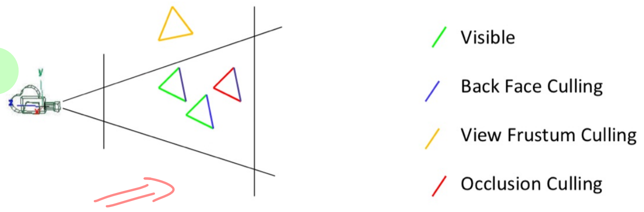
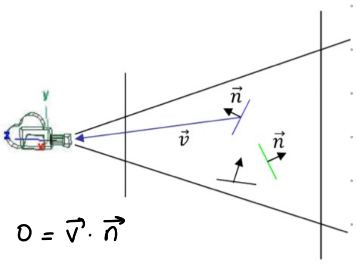
Culling

culling: avoids (fully) processing every triangle/model
- back face culling
- view frustum culling
- occlusion culling

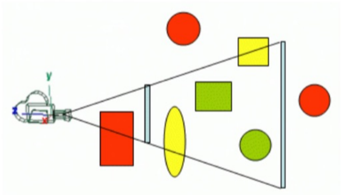


Back Face culling: do not process triangles facing away from the camera, eliminating a large number of triangles. Performed in hardware for every triangle - implies triangle submission. The hardware based approach still requires to draw the vertices. The elimination only occurs in the pipeline after the primitives are built. Ideally we would avoid the unnecessary requests and processing the vertices. However, a CPU based solution for individual triangles would be too slow.
- group triangles according to their normal and work with groups instead of individual triangles.
- eliminate triangle/object outside of the view frustum
- steps:
  - → get the frustum plane equations (once per frame)
  - → test for each triangle/object/vertex if it is inside of the frustum

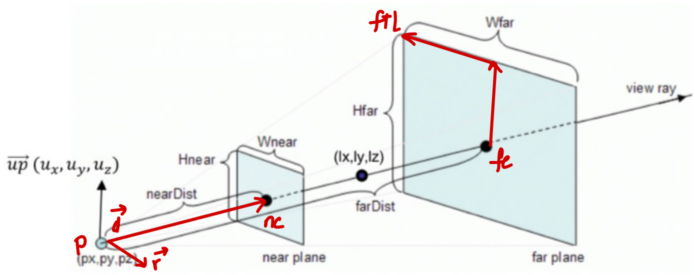


$O = \vec{v} \cdot \vec{n}$

if (O > 0) render
else cull



View Frustum culling:
- setup: get the frustum plane equations (once per frame)
- test: for each vertex/triangle/object if it is inside/outside the frustum



A plane is defined by a normal and a point. Far plane can be defined by normal: $-\vec{d}$ and a point:

$fc = p + \vec{d} * farDist$

To define the normal we need 3 points:

$ftl = fc + \left(\vec{up} \times \frac{Hfar}{2}\right) - \left(\vec{r} \times \frac{Wfar}{2}\right)$

gluPerspective(fov, ratio, nearDist, farDist);
gluLookAt(px, py, pz, lx, ly, lz, ux, uy, uz);

$Hnear = 2 * \tan\left(\frac{fov}{2}\right) \times nearDist$

$Wnear = Hnear \times ratio$

$Hfar = 2 * \tan\left(\frac{fov}{2}\right) \times farDist$

$Wfar = Hfar * ratio$

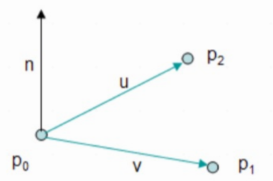
Normalized Plane Equation:

$\vec{n} = (n_x, n_y, n_z) = \vec{u} \times \vec{v}$

$\vec{n} = \vec{n} / |\vec{n}|$

$A = n_x \quad B = n_y \quad C = n_z$

$$\boxed{\begin{aligned} &A p_{0x} + B p_{0y} + C p_{0z} + D = 0 \\ (=)\ \ &D = -A p_{0x} - B p_{0y} - C p_{0z} \end{aligned}}$$



Distance from point to plane:

$$\boxed{dist(p) = Ap_x + Bp_y + Cp_z + D}$$

if $dist(p) > 0$ then p is on the side where the normal is pointing

Test:

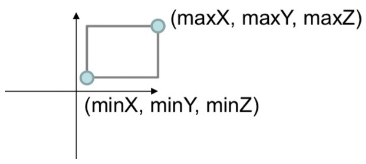
Assuming normals point to the frustum's inside

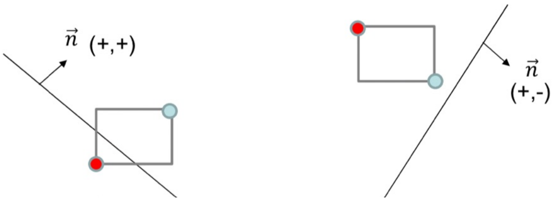
```
int FrustumG::pointInFrustum(Vec3 &p) {

        int result = INSIDE;
        for(int i=0; i < 6; i++) {
                if (pl[i].distance(p) < 0)
                        return OUTSIDE;
        }
        return(result);
}
```

- spheres:

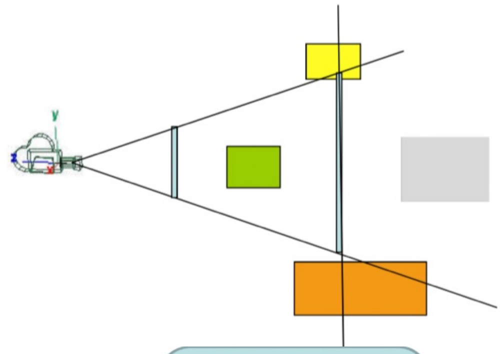
```
int FrustumG::sphereInFrustum(Vec3 &p, float radius) {
        float distance;
        int result = INSIDE;

        for(int i=0; i < 6; i++) {
                distance = pl[i].distance(p);
                if (distance < -radius)
                        return OUTSIDE;
                else if (distance < radius)
                        result = INTERSECT;
        }
        return(result);
}
```

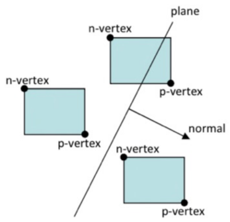
- white board: axis aligned boxes





- boxes: accept all boxes whose corners are not on the wrong side of a single plane






```
p = (xmin,ymin,zmin)
if (normal.x >= 0)
        p.x = xmax;
if (normal.y >=0)
        p.y = ymax;
if (normal.z >= 0)
        p.z = zmax;
```

```
n = (xmax,ymax,zmax)
if (normal.x >= 0)
        n.x = xmin;
if (normal.y >=0)
        n.y = ymin;
if (normal.z >= 0)
        n.z = zmin;
```

- can also be applied in:
  → clip space: let M be the modelview matrix, P the projection matrix and p a point in world space. Then A converts points from local space to clip space and p' is a point in clip space.

$\boxed{A = PM \quad p' = Ap}$

setup:

```
float M[16],P[16];

glGetFloatv(GL_MODELVIEW_MATRIX,M);

glGetFloatv(GL_PROJECTION_MATRIX,P);
```

```
glPushMatrix();

glLoadMatrixf(P);
glMultMatrixf(M);
float A[16];
glGetFloatv(GL_MODELVIEW_MATRIX, A);

glPopMatrix();
```

test:

- visible points inside the cube, centered in the origin, with dimension = 2 (coordinates after the perspective divide between -1 and 1 in all axis).
- let p be a point in world space, then $p' = (x', y', z', w') = Ap$ is a point in clip space. p' is inside the view frustum if

$$-w' < x' < w'$$
$$-w' < y' < w'$$
$$-w' < z' < w'$$

- required operations:
  - 16 multiplications + 12 additions to get the point in clip space
  - up to 6 tests (<, >) to determine if it is inside / outside

→ world / global space: let $p = (x, y, z, w)$ and $p' = Ap = (x', y', z', w')$. we know that $-w' < x' < w'$

setup:

$$A = \begin{bmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \end{bmatrix} \qquad p' = Ap = \begin{bmatrix} l_1 p \\ l_2 p \\ l_3 p \\ l_4 p \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

then in clip space if $-w' < x' < w'$ we get in world space:

$$-p\,l_4 < p\,l_1 < p\,l_4$$

if x is on the right side of the left plane then:

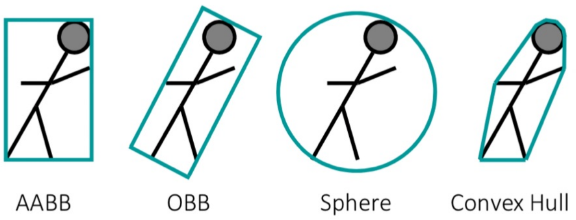
$$-pl_4 < pl_1 \iff 0 < pl_1 + pl_4 \iff 0 < p(l_1 + l_4)$$
$$\iff 0 < x(a_{11} + a_{41}) + y(a_{12} + a_{42}) + z(a_{13} + a_{43}) + w(a_{14} + a_{44})$$

The left plane is defined as $\boxed{x(a_{11}+a_{41}) + y(a_{12}+a_{42}) + z(a_{13}+a_{43}) + w(a_{14}+a_{44})}$ this way it is possible to extract the remaining planes that can be directly computed from $A = MP$.

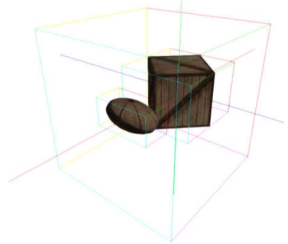
- Translation-rotation coherency: if an object is rejected by the left plane and the camera rotates to the right then the object will remain outside the view frustum. If an object is rejected by the near plane and the camera moves forward, then the object will still be outside the frustum.
- temporal coherency: store for each object the plane that caused it to be rejected. The stored plane should be the first to be tested.

Bounding volumes: a closed volume that completely contains an object or objects.

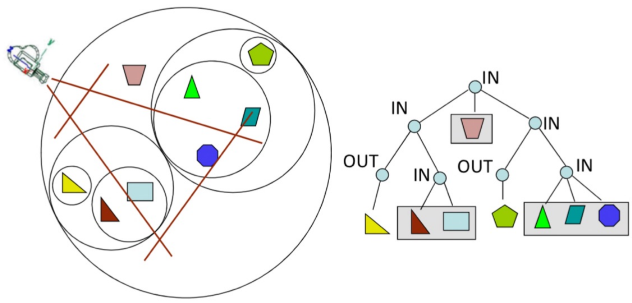
- types:

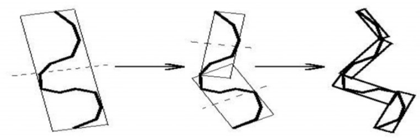


AABB = axis aligned bounding box
OBB = object aligned bounding box

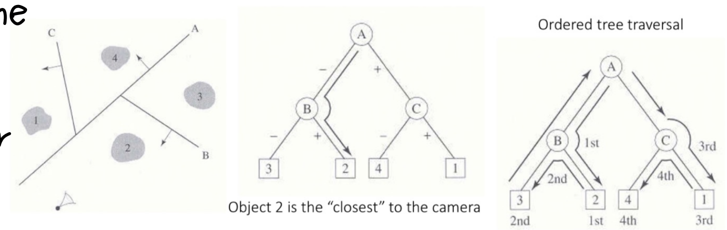


- testing the BV allows the elimination of complex geometry with simple tests when the volume is partially inside the VF:
  - → greater probability of rejection since we have less "empty space"
  - → more tests are required, potentially less triangles are drawn
- Hierarchy:





- A bounding volume based solution requires the explicit definition of objects
  object = { triangles }
- Sparse Partitioning:
  - → binary space partition (BSP): using planes to recursively split the world in two resulting in a binary tree. The planes can be arbitrary
  - → K-D trees: similar to BSP but the planes are perpendicular to the axis. To build a tree:
    - Δ pick an axis, pick perpendicular plane and split the world in 2 regions

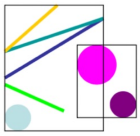


    - Δ select a different axis. Select a new perpendicular plane for a each region (may have different planes for each region)
    - Δ iterate over all axis, and then restart the process.
  - → Quad trees: divide the world recursively into quadrants. However, the recursion is not homogeneous
  - → octrees: recursively divide the world into octants with criteria to stop the subdivision - cell polygon count has reached a threshold, tree's depth is getting too large, cell is too small. If an object/polygon occupies more

than one cell - include it in the parent cell, include it in both cells and split it such that each part fits in a single cell

Bounding Volume Hierarchies

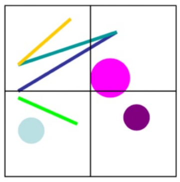
- Tightly fits objects
- Redundant spatial representation



Volumes overlap multiple objects

Space Partitioning

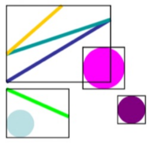
- Tightly fills space
- Redundant object representation



Objects overlap multiple volumes

Bounding Volume Hierarchies

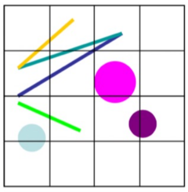
- Tightly fits objects
- Redundant spatial representation



Volumes overlap multiple objects

Space Partitioning

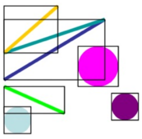
- Tightly fills space
- Redundant object representation



Objects overlap multiple volumes

Bounding Volume Hierarchies

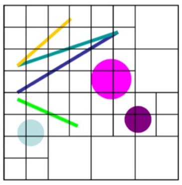
- Tightly fits objects
- Redundant spatial representation



Volumes overlap multiple objects

Space Partitioning

- Tightly fills space
- Redundant object representation



Objects overlap multiple volumes

- Hierachical Partion :
  → masking: considering an object partially inside the VF, then the child nodes must be tested. If the object is completely on the inside of a plane, then:
    - ∆ it's child nodes will also be on the inside of the same plane (the plane does not need to be tested)