
Building a Game World

# An Ideal 3D Game Environment

- Easy to generate
  - World-editing tools
- Dynamic
  - Easy & inexpensive to modify
- Accurate collision detection
- Beautiful, high detail
  - Inexpensive to render
- Easy to transmit over a network
  - Compact representation
- AI can easily navigate all areas

# Back to Reality

- Tradeoffs between detail and object count
- Choices depends on platform, game mechanics
  - Much higher detail where player is likely to focus
  - e.g. Car models vs. background trees in racing games
  - e.g. Dynamic worlds do not have compact representations (MMO)

# World Representation

- This lecture focuses on data structures for organizing a 3D world
- Choices impact our design goals:
  - Easy to generate
    - Tools to create world depend on data structure
  - Dynamic
    - Does the structure need to be rebuilt on every change?  Is this slow?
  - Accurate collision detection
    - Cheaper collision tests => ability to add more complex collision geometry
  - Inexpensive to render
    - Some built for rendering optimations

# Basic World Representation

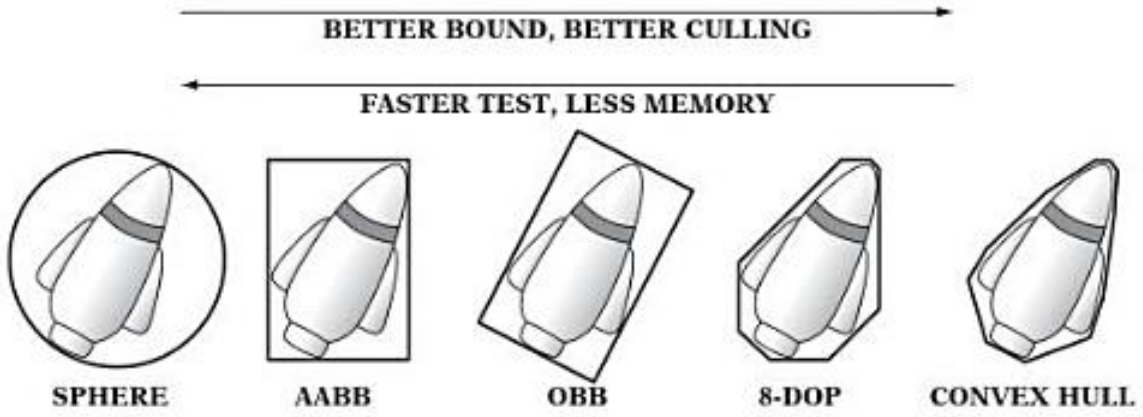
- Polygon soup
  - Most basic form of world representation
  - Unordered collection of polygons
    - No connectivity or hierarchy information
  - Easy for level designers to generate
  - Difficult for collision algorithms
    - Not easy to tell where solid objects are
- Polygon mesh
  - Polygons share vertices, ensure closed mesh
  - Good for representing world, usually want simpler models for entities

# Spatial Organization

- Standard collision detection is $O(n^2)$
- Too slow for large game worlds
- Solution: spatially organize objects to quickly discard far-away data
- Many approaches used in games
  - Bounding volumes
  - Bounding volume hierarchy
  - Uniform grid
  - Hierarchical grid
  - Octree
  - BSP tree
  - Portal-based environment

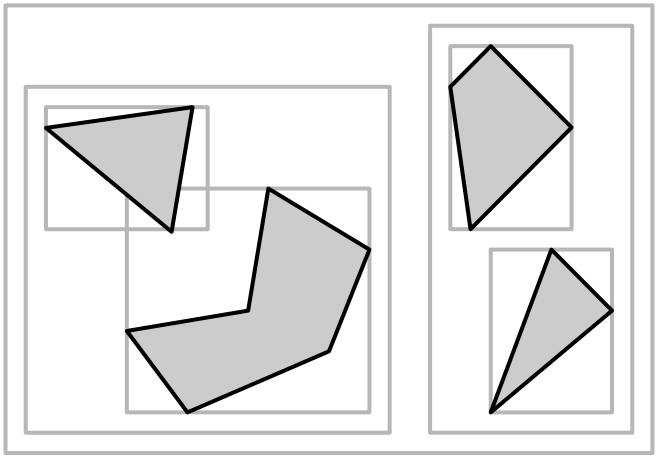
# Bounding Volumes

- Wrap objects in simple geometry to speed up collisions
  - If objects are far apart, only have to do simple test
- Bounding volumes alone don't fix $O(n^2)$ collision issue...



# Bounding Volume Hierarchy (BVH)

- For speeding up collisions against dynamic entities
  - Children have smaller volumes than parents
  - Useful for both collisions and rendering
- Not spatial partitioning: BVH volumes may overlap
- Algorithms exist for bottom-up or top-down construction



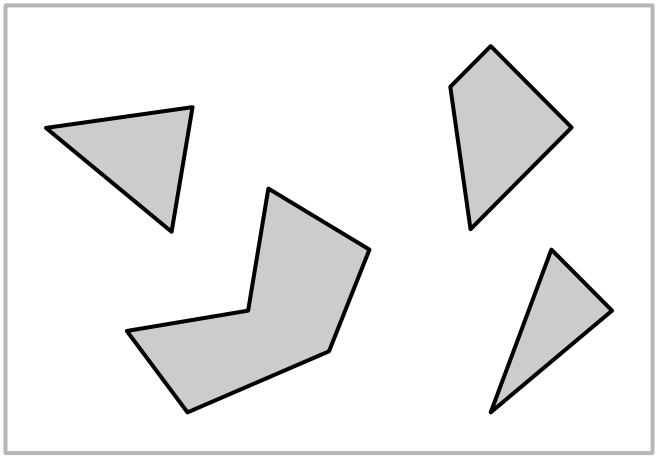
# BVH: Construction

- Top-down construction by far the simplest
- Recursive algorithm:

```
buildBVH(node, objects):
    fit bounding volume of node to objects
    if numObjects <= minObjectsPerLeaf:
        node->objects = objects
    else:
        partition objects according to a heuristic
        buildBVH(node->left, leftObjects)
        buildBVH(node->right, rightObjects)
```

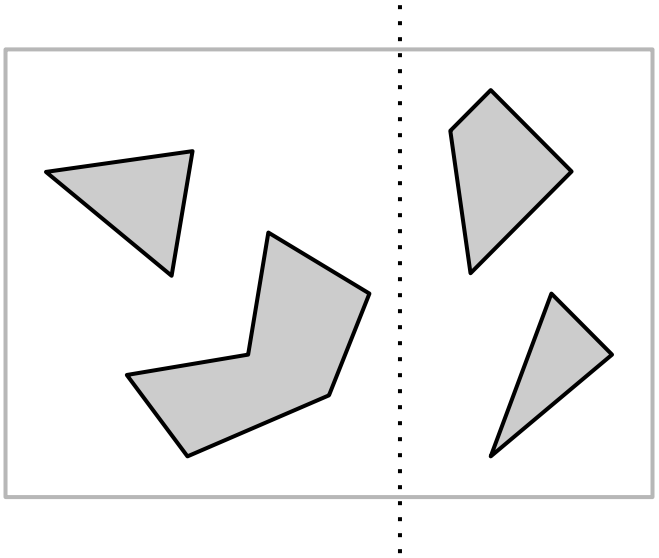
# BVH: Construction

- Top-down construction with AABBs is easiest
  - Start with volume containing all objects
  - Partition children via heuristic (partition discarded)



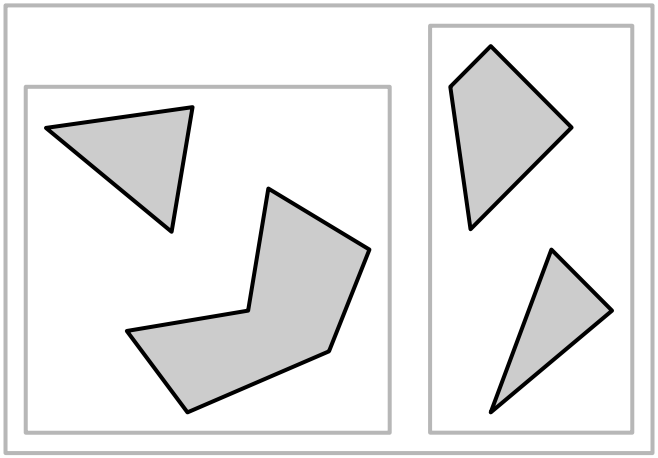
# BVH: Construction

- Top-down construction with AABBs is easiest
  - Start with volume containing all objects
  - Partition children via heuristic (partition discarded)



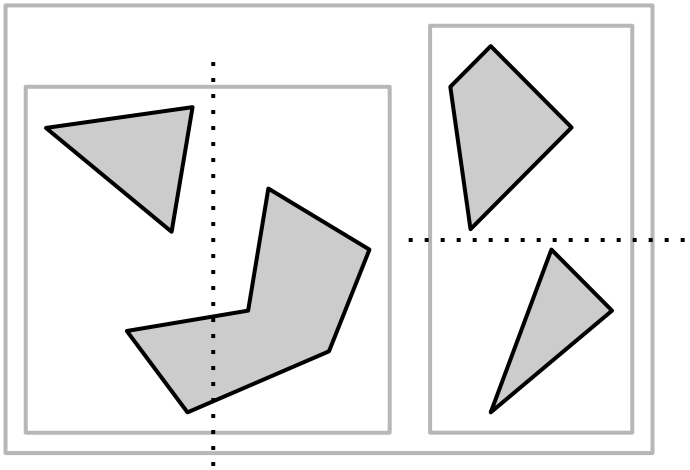
# BVH: Construction

- Top-down construction with AABBs is easiest
  - Start with volume containing all objects
  - Partition children via heuristic (partition discarded)



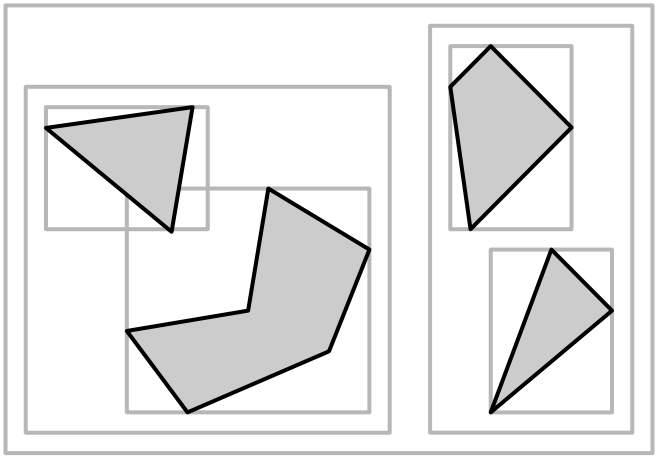
# BVH: Construction

- Top-down construction with AABBs is easiest
  - Start with volume containing all objects
  - Partition children via heuristic (partition discarded)



# BVH: Construction

- Top-down construction with AABBs is easiest
  - Start with volume containing all objects
  - Partition children via heuristic (partition discarded)



# BVH: Construction Heuristics

Goals:

- Node is of minimal volume
- Balanced structure
- Minimal overlap between sibling nodes

A few strategies:

- Median-cut: Divide into equal-size parts with respect to a selected axis
- Minimize sum of child volumes
- Maximize separation of child volumes

Choosing partitioning axes is also difficult

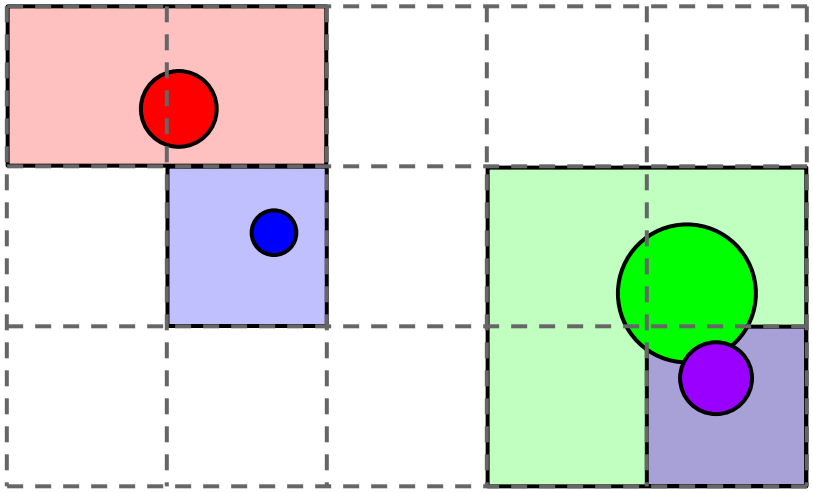
# BVH: Querying

```
collideBVH(object, volume):
    for each child c of volume:
        if c intersects object:
            if c is a leaf:
                collide object with objects in c
            else:
                collideBVH(object, child)
```

- volume is initially the root of the BVH

# Uniform Grid

- For speeding up collisions against dynamic entities
- Axis-aligned cube cells (voxels)
  - Each cell has list of objects
- May be dense (array) or sparse (hash-based)
- Each frame: update cells for objects that moved



# Uniform Grid

Pros:

- Simple to generate and modify
- Good for static and dynamic objects
- O(1) access to neighbors (if grid spacing is max of object sizes)

Cons:

- Not appropriate for objects of greatly varying sizes
- Wastes memory in empty regions if dense representation is used

# Hierarchical Grid

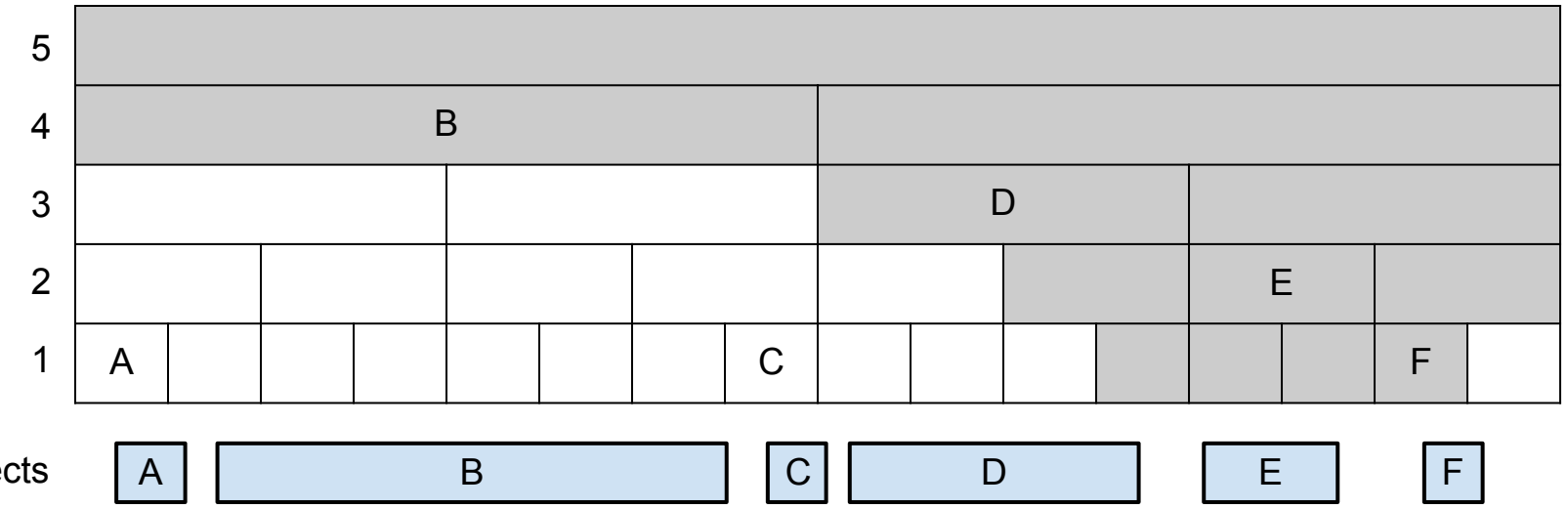
- Hierarchy of different-size uniform grids
  - *n* levels, cells at top level are big enough to cover bounding box of largest game object
  - Cell in level *m* twice the size of cell in level *m* - 1

Construction:

- Insert all game objects into the grid
- To insert:
  - Find lowest level where object fully fits into one cell
  - Insert into cell containing object's center

# Hierarchical Grid: Querying

- Traverse over all levels
  - Test cells containing object's bounding box and neighboring cells
  - Large objects traverse many cells at lower levels
- 1D example
  - Shaded cells are traversed when colliding object E



# Hierarchical Grid

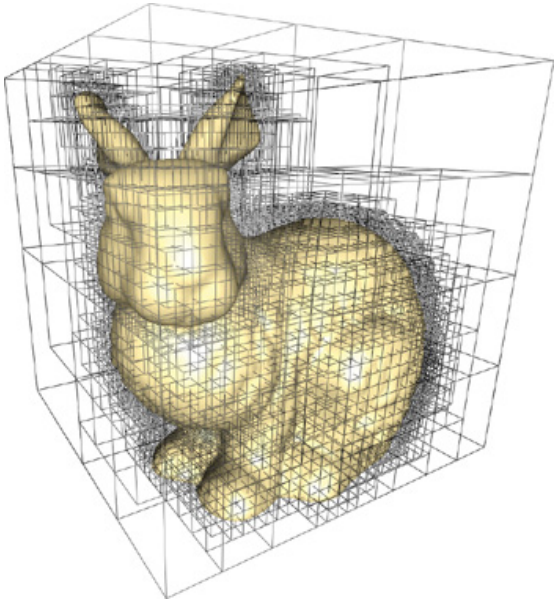
Pros:

- All the benefits of a uniform grid
- Can handle objects of greatly varying sizes
- Insertion is O(1) instead of O(radius$^3$)

Cons:

- Very memory expensive when implemented with a dense array

# Octree



- Axis-aligned hierarchical partitions
- Hierarchical grid with one top-level cell

Pros:

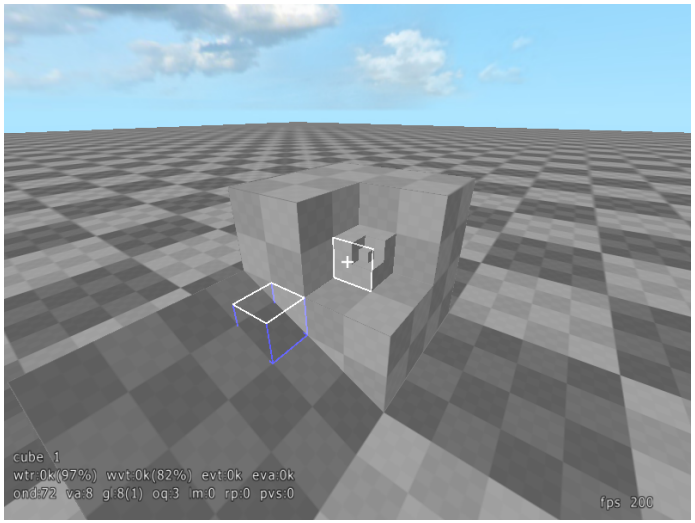
- Simple construction for static scenes
- Easy collision tests: all AABBs

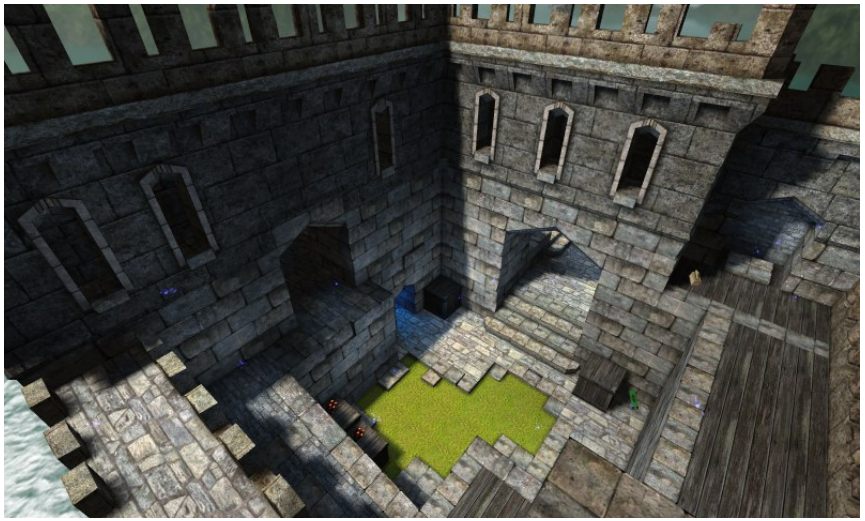
Cons:

- Traversing the tree is expensive if the tree is deep or unbalanced

# Case study: Cube Engine

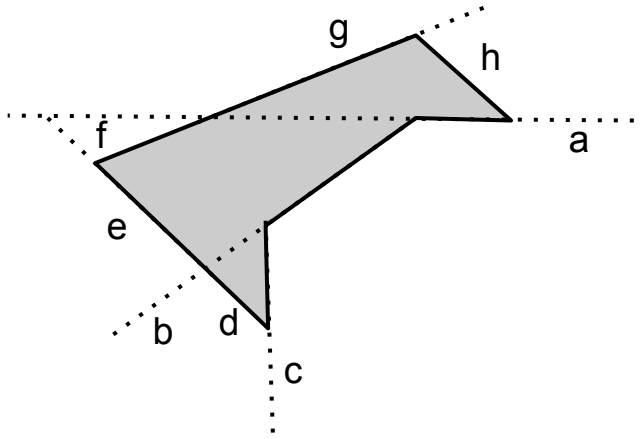
- Open-source networked FPS engine
- Elegant use of octree to represent world
  - "6-directional heightfield deformable cube octree"
    - Vertices of leaf nodes can be moved
    - Ramps, heightfields, and curved surfaces
  - Real-time collaborative map editing

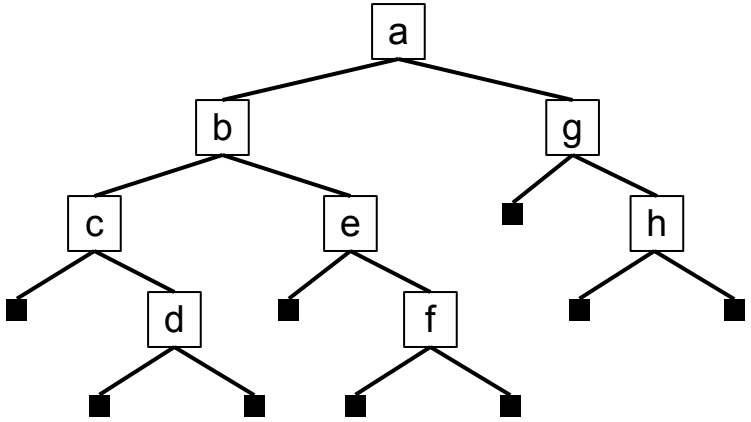


# Binary Space Partitioning (BSP) Tree

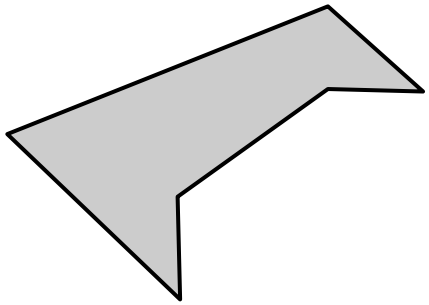
- Hierarchy of planar half-spaces
  - Each node has a plane, divides the space of the parent in two
  - The tree models solid/empty space
- Old technique, used by original Doom and Quake
  - Still used in engines today (Source, Unreal, Call of Duty)
  - Works best for indoor environments (flat, man-made surfaces)



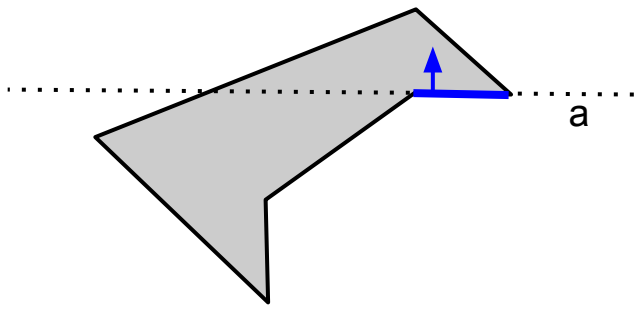
# BSP Tree: Construction

- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance



# BSP Tree: Construction

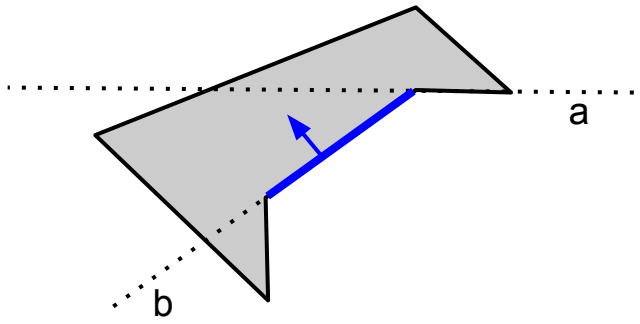
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

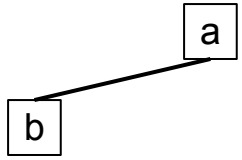



a

# BSP Tree: Construction

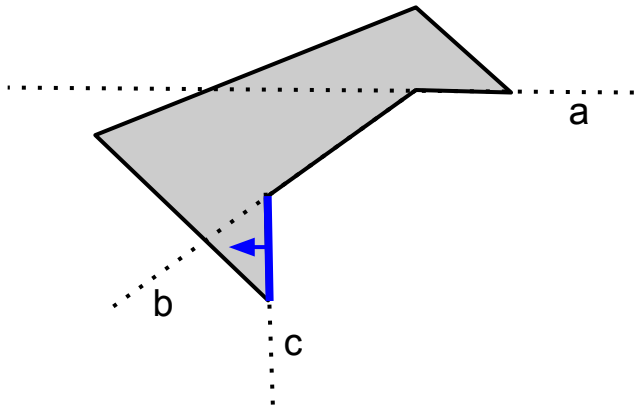
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

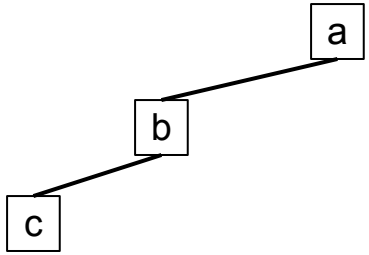


# BSP Tree: Construction

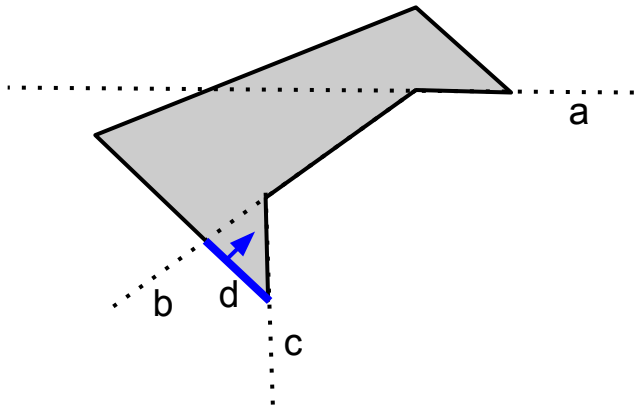
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

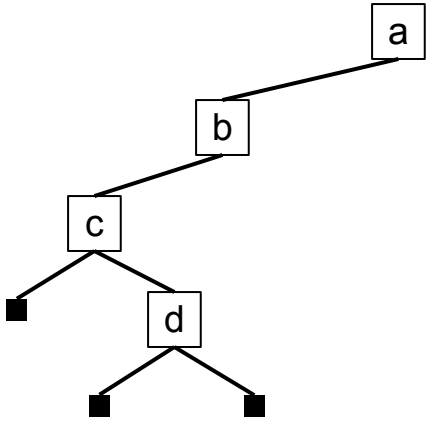


# BSP Tree: Construction

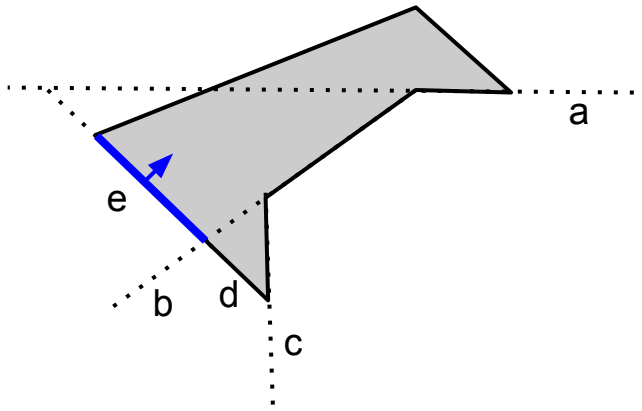
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

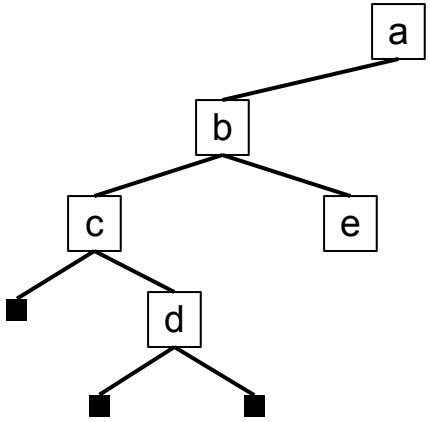


# BSP Tree: Construction

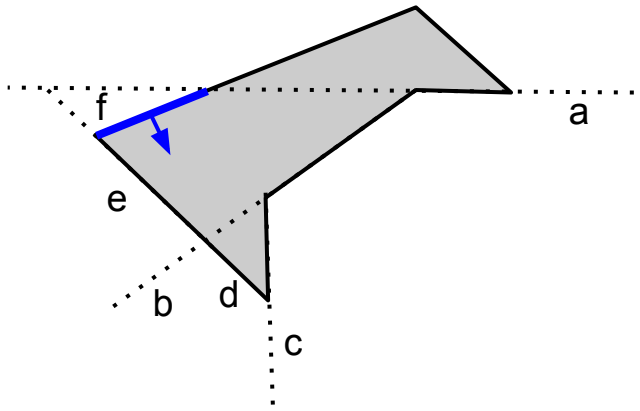
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

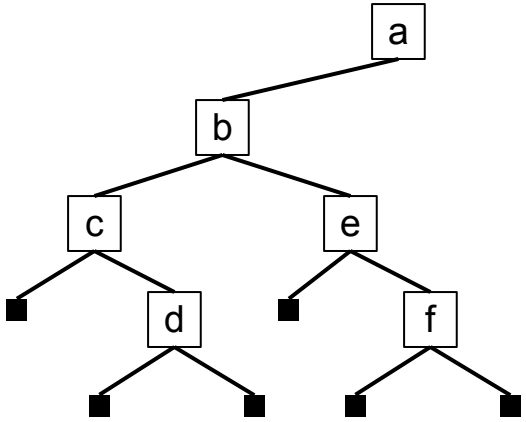


# BSP Tree: Construction

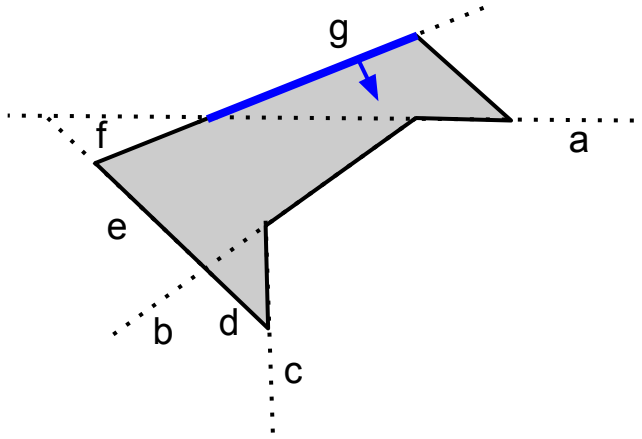
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

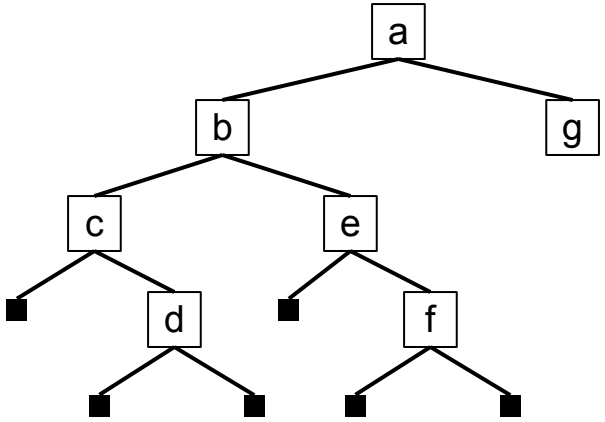


# BSP Tree: Construction

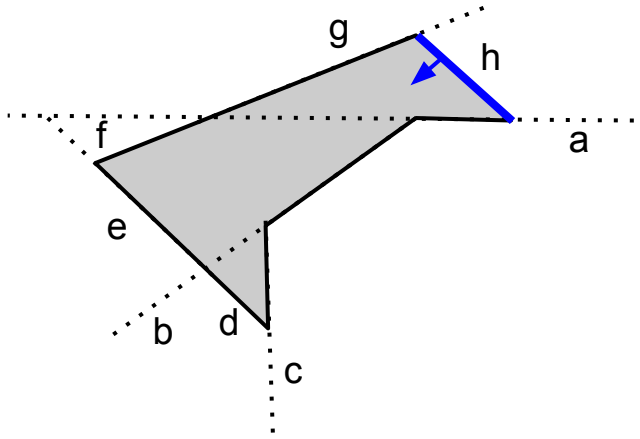
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

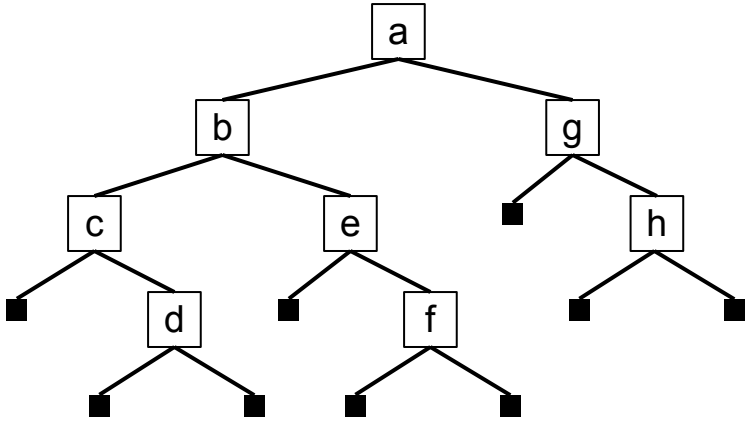


# BSP Tree: Construction

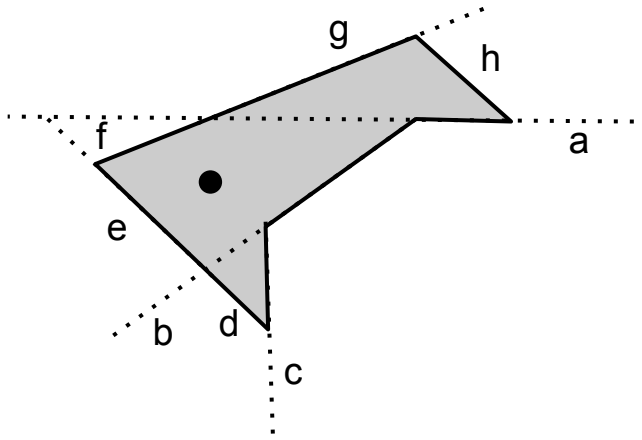
- Overhead view of a 3D room
  - Outline represents walls
  - Gray area represents interior
- Recursively pick a triangle according to a heuristic, split scene along that plane
  - Tree needs to be balanced for good performance

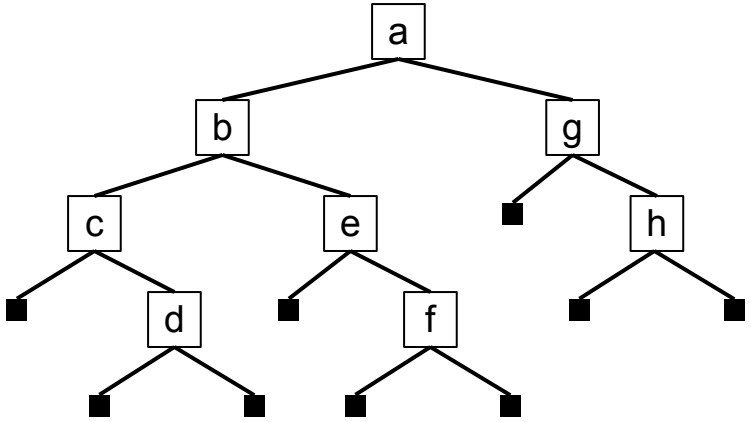


# BSP Tree: Querying

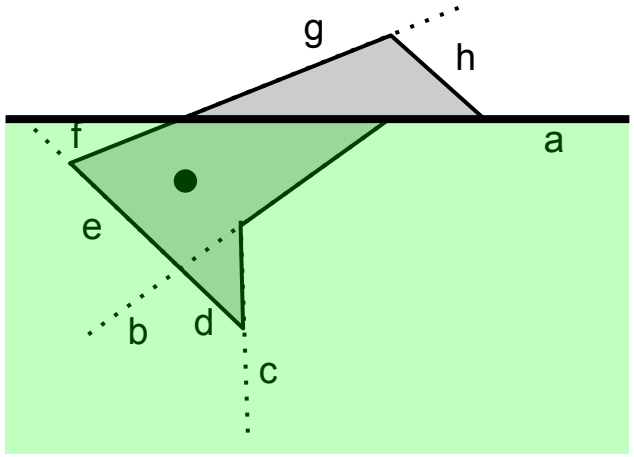
- Test if a point is inside or outside
  - Start at root
  - Visit front child if point in front of plane, otherwise visit back child
  - In front of leaf node = outside, otherwise inside
- Graph: left branch is back, right branch is front

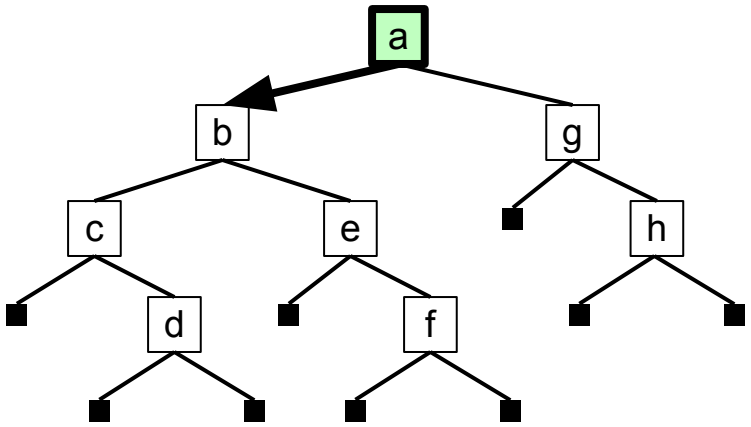


# BSP Tree: Querying

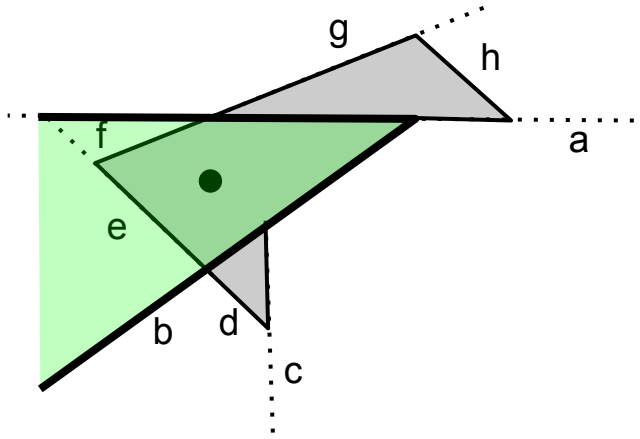
- Test if a point is inside or outside
  - Start at root
  - Visit front child if point in front of plane, otherwise visit back child
  - In front of leaf node = outside, otherwise inside
- Graph: left branch is back, right branch is front

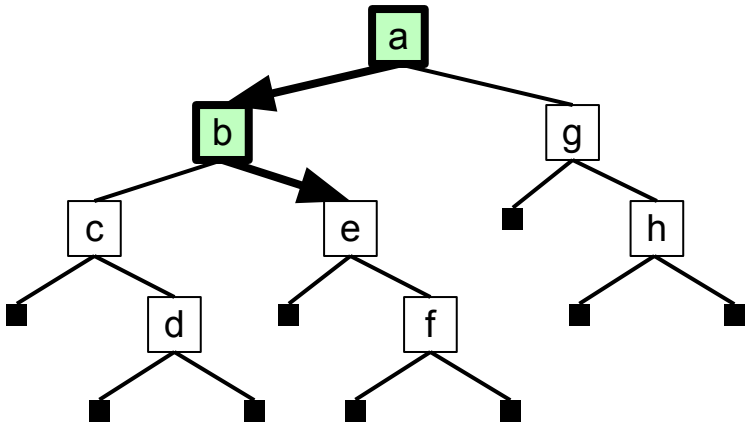


# BSP Tree: Querying

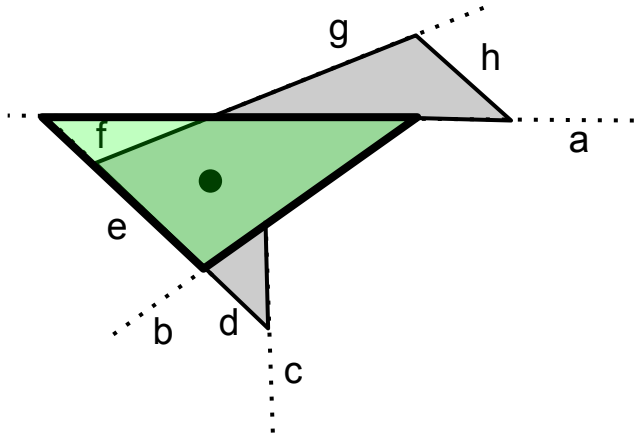
- Test if a point is inside or outside
  - Start at root
  - Visit front child if point in front of plane, otherwise visit back child
  - In front of leaf node = outside, otherwise inside
- Graph: left branch is back, right branch is front

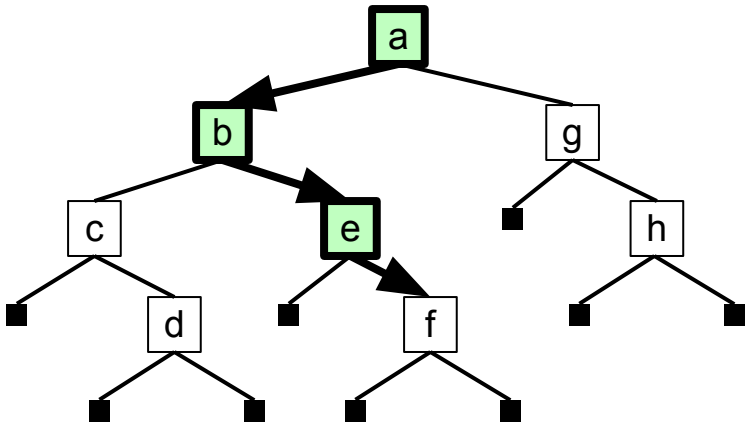


# BSP Tree: Querying

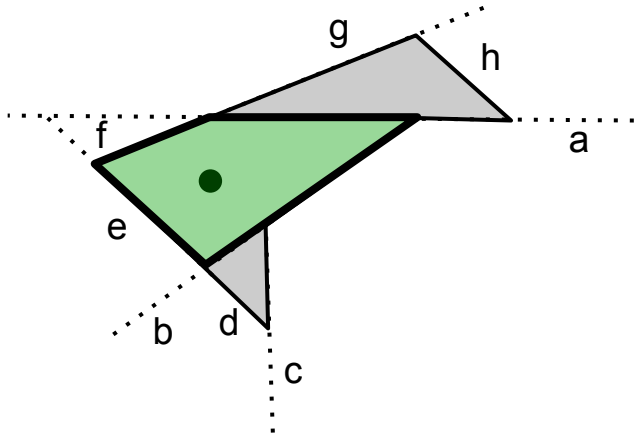
- Test if a point is inside or outside
  - Start at root
  - Visit front child if point in front of plane, otherwise visit back child
  - In front of leaf node = outside, otherwise inside
- Graph: left branch is back, right branch is front

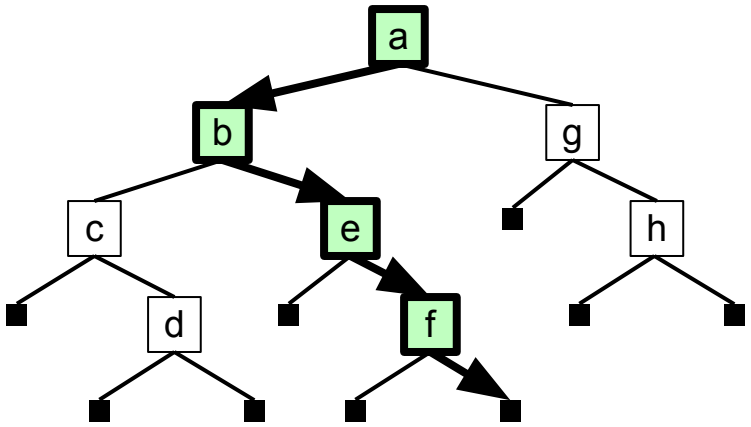


# BSP Tree: Querying

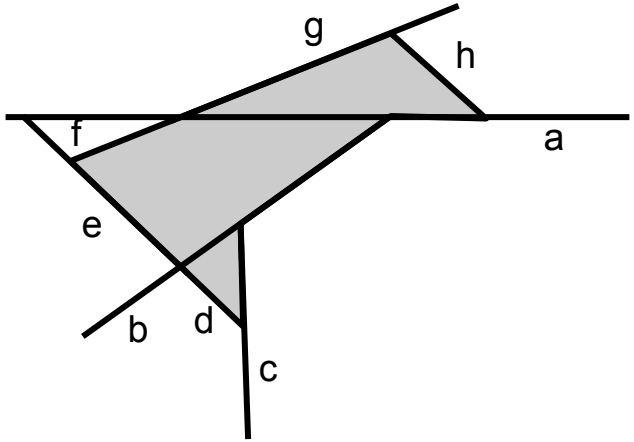
- Test if a point is inside or outside
  - Start at root
  - Visit front child if point in front of plane, otherwise visit back child
  - In front of leaf node = outside, otherwise inside
- Graph: left branch is back, right branch is front

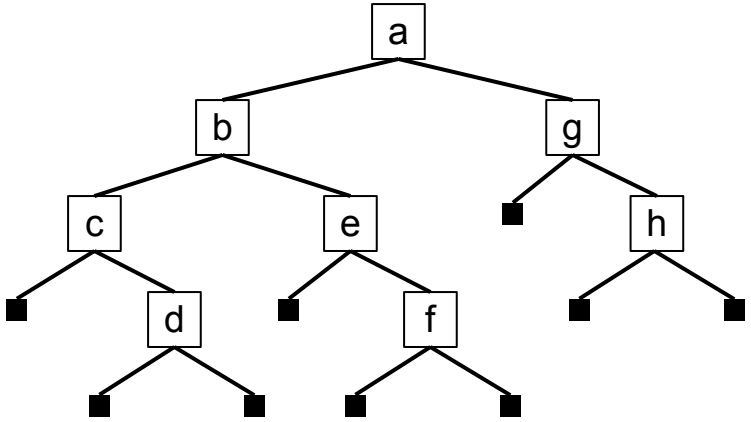


# BSP Tree: Intuition

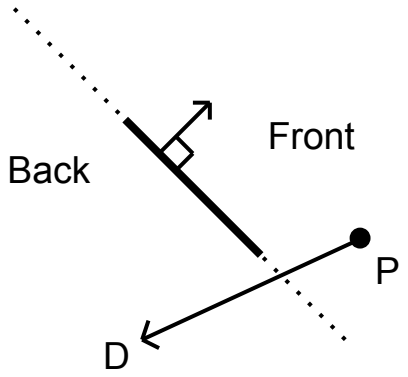
- Each null child represents a convex polytope (polygon in 2D, polyhedron in 3D)
  - Some are solid and some are empty
  - Some have infinite area/volume
  - Shown as small black boxes in graph



# BSP Tree: Ray Traversal

- Given ray (P, D)
- Given root node
- If P is behind plane
  - Recursively traverse back node
  - Hit test polygons on plane
  - Recursively traverse front node (if D • normal > 0)
- If P is in front of plane
  - Recursively traverse front node
  - Hit test polygons on plane
  - Recursively traverse back node (if D • normal < 0)



# BSP Tree: Pros & Cons

Pros:

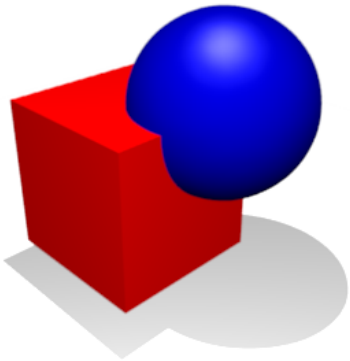
- Generalization of other trees: O(log n) object tests
- Easy collision detection and raycasting

Cons:

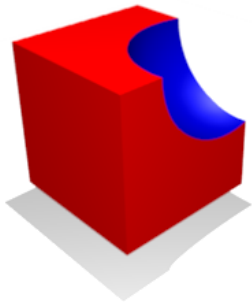
- Very expensive to build (optimal is NP-complete)
- Not suitable for dynamic objects
- Numeric stability can be tricky

# Constructive Solid Geometry (CSG)

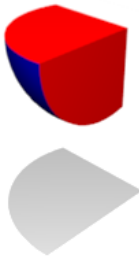
- Boolean set operations on 3D solids
- Useful for level building
- Level editors for games like Unreal and Quake II
  - Valve level editor (Hammer) and Unreal editor (UnrealEd) still based on CSG



Union



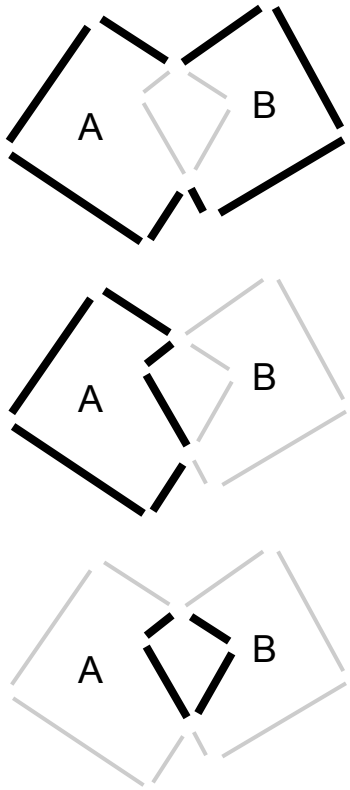
Subtraction



Intersection

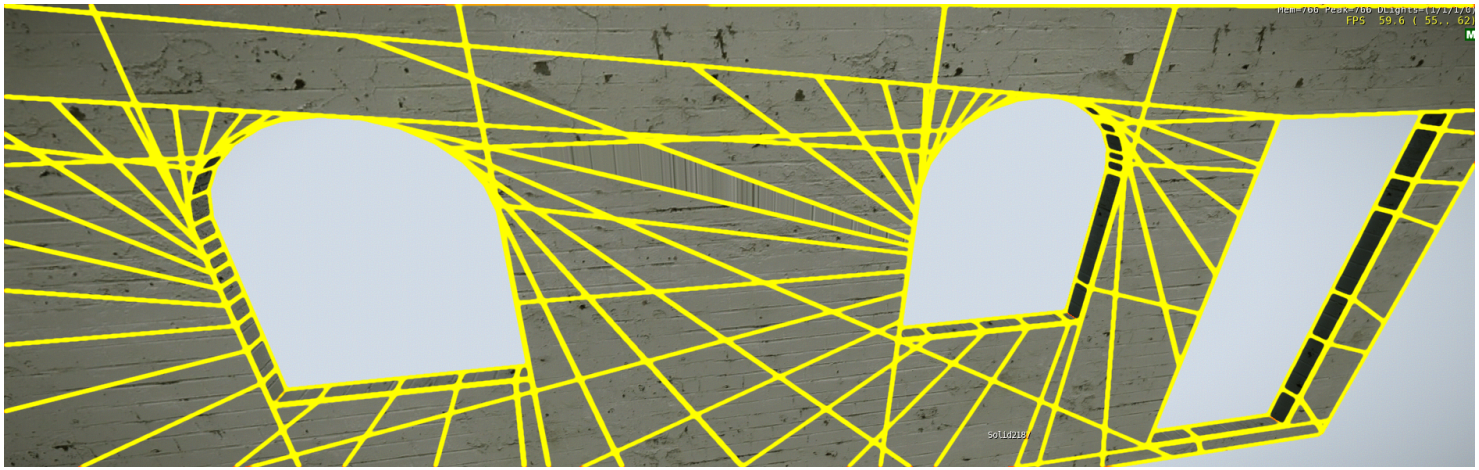
# Constructive Solid Geometry (CSG)

- Elegantly handled by merging BSP trees A and B
  - Split polygons in A by planes in B, get A'
  - Split polygons in B by planes in A, get B'
- Union (A | B)
  - Remove polygons in A' that are inside B
  - Remove polygons in B' that are inside A
- Subtraction (A - B)
  - Remove polygons in A' that are inside B
  - Remove polygons in B' that are outside A
  - Flip orientation of polygons in B'
- Intersection (A & B)
  - Remove polygons in A' that are outside B
  - Remove polygons in B' that are outside A



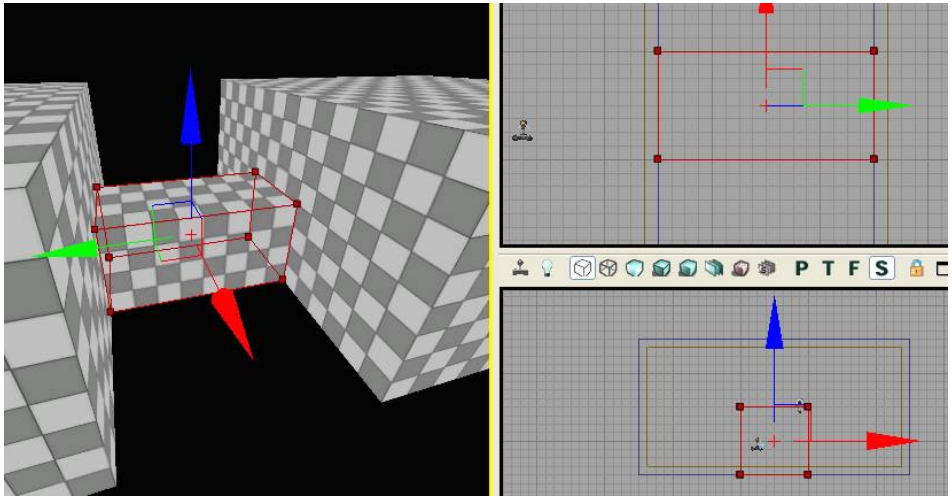
# Constructive Solid Geometry (CSG)

- Problems with BSP trees for CSG
  - Coplanar polygons (neither inside nor outside)
  - Extra geometry due to BSP splits
  - T-junctions



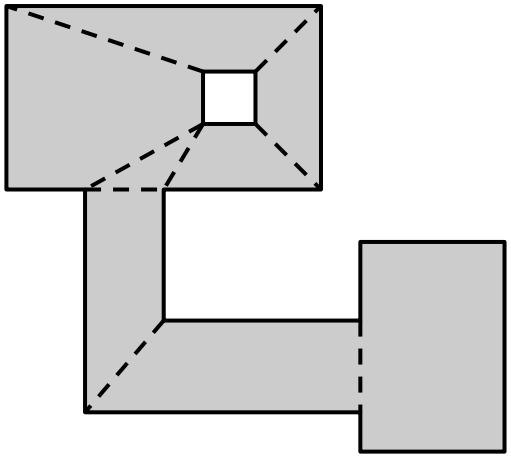
# Case Study: Unreal Engine

- Originally used CSG via BSP merging exclusively
  - Palette of solid "brushes" (cube, cylinder, stairway)
  - World is initially solid, rooms are subtracted
- Modern versions use BSP just for rough level shape
  - Use static meshes for details
  - Avoids problems from previous slide

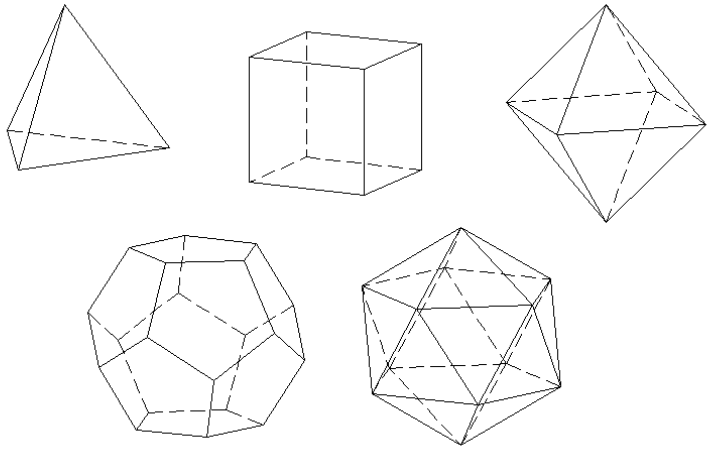



# Portal-Based Environment

- Models empty space, carves world up into non-overlapping convex polyhedra
  - Shared faces between two polyhedra are "portals"
  - Appropriate for indoor environments
- Key insight: no obstruction from any point to any other point inside a polyhedron



Room with portals marked

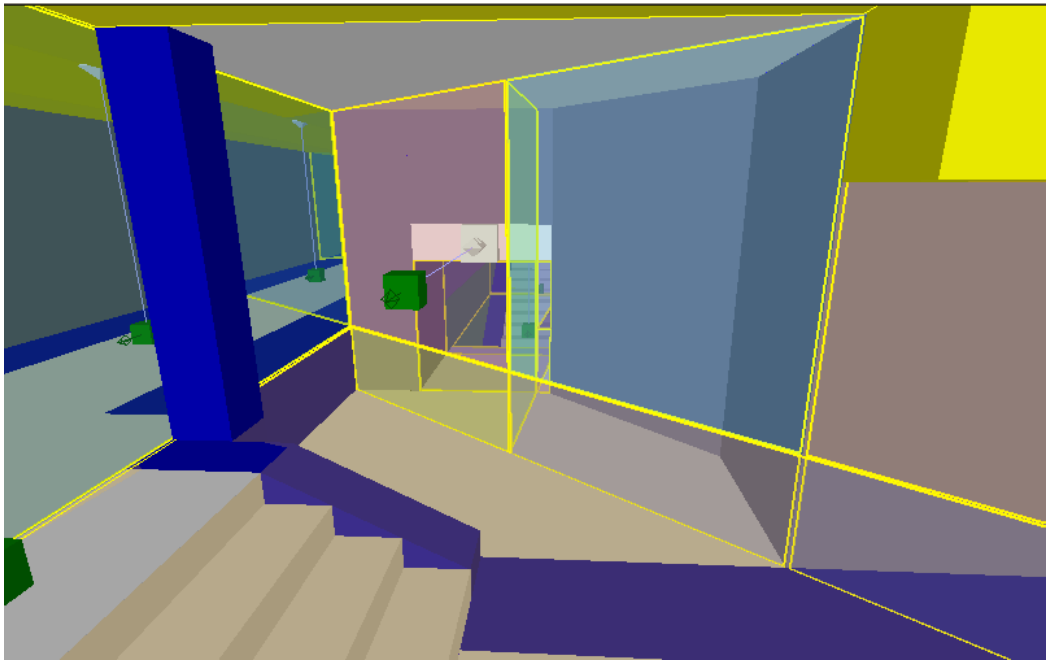


Convex polyhedra

# Portal-Based Environment

Construction methods:

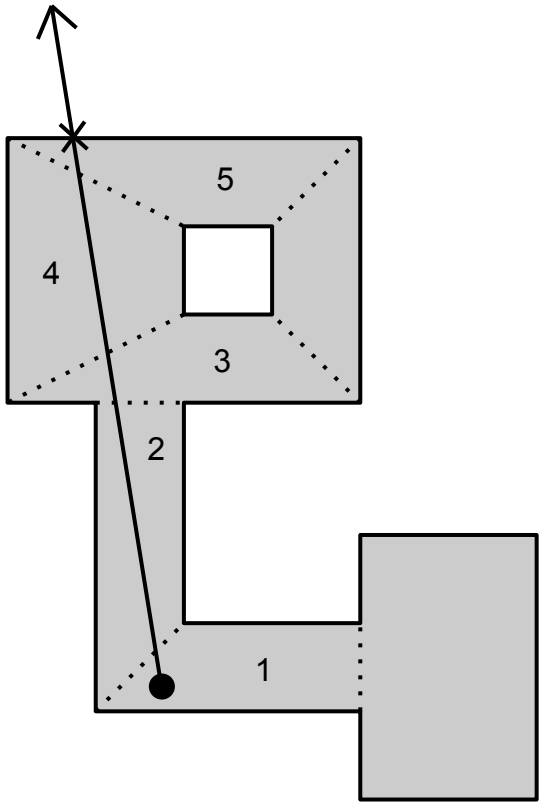
- Model using polyhedra directly
- Use leaves of BSP tree as polyhedra



# Portal-Based Environment

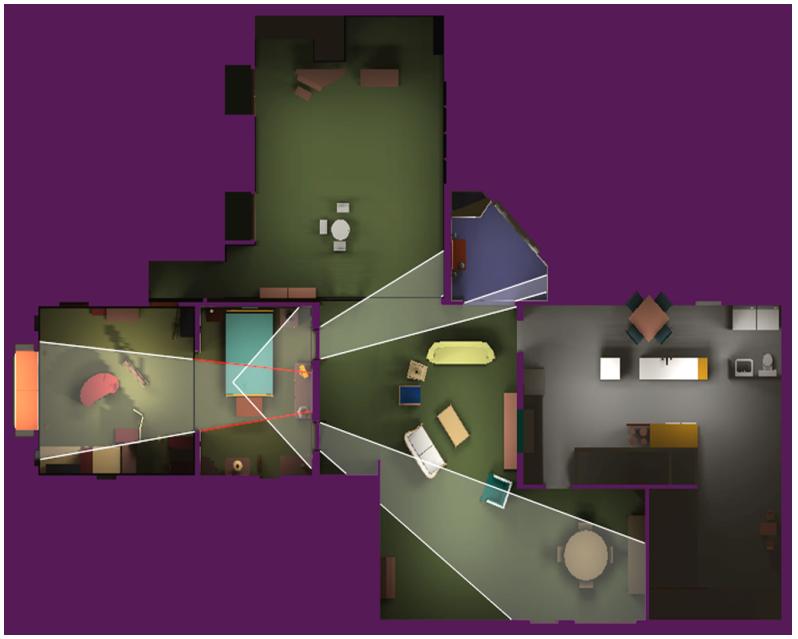
Ray traversal method:

- Start in one polyhedron
- Trace ray to face
  - If solid, stop
  - If portal, move to neighbor



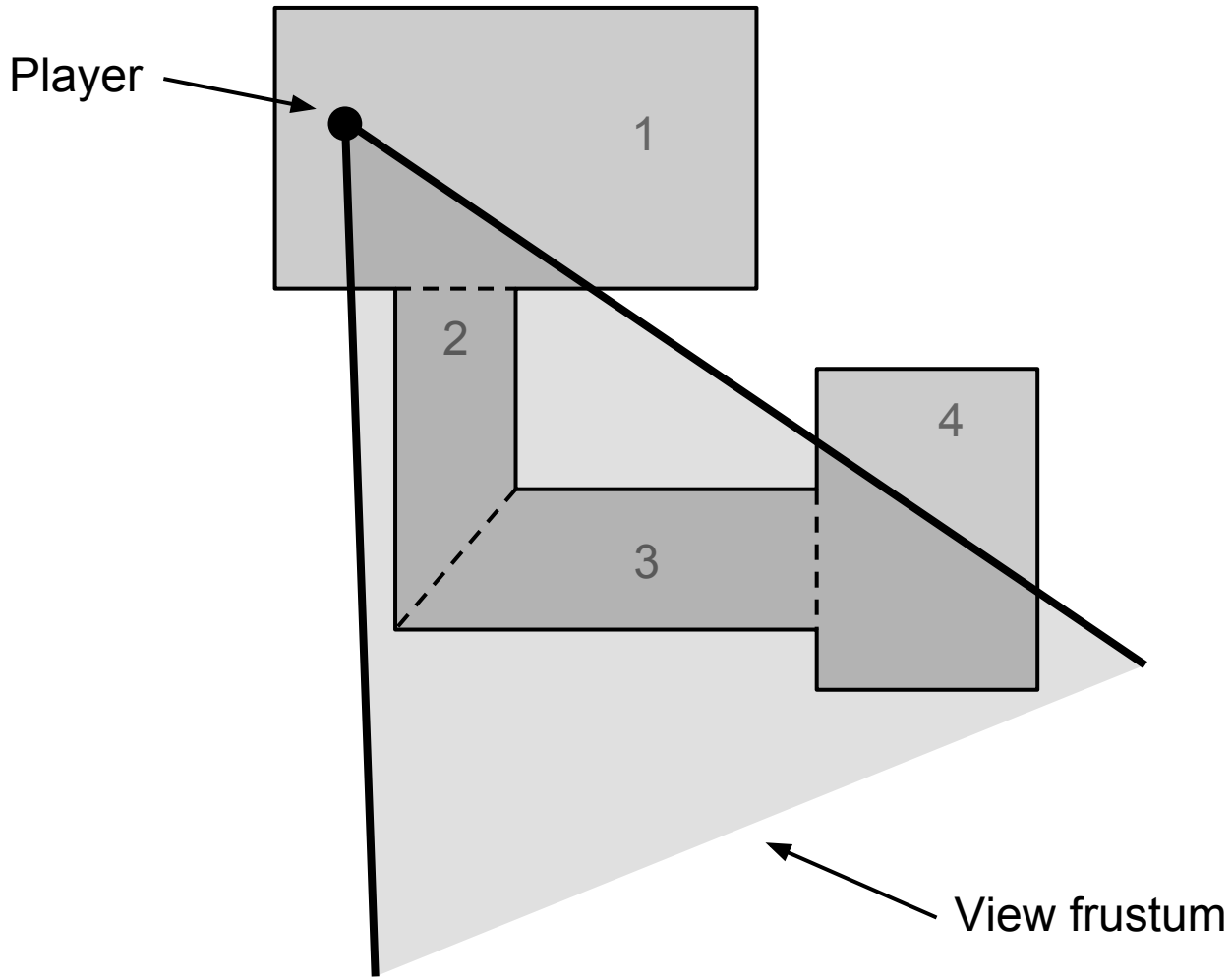
# Portal-Based Environment

- Dynamic visibility set
  - Determine set of objects visible from within each polyhedron on the fly
  - Recursively clip against portal bounds



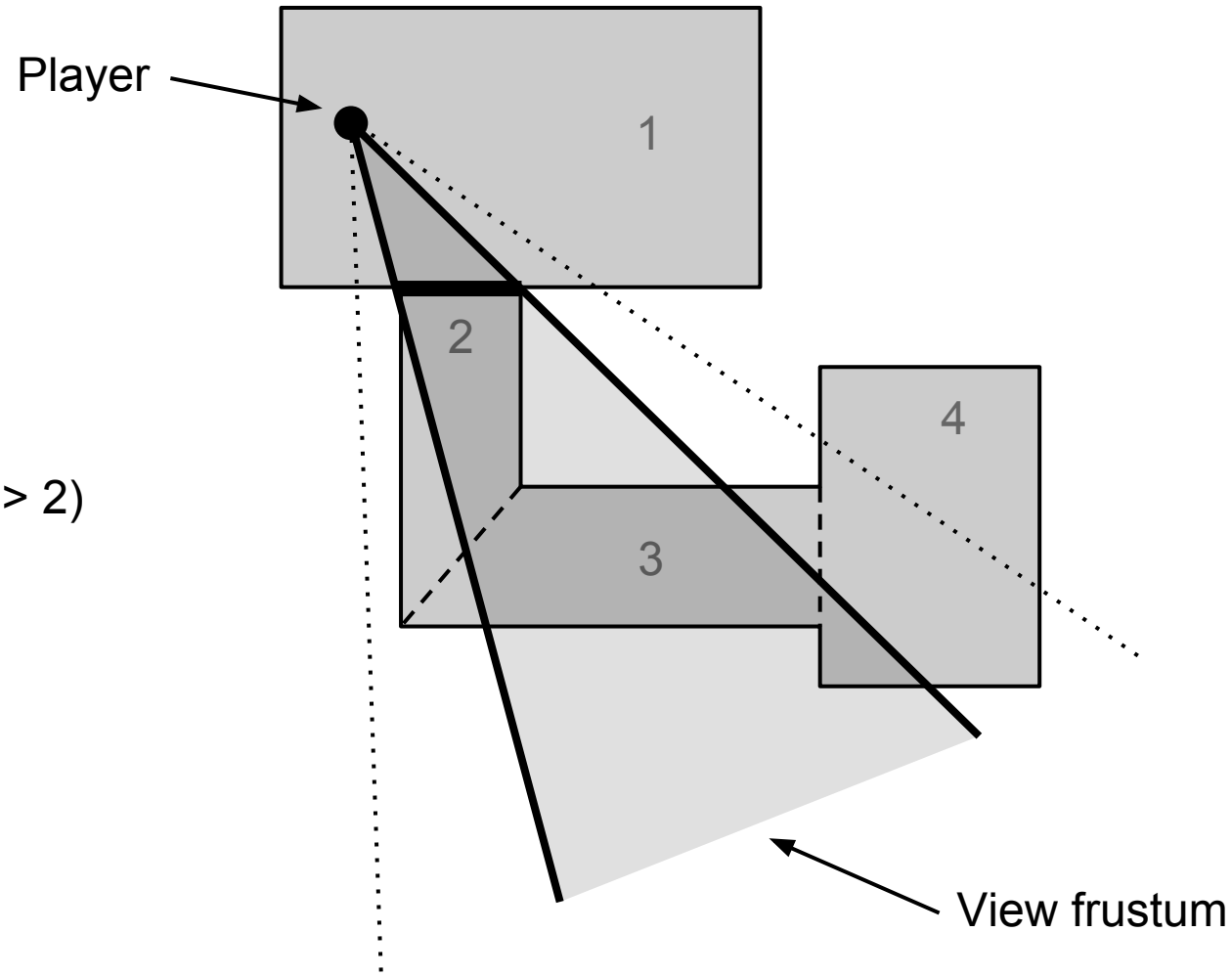
# Portal-Based Environment

Draw room 1
(contains the player)



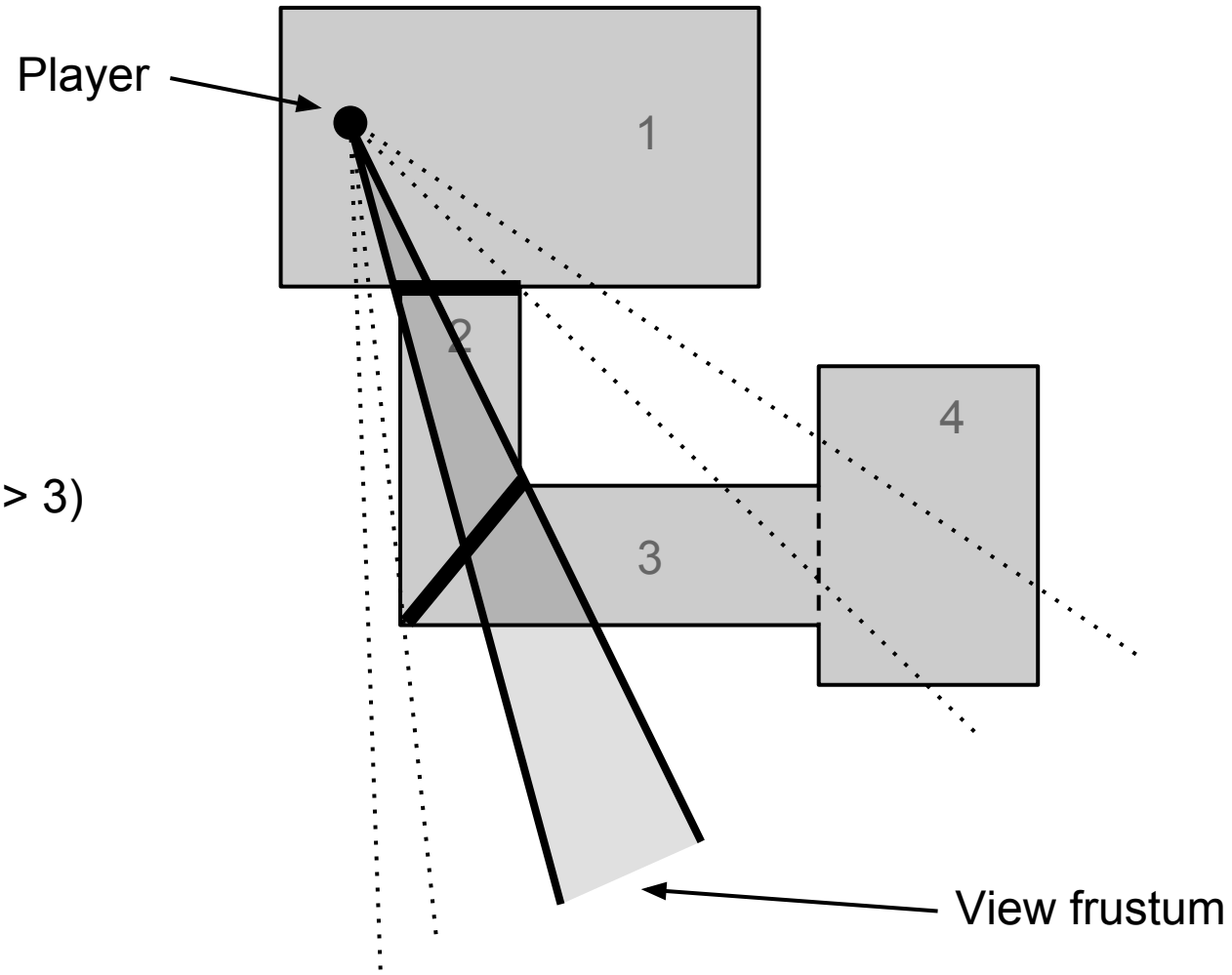
# Portal-Based Environment

Draw room 2
(seen from portal 1 => 2)

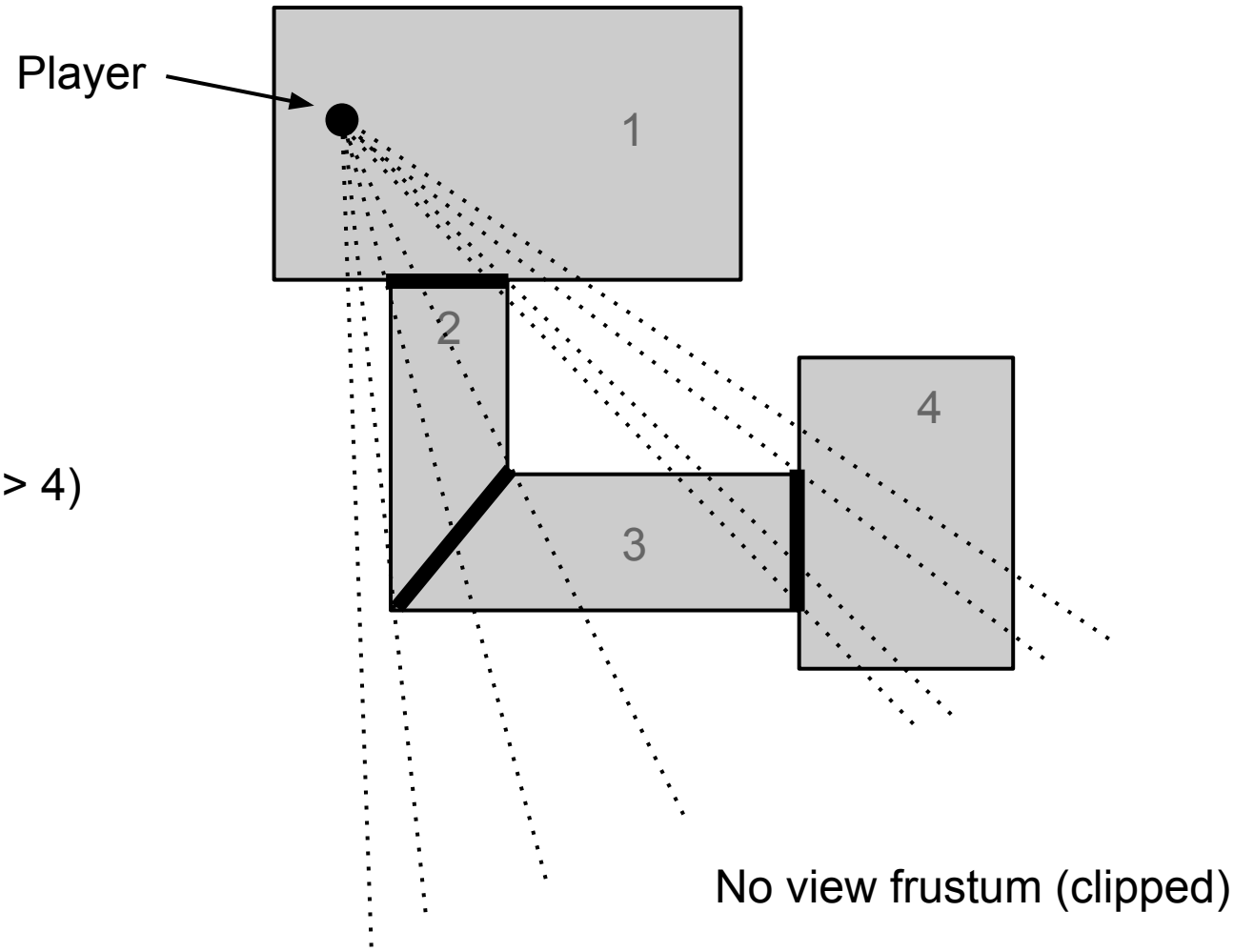


# Portal-Based Environment

Draw room 3
(seen from portal 2 => 3)



# Portal-Based Environment




Don't draw room 4
(seen from portal 3 => 4)

No view frustum (clipped)

# Portal-Based Environment

Pros:

- Easy collision detection and raytracing
- Can be used for dynamic scenes (unlike BSP trees)
- Portals can point to non-adjacent polyhedra
- Mirrors, teleportation, changes in size
- Easy pathfinding, 3D graph of polyhedra (navigation mesh)

Cons:

- Implementation is complex
- Hardware can do visible surface determination better now
- Hierarchical z-buffer occlusion culling

# Case Study: Descent Engine

- One of the first true 3D games
  - Fly with 6 degrees of freedom through tight hallways
- Used deformed cubes as polyhedra
  - Easy to edit, manipulate empty space directly
  - Portal-based rendering led to super-efficient software renderer

# Assignment 2: Minecraft

# Assignment 2: Minecraft

First major assignment, split into three weekly checkpoints

Week 1

- Build a world with textures and terrain

Week 2

- Make rendering more efficient and scalable
- Add a player with collision detection and response

Week 3

- Add simplistic enemies
- Also ability to add and remove blocks
- Stream in chunks as the player moves

Beyond...

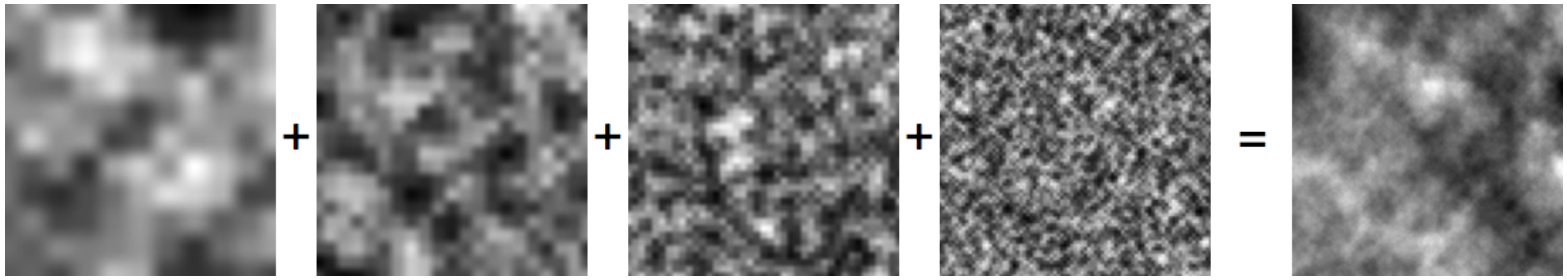
- Play your own version of Minecraft!

# Minecraft: World Format

- Two-layer hierarchical grid
  - 1x1x1 cells grouped into chunks
  - Assignment: 32x32x32 chunks
  - Original Minecraft: 32x128x32 chunks
- Huge number of blocks, memory usage becomes a problem
  - Each block holds one char specifying its type
  - Determine textures, transparency, etc. from type
  - Original game uses a second byte for lighting data
- Why did Notch (creator of Minecraft) choose this data structure?

# Minecraft: Terrain Generation

- Noise function mapping (x, z) to a terrain height (y)
- Why not use rand() to generate random terrain heights?
  - Way too noisy, no correlation between nearby points
  - We want big rolling hills with smaller details
- Octave noise
  - Sum layers of interpolated noise at different scales
  - Look into Perlin noise for more information
  - For a layer, interpolate a hash function sampled in a grid



# Minecraft: Rendering

- Only draw faces that neighbor an empty cell
- Use a `glBegin()` / `glEnd()` block to draw the faces as textured quads
- Rendering will still be slow with lots of blocks
- Next week, you will learn how to make the rendering much faster!

# Reminder: Weekly Next Week!

- 2 minute presentation for checkpoint 1 of Minecraft
- Very informal, no slides / notes
  - Chance to see each other's work
- What to talk about
  - Engine design / class layout
  - Data structure for organizing chunks
  - Terrain generation

# C++ Tip of the Week

- Struct alignment

```
struct Foo {
    char m1;
    char m2;
    short m3;
};
```

- On a 32-bit machine, a *Foo* instance will occupy 12 bytes
  - By default, struct data members are word-aligned
  - Reading / writing aligned memory is faster
  - What if we want to reduce memory usage?

# C++ Tip of the Week

- Struct alignment
  - C++ compilers allow you to pack structs
  - Useful for network buffers and large data sets

```
struct Foo {
    ...
} __attribute__((packed));
```

gcc (top) or msvc (bottom)

```
#pragma pack(push, 1)
struct Foo {
    ...
};
#pragma pack(pop)
```

# References

Ericson, Christer (2005). *Real Time Collision Detection*. Boston, MA: Morgan Kaufmann Publishers.