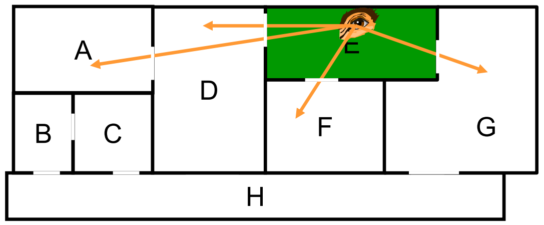
Portals

- A portal is a hole in the wall
- Used for indoor environments
- A portal gives access from one room to another
- Portals for visibility:
  - only render the room the player is in
  - plus any rooms seen through portals
  - plus any rooms seen through portals seen through portals, plus …

Portals (cont.)



Portals (cont.)

- Two variations:
  1. Pre-compute what rooms are visible from each room
  2. Compute what rooms are visible from the current view position during game
- Like PVS or raycasting, but at a coarser level

Portals (cont.)

- Portals do not need to be bi-directional, or lead to adjacent rooms
  - Teleporters, TVs
- Portals are usually represented in the world by a polygon
- Portals can be any shape. Use a simple shape like a rectangle for visibility testing

Portals (cont.)

```
// Example room with portals structure
struct Portal
{
   Polygon poly;       // Polygon of portal
   Room* otherroom;    // The room on the other side
   Polygon otherpoly;  // Polygon of portal exit on the other side
};
struct Room
{
   vector<Portal> portals;     // Portals of the room
   vector<Room*> visiblerooms;// Rooms visible from this room
};
```

Portals (cont.)

- Room-to-room visibility
  - A room R_1 is visible from room R_0 if there is a ray from room R_0 through one or more portals to room R_1
  - Only need to check rays that go through portals of R_0 and eventually exit a portal of R_1
  - So actually checking portal-to-portal visibility

Portals (cont.)

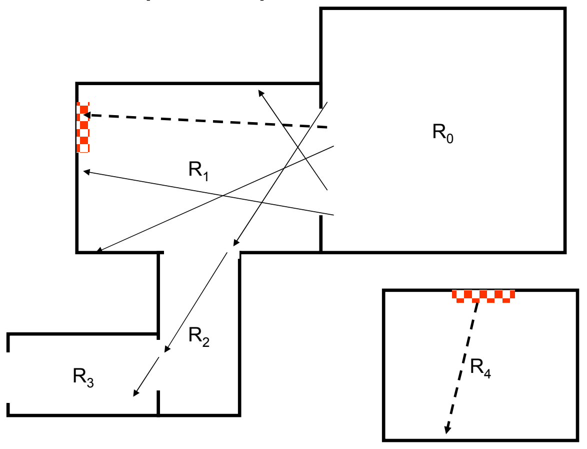
- Algorithm overview:

```
1. Create empty list L of rooms visible from room R
2. Add room R to L
3. For all portals P in room R
     1. For all rays through portal P
          1. Cast ray through portal P of room R

Cast ray through portal P of room R:
1. Let R' be the room on the other side of P
2. Transform ray to room R'
3. Add R' to list of visible rooms L
4. Trace ray through R'
5. If ray intersects a portal P' in R'
     1. Cast ray through portal P' of room R'
```

- May need to include protection against infinite recursion
  - Limit to total length of ray
  - Limit number of recursive `Cast ray` calls
  - Limit number of times a room can be visited

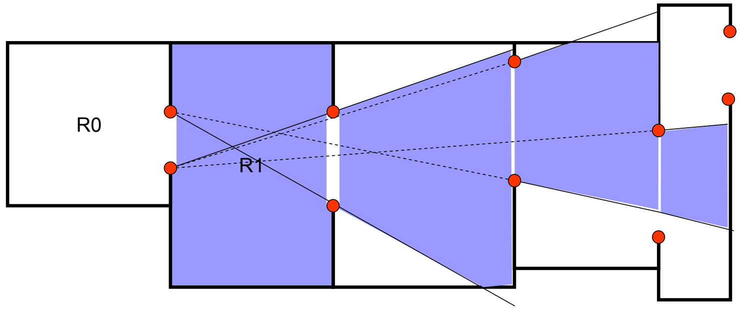
Portals (cont.)



Portals (cont.)

- But there are an infinite number of rays!
- Could organise world using a regular grid
  - Use raycasting as before
  - Compute PVS for each grid square at a portal in room R_0 , keeping track of what rooms are being visited by the raycasting
- Alternatively, can determine visibility by looking at the range of rays that go through a pair of portals, and clipping against subsequent portals

Portals (cont.)



Portals (cont.)

```
1.  Create empty list L of visible rooms
2.  Add room R to L
3.  For each portal P of room R
    1.  Let R’ be the room on the other side of P
    2.  Add R’ to visible list L
    3.  For each portal P’ of room R’
        1.  Find viewing extent V from P to P’
        2.  Find rooms visible from R’ through P’ using V

Find rooms visible from R through P using V:
1.  Let R’ be the room on the other side of P
2.  Add R’ to visible list L
3.  For each portal P’ of room R’
    1.  Let V’ be the viewing extent V clipped to portal P’
    2.  If V’ is not empty
        1.  Find rooms visible from R’ through P’ using V’
```

Portals (cont.)

- Portals define a depth sorted order:
  - Anything seen through a portal will never appear in front of something in the room

```
Render room:
1.  Render polygons in room
2.  For all portals in the room
    1. Render room on other side of portal
```

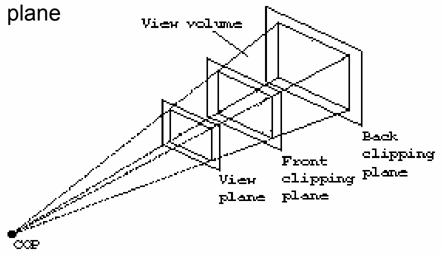
- View frustum culling for faster portal rendering
  - To render a room through a portal, set clipping to render only in the portal
    - glClipPlane() using camera position and edges of portal
    - glViewport() using bounding rectangle of projected portal
    - glScissor() using bounding rectangle of projected portal

Portals (cont.)

- Portals as mirrors
  - Set the room on the other side of the portal to be the same room
  - Set a transformation to mirror any ray going through the portal
- Portals do not need to appear as doors or windows
  - Can be used to divide the world into more manageable chunks
  - Invisibly transport the player to a different level

View frustum culling

- The view frustum is the pyramid which defines the extent of the camera's view
  - Defined by four planes, going through the camera's centre of projection and each side of the view plane, plus a front and back clipping plane



View frustum culling (cont.)

- Arbitrarily define the view frustum planes to be outward facing (normals pointing away from the view frustum)
  - The outside of a plane is the region where the normal is pointing towards. The other side is the inside.
  - Any point on the plane is considered inside
- Anything outside of the view frustum will not be seen, and is therefore a candidate for culling
- To check if a vertex is within the view frustum:
  - If vertex is on the inside of all six view frustum planes, vertex is inside view frustum
  - If vertex is on the outside of any of the planes, vertex is outside view frustum

View frustum culling (cont.)

- Testing a vertex against a plane:
  - Plane equation:

    $$ax + by + cz + d = 0$$

    (a,b,c) is the normal N of the plane

    $-d$ is the distance of plane from the origin along N
  - To determine which side a vertex $V = (x,y,z)$ is on, project it onto the plane normal N using dot product:

    $$v' = N . V = ax + by + cz$$

    v' is the distance of V along the normal N. The plane is at distance $-d$ along the normal, so distance of V from plane is:

    $$d_V = v' - (-d) = ax + by + cz + d$$

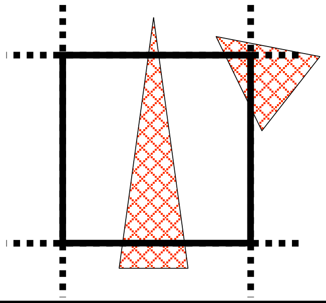
    If $d_V = 0$, V is on the plane

    If $d_V < 0$, V is inside the plane

    If $d_V > 0$, V is outside the plane

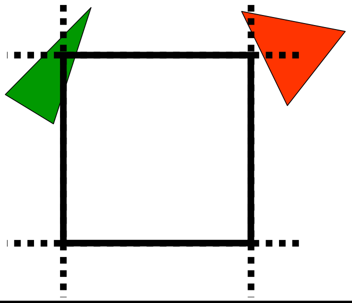
View frustum culling (cont.)

- To test a polygon against the view frustum
  - WRONG: take each vertex and test it against the view frustum. Cull polygon if all are outside



View frustum culling (cont.)

- To test a polygon against the view frustum
  - HALF-RIGHT: take all vertices and test them against each plane. If all are outside the same plane, cull polygon



View frustum culling (cont.)

```
Test polygon against view frustum:
1.   For all view frustum planes
   1.   Test polygon against plane
   2.   If polygon is outside plane
      1.   Return polygon outside view frustum

Test polygon against plane:
1.   For all vertices of polygon
      1.   Test vertex against plane
      2.   If vertex is inside plane
         1.   Return polygon is inside
2.   Return polygon is outside
```

View frustum culling (cont.)

- To test a polygon mesh against the view frustum
  - The slow way: like with polygons
    - For each view frustum plane, test mesh polygons against plane. If all polygons are outside that plane, mesh is outside view frustum
  - The fast way: use a simple bounding volume
    - Test bounding volume against view frustum. Cull mesh if bounding volume is outside view frustum
    - For example, given bounding sphere:
      - If distance of sphere centre to any frustum plane is larger than the sphere radius, sphere is outside the view frustum, and hence the enclosed mesh is outside the view frustum

View frustum culling with quadtrees

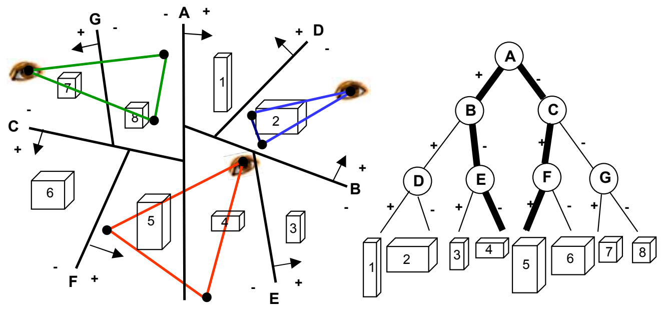
- Quadtrees are hierarchical
  - The bounding square of each child node is within the bounding square of its parent
- Implies that if a node in the quadtree is not visible, then none of the children will be visible
- Test the bounding square of the current quadtree node against the view frustum
- If node is visible, recurse by going down the tree checking each of the four children
- Optimisation: reuse the visibility computed for the grid points of the parent

View frustum culling with octrees

- Same as with quadtrees, except the visibility test is between the 3D bounding box and the view frustum

Culling with BSP trees

- If a partition plane does not intersect the view frustum, only the side that the camera is in will be visible



View Frustum Culling: View #1: A, B, D, 2 ; **View #2:** A, C, G, 7, 8 ; **View #3:** A, B, E, 4, C, F, 5

Culling with BSP trees (cont.)

```
Determine BSP visibility of node:
1.  If node is a leaf node
    1.  Node is visible
2.  Else
    1.  If partition plane of node does not intersect view frustum
        1.  If camera is on left side of partition plane
            1.  Entire right sub-tree is invisible
            2.  Determine BSP visibility of left child
        2.  Else
            1.  Entire left sub-tree is invisible
            2.  Determine BSP visibility of right child
    2.  Else
        1.  Determine BSP visibility of left child
        2.  Determine BSP visibility of right child
```

Culling with BSP trees (cont.)

```
Render BSP node:
1.    If node is a leaf node
    1.    Render geometry in node
2.    Else
    1.    If partition plane of node does not intersect view frustum
        1.    If camera is on left side of partition plane
            1.    Render BSP left child node
        2.    Else
            1.    Render BSP right child node
    2.    Else
        1.    If camera is on left side of partition plane
            1.    Render BSP left child node
            2.    Render BSP right child node
        2.    Else
            1.    Render BSP right child node
            2.    Render BSP left child node
```

Non-view frustum culling

- Can generalise to culling to work with any object or polygon, not just the view frustum
- To cull everything outside a given region:
  - Define a set of outward facing planes that enclose the region
  - Do visibility culling using that set of planes
    - Exactly like with the view frustum planes
- To cull everything inside a given region:
  - Define a set of inward facing planes that enclose the region
  - Cull anything that is entirely outside all planes
- Planes defined by e.g. camera position plus edges of a convex polygon, or the sides of a bounding box, or the player position and the sides of a portal