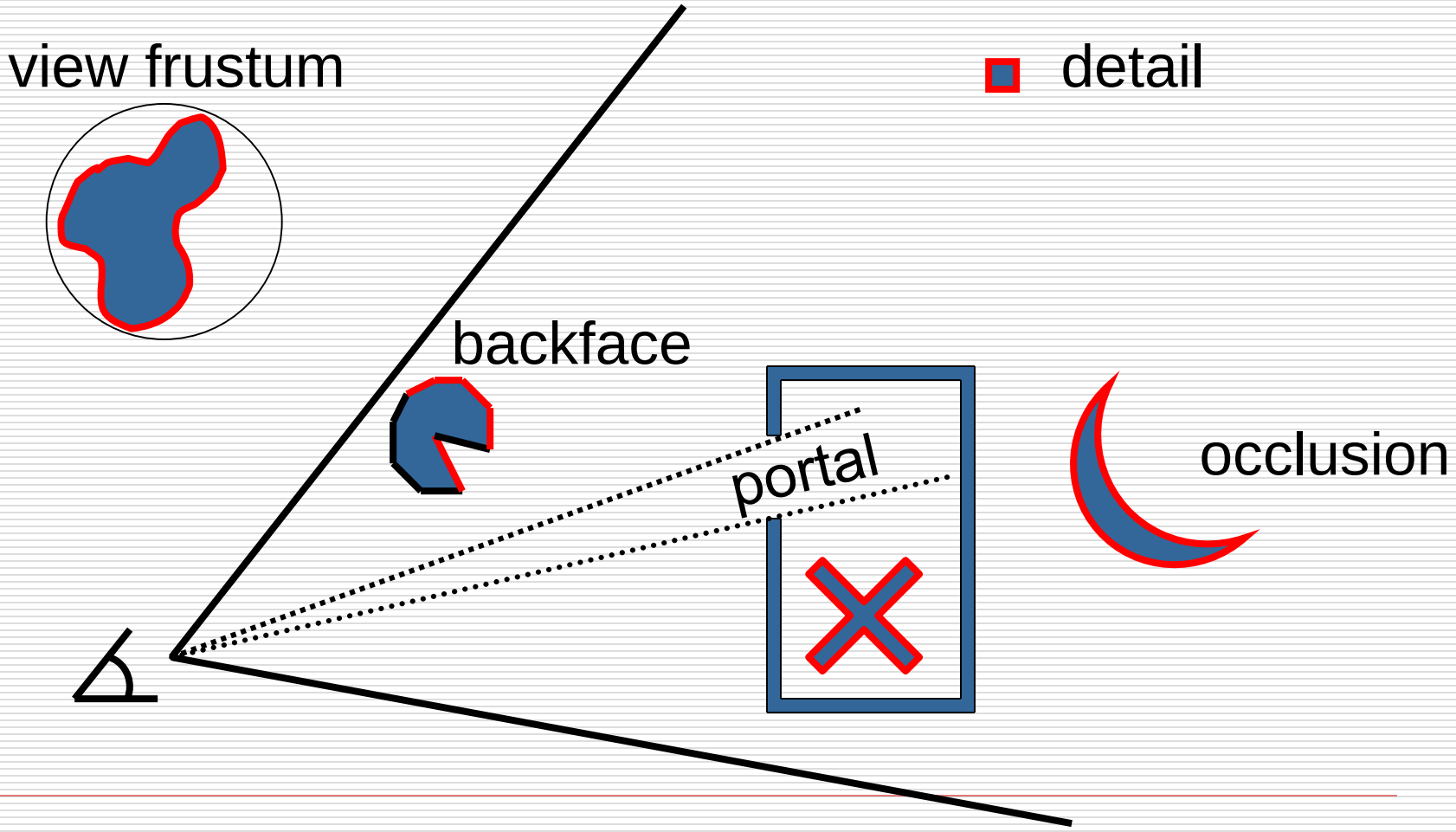
- 动态LOD型
  - 在场景漫游过程中动态地根据相机的位置和物体的重要性动态简化网格；
  - 简化算法：
    - 基于顶点的删除
    - 基于边的删除：定义代价函数，删除代价最小的边
    - 基于面的删除
- 渐进网格和连续多分辨拉绘制

- Use different levels of detail at different distances from the viewer
- More triangles closer to the viewer

三维场景的快速可见性判断与消隐

- 可见型判断
  - 物体层的算法：场景聚类技术
    - 决定相对于视点面的前后排列
    - 二叉树和八叉树，入口技术，潜在可见集，单物体分块
  - 顶点层算法：顶点剔除算法
    - 背面剔除
    - 视域剔除
    - 裁剪面剔除
  - 象素层的算法：图像空间排序
    - 深度缓冲消隐
    - 层次缓冲消隐
    - 时空连贯性

Different culling techniques



- 基于入口技术的可见性判断
  - 适用于室内游戏
  - 根据单元与单元之间的邻接图，通过深度遍历建立单元与单元的入口序列，从而得到单元对单元的可见性集合。
  - 入口技术的优点：
    - 方便定义场景：可方便地动态创建和修改场景
    - 快速绘制

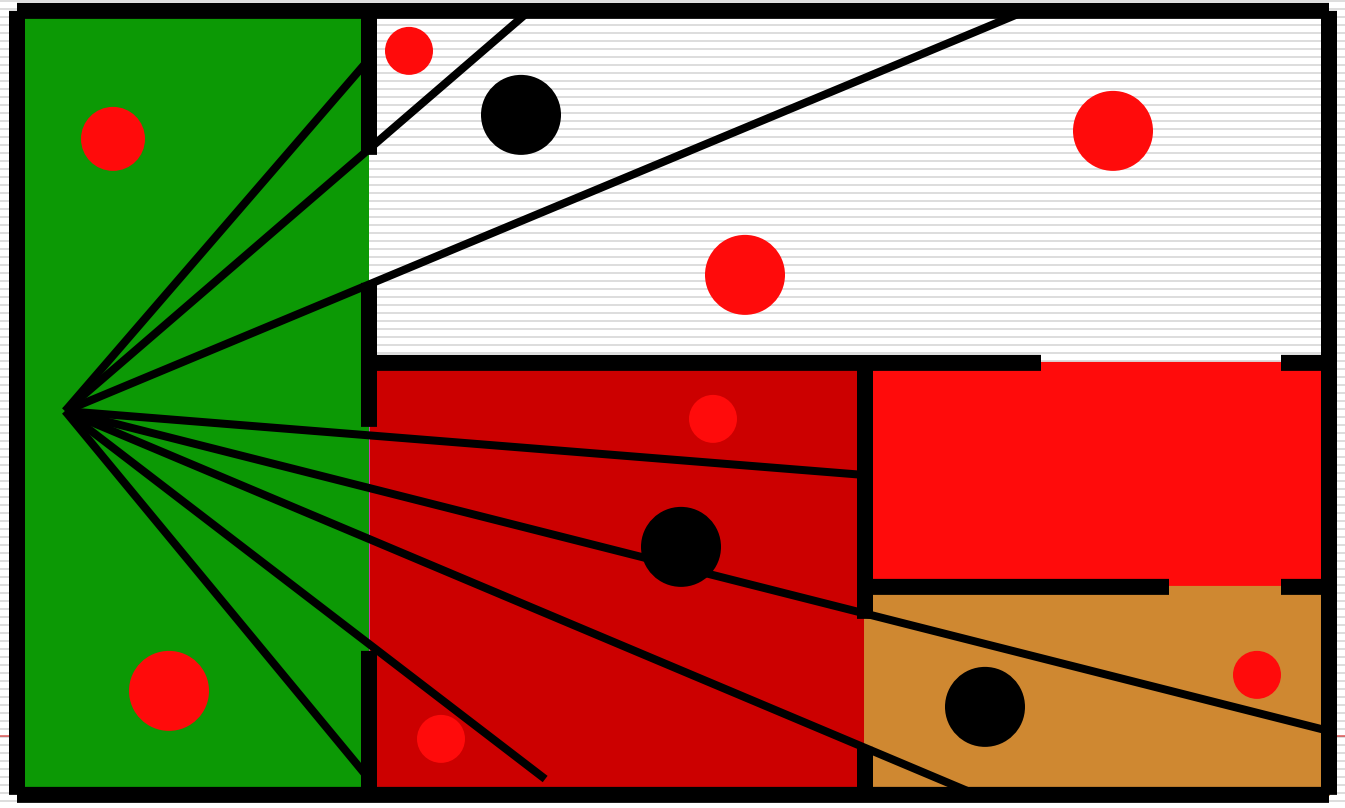
Portal Culling

Images courtesy of David P. Luebke and Chris Georges

- Average: culled 20-50% of the polys in view
- Speedup: from slightly better to 10 times

Portal culling example

- In a building from above
- Circles are objects to be rendered



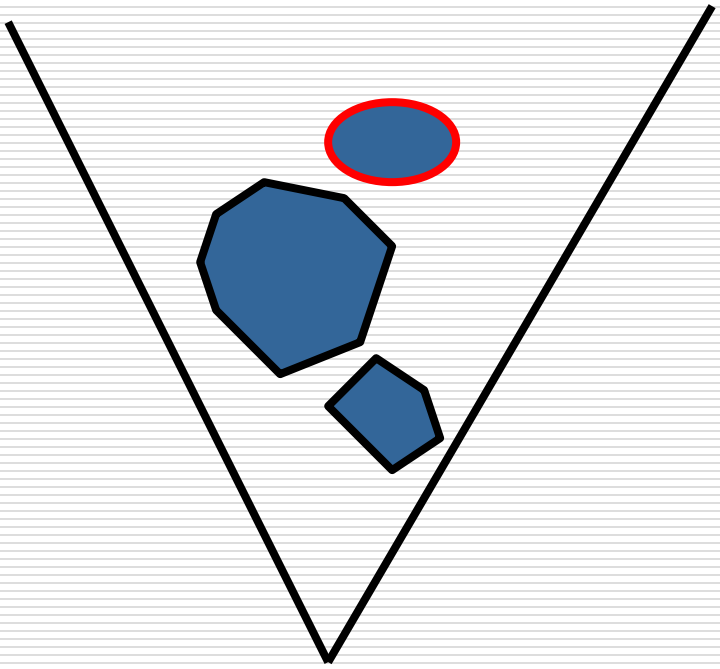
Portal Culling Algorithm (1)

- Divide into cells with portals (build graph)
- For each frame:
  - Locate cell of viewer and init 2D AABB to whole screen
  - * Render current cell with View Frustum culling w.r.t. AABB
  - Traverse to closest cells (through portals)
  - Intersection of AABB & AABB of traversed portal
  - Goto *

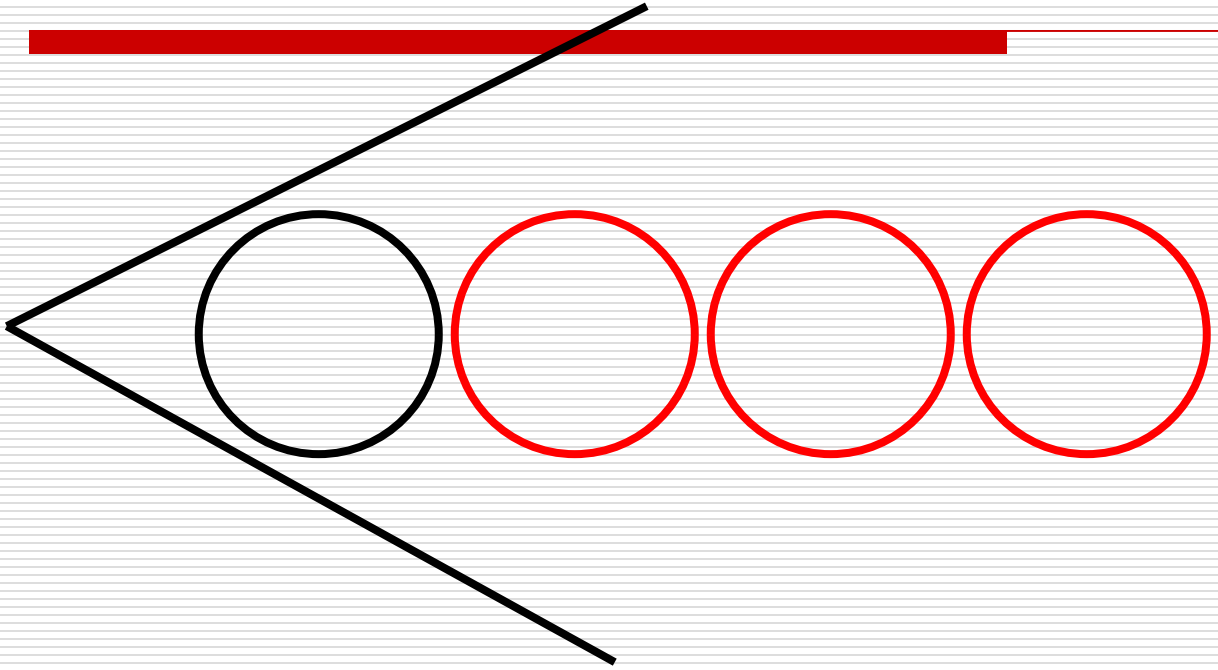
- 遮挡面剔除技术
  - 基于阴影体的遮挡剔除
    - 选择遮挡体，生成一个阴影体：尺寸最大的多边形或多个三角形组成的一个大的凸包；
    - 利用包围盒测试与遮挡体的关系
    - 基于视点的遮挡剔除
    - 对多个遮挡体，按重要性排序

Occlusion Culling

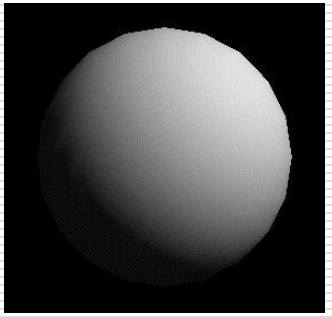
- Main idea: Objects that lies completely “behind” another set of objects can be culled
- Hard problem to solve efficiently
- Lots of research in this area



Example



final image



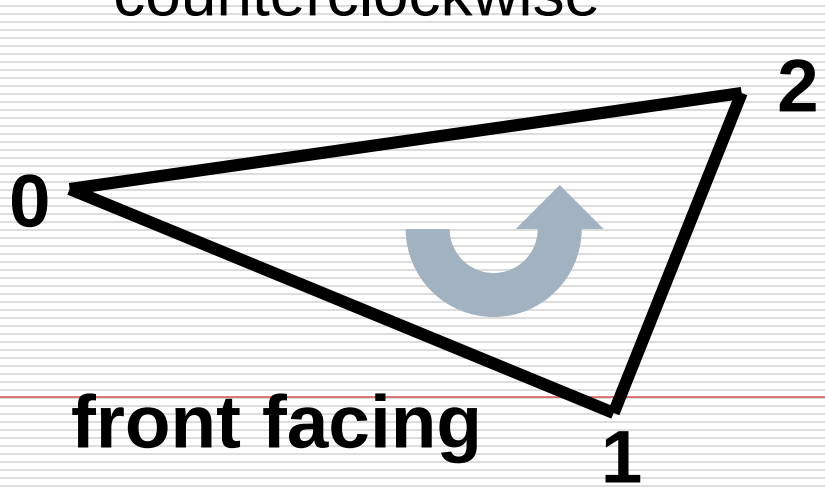
- Note that “Portal Culling” is type of occlusion culling

Backface Culling

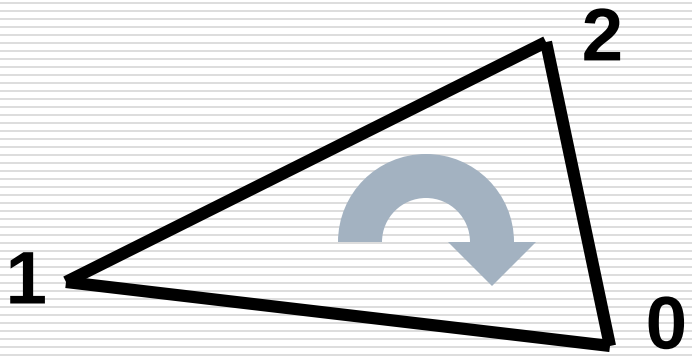
- Simple technique to discard polygons that faces away from the viewer
- Can be used for:
  - closed surface (example: sphere)
  - or whenever we know that the backfaces never should be seen (example: walls in a room)
- Two methods (screen space, eye space)
- Which stages benefits? Rasterizer, but also Geometry (where test is done)

Backface culling (cont'd)

- Often implemented for you in the API
- OpenGL: `glCullFace(GL_BACK);`
- D3D: **`SetRenderState(D3DRS_CULLMODE, TRUE);`**
- How to determine what faces away?
- First, must have consistently oriented polygons, e.g., counterclockwise



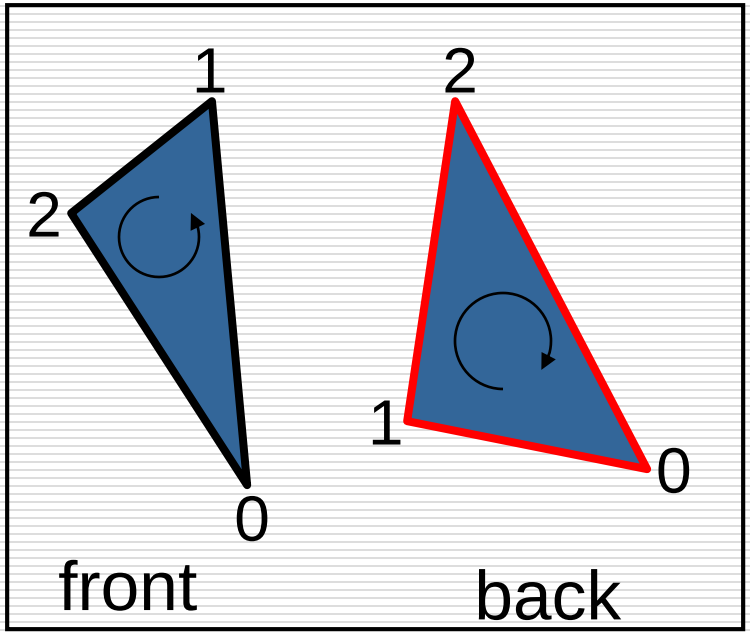
front facing



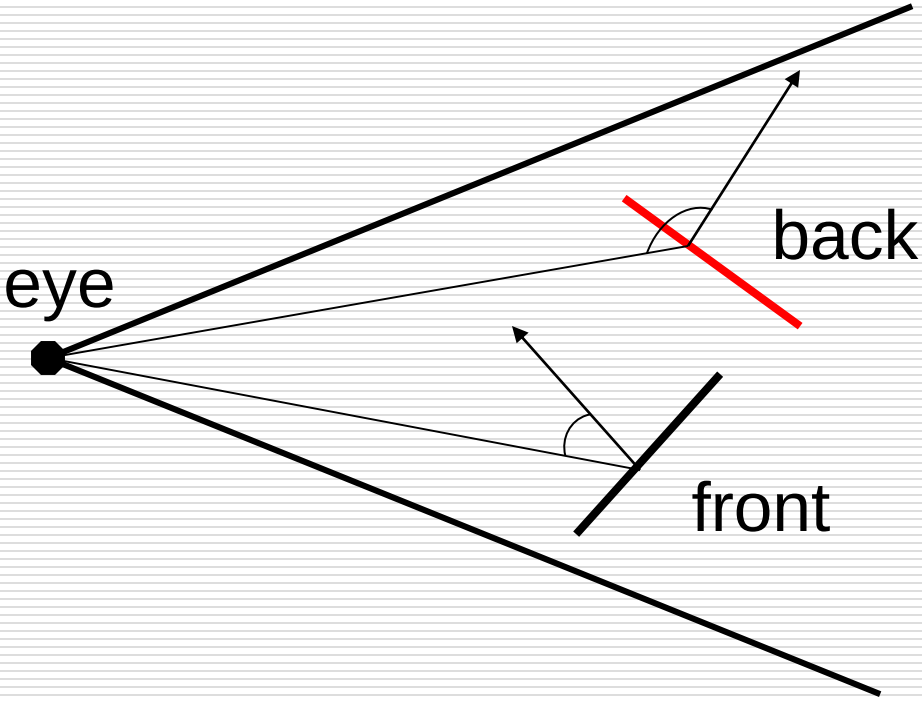
back facing

How to cull backfaces

- Two ways in different spaces:



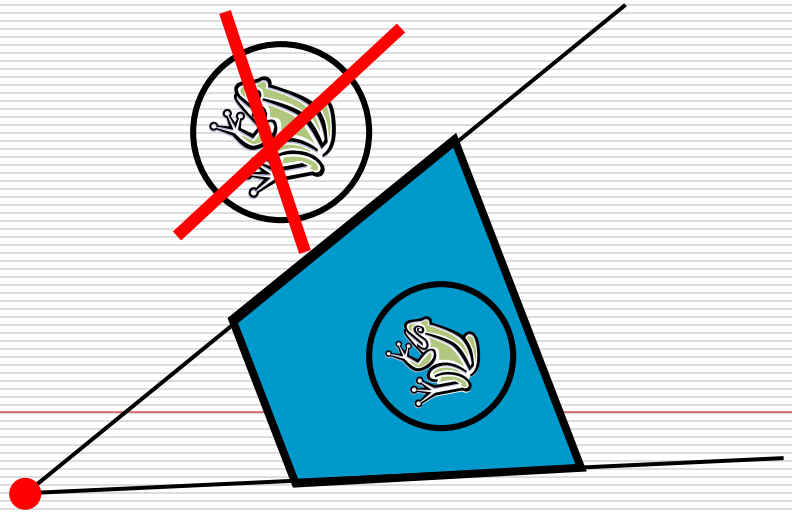
screen space



eye space

View-Frustum Culling

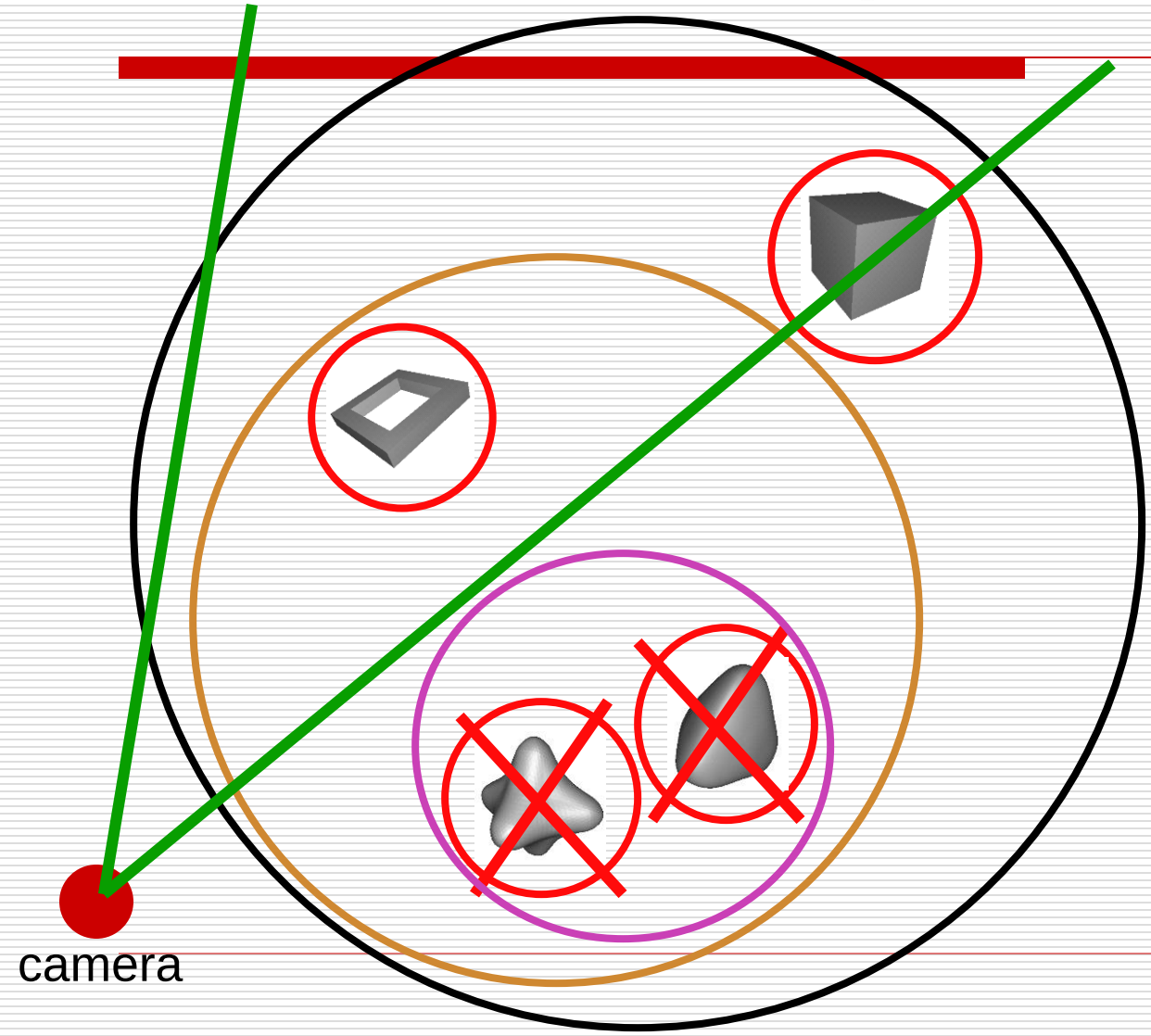
- Bound every “natural” group of primitives by a simple volume (e.g., sphere, box)
- If a bounding volume (BV) is outside the view frustum, then the entire contents of that BV is also outside (not visible)

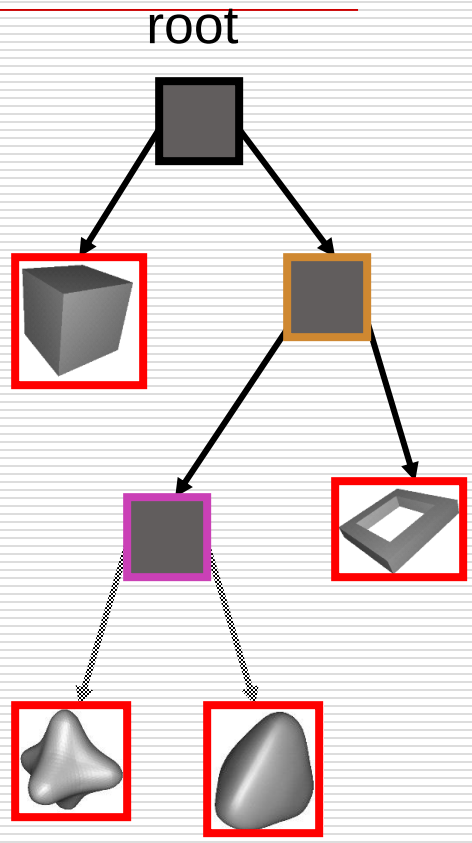


Can we accelerate view frustum culling further?

- Do what we always do in graphics…
- Use a hierarchical approach, e.g., a spatial data structure (BVH, BSP, scene graph)
- Which stages benefits?
  - Geometry and Rasterizer
  - Bus between CPU and Geometry

Example of Hierarchical View Frustum Culling



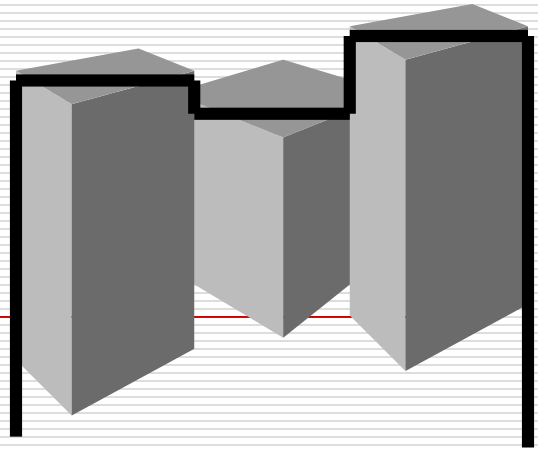


Occlusion culling algorithm

Use some kind of occlusion representation $\boldsymbol{O_R}$

```
for each object g do:
   if( not Occluded(O_R ,g))
      render(g);
      update(O_R ,g);
   end;
end;
```

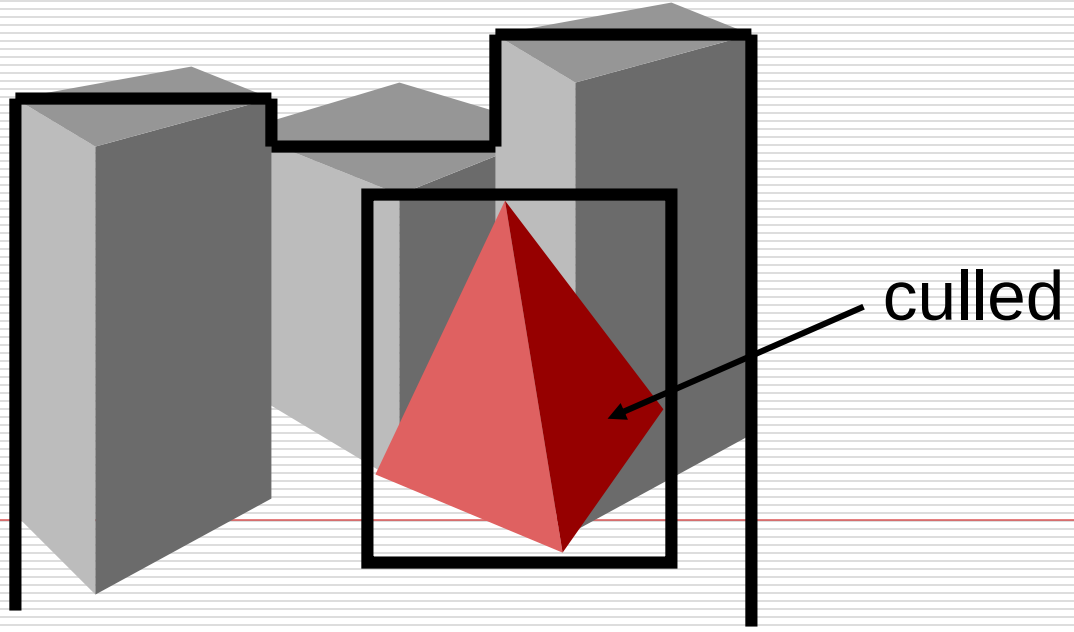
Occlusion Horizon: A simple algorithm



- Target: urban scenery
  - dense occlusion
  - viewer is about 2 meters above ground
- Algorithm:
  - Process scene in front-to-back using a quad tree
  - Maintain a piecewise constant horizon
  - Cull objects against horizon
  - Add visible objects' occluding power to the horizon

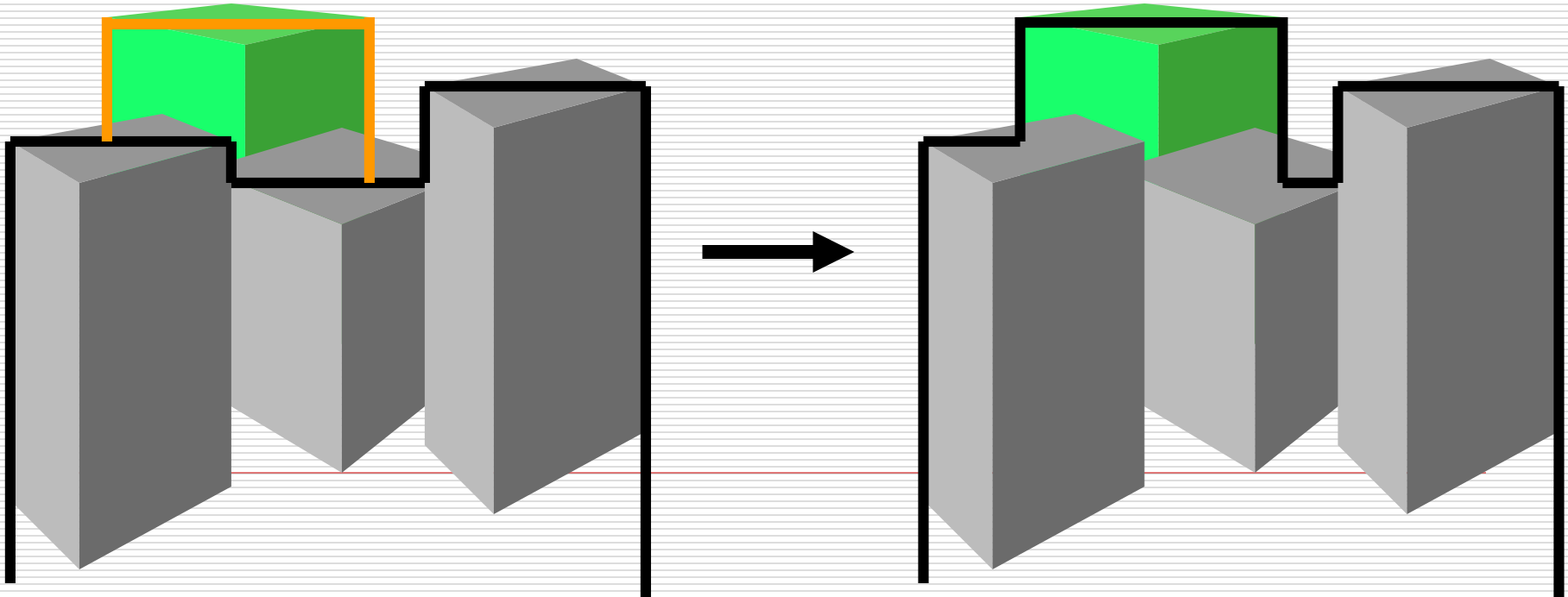
Occlusion testing with occlusion horizons

- To process tetrahedron (which is behind grey objects):
  - find axis-aligned box of projection
  - compare against occlusion horizon



Update horizon

- When an object is considered visible:
- Add its “occluding power” to the occlusion representation



地形的绘制与漫游

- 基于高度图生成地形高度
- 基于四叉树来遍历绘制

- 采用LOD加速
- 纹理压缩减少存储量

一个D3D漫游系统的绘制

- 建模:
  - 模型导入
  - 程序生成
- 绘制:
  - 纹理映射
  - 特殊技巧
- 漫游控制:
  - 第一人称
  - 第三人称视角

- END