Visibility culling

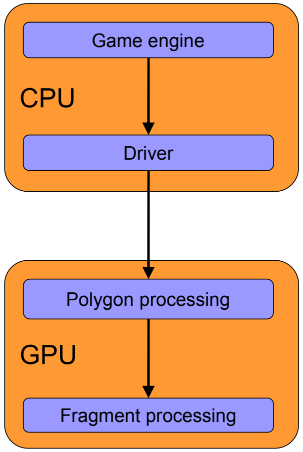
- Do not render what you do not see
- Typically only see a small part of the game world
- Why not to render everything:
  - Fillrate and polygon setup limits
  - Memory limits (CPU and GPU)
  - Bus transfer limits
- Can also be used to speed up other computations such as AI and collision detection

Culling reasons

- Part of the world may be culled for various reasons:
  - Outside the view frustum
    - Anything outside the field of view does not affect what is shown on the screen
  - Occlusions
    - Anything hidden behind something else will not be seen, so does not need to be rendered
  - Too far away
    - Limit how far you can see (e.g. fog); only render objects near the viewer

Culling stages

- Culling done at various stages in the rendering pipeline
- The earlier you cull, the bigger the savings
- But, culling earlier is often more expensive and conservative



Culling levels

- Culling can be done at various levels:
  - Fragment level: do not draw pixels that can't be seen
  - Polygon level: do not process unseen polygons
  - Object level: cull an object, for example a monster out of view or behind a wall
  - World level: cull entire groups of objects, for example an unseen room, or everything except the room the player is in

Culling levels: Fragments

- Fragment (pixel) level
  - Done at the fragment stage on the GPU
  - Clipping: do not draw outside the view frustum
    - Needs to know projected position of fragment
  - Z-buffer: do not draw behind existing render
    - Needs to know position and depth of fragment
    - Most effective when rendering front-to-back, as it prevents too much “overdraw”

Culling levels: Polygons

- Polygon level
  - Often done by the driver and GPU
  - View frustum culling
    - Do not process polygons that are entirely outside the view frustum
  - Back-face culling
    - Do not process polygons that are facing away from the camera

Culling levels: Objects

- Typically done by the game engine, as there is little concept of an object at the driver or GPU level
  - View frustum culling
    - Do not process objects that are entirely outside the view frustum
  - Hidden object culling
    - Do not process objects which are entirely hidden

Culling levels: World

- Done by the game engine
- Ignores entire regions of the game world not relevant to the current state
- Ideal for indoor environments

Visibility determination methods

- PVS: Potential Visibility Set
- Raycasting on grids
- Portals
- Quadtrees and Octrees
- BSP trees

Potential Visibility Set (PVS)

- Concept: pre-compute what is visible at any point in the game world; only draw what is visible
- Used for static parts of the game world

PVS (cont.)

- Divide the world into small regions, such as a regular grid
- Assign regions to all static geometry
- For each region, pre-compute what regions can be seen from it
  - Use one of the other visibility algorithms
- At runtime, find what region viewer is in, and process only the regions visible from the viewer's region

PVS (cont.)

- Two ways of storing visibility info:
  - For each region keep a list of visible regions
  - For each region store boolean array indicating visibility for all regions
- Method to use depends on factors like:
  - Average number of visible regions (few: list, many: boolean array)
  - Use of compression (boolean array with large blocks of 0s or 1s compresses very well)
  - Rendering algorithm, AI algorithms

PVS (cont.)

- Conservative: always renders what can be seen from a grid square, but may render what can't be seen at viewer's position
- Higher resolution grid gives more accurate visibility, but increases memory usage
- Reduce memory requirements by using compression, but decompression takes time

PVS (cont.)

```
// Position of a grid square
typedef short Coord;
struct GridPos
{
   Coord x, y;
};

// The PVS is simply an array of positions of visible grid squares
typedef std::vector<GridPos> PVS;

// Information about a grid square
struct Square
{
   bool wall;     // True if this square is a wall
   PVS pvs;       // PVS of all squares visible from this square
};
```


PVS (cont.)

```
// Compute the PVS for a given viewing position
void ComputePVSSquare(const GridPos& vp)
{
   Square& square = world[vp.x][vp.y];

   square.pvs.clear();

   // Go through all squares in the visible range and find which
   // are visible from the viewing position
   GridPos p;
   const Coord minx = max(vp.x - MAX_VISIBLE_DISTANCE, 1);
   const Coord miny = max(vp.y - MAX_VISIBLE_DISTANCE, 1);
   const Coord maxx = min(vp.x + MAX_VISIBLE_DISTANCE + 1, WORLD_SIZE-1);
   const Coord maxy = min(vp.y + MAX_VISIBLE_DISTANCE + 1, WORLD_SIZE-1);
```


PVS (cont.)

```
   for(p.x = minx; p.x < maxx; p.x++)
   {
         for(p.y = miny; p.y < maxy; p.y++)
         {
                  // Use some method to determine if p is visible
                  // from vp
                  if(IsSquareVisible(vp, p))
                           square.pvs.push_back(p);
         }
   }
}

// Compute the PVS for all squares in the world
void ComputePVS(void)
{
   GridPos p;
   for(p.x = 0; p.x < WORLD_SIZE; p.x++)
         for(p.y = 0; p.y < WORLD_SIZE; p.y++)
                  ComputePVSSquare(p);
}
```

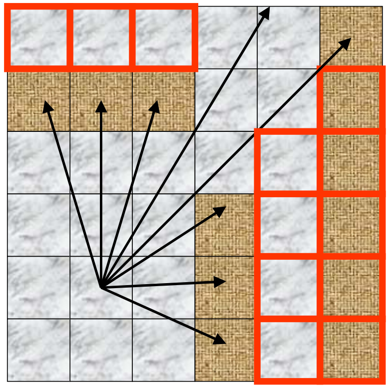
PVS (cont.)

```
void RenderWorld(void)
{
   const Square& playersquare = world[player.x][player.y];

   for(PVS::const_iterator vi = playersquare.pvs.begin();
       vi != playersquare.pvs.end(); vi++)
   {
       RenderSquare(world[vi->x][vi->y]);
   }
}
```


Raycasting on grids

- Method for games using 2D grid maps
- Determine what grid squares are visible by casting a set of rays from the viewer's grid square
- For each ray, store a "ray depth": distance to nearest grid square where the ray is blocked
- Assume anything further away is not visible

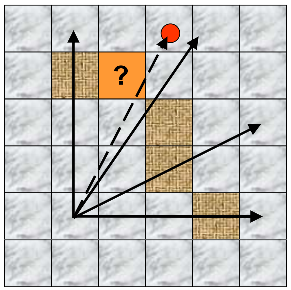


Raycasting (cont.)

- Recompute ray depths when player moves
- When rendering a grid square:
  - Determine which rays from viewer intersect the square
  - If ray depth of any of those rays is larger than distance between grid square and viewer, square is visible.

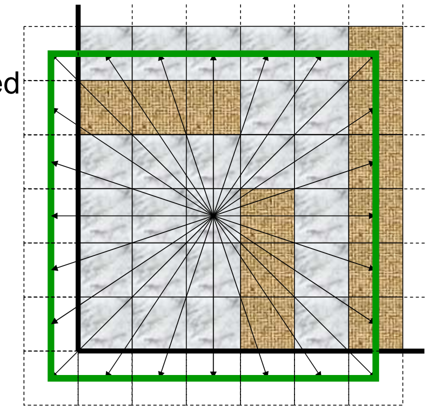
Raycasting (cont.)

- Set of rays cast must be dense enough so that all grid squares are covered
- Limit view to a maximum distance
  - Fog
  - Level design



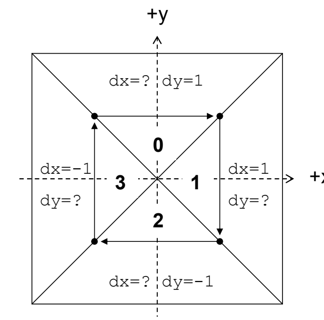
Raycasting (cont.)

- Set of rays does not need to be distributed evenly over viewing directions
- Cast rays to every grid square in an outer ring
  - Faster, easier math
  - Covers all squares



Raycasting (cont.)

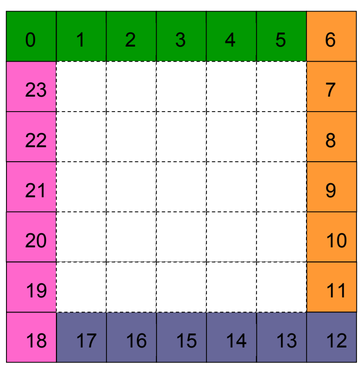
- Casting a ray:
  - Step by (dx,dy) with
    `max(fabs(dx), fabs(dy)) == 1`
  - Four sectors to consider
  - Make sure sectors don't overlap



Raycasting (cont.)

- Store ray depths in an array
- Need to enumerate the rays

```
number of rays in ring = R * 8
rays per sector = R * 2
sector = ray / rays per sector
```



Raycasting (cont.)

- Converting ray number to step direction:

```
void RayNumberToStepDirection(int ray, float& dx, float& dy)
{
   // Compute sector number with fraction
   const float sector = (float)ray / RAYS_PER_SECTOR;

        // Sector 0:  0 <= sector < 1,  -1 <= dx < 1,  dy = 1
        if(sector < 1) { dx = sector * 2 - 1; dy = 1; }
        // Sector 1:  1 <= sector < 2,  dx = 1,  1 >= dy > -1
   else if(sector < 2) { dx = 1;              dy = 3 - sector * 2; }
        // Sector 2:  2 <= sector < 3,  1 >= dx > -1,  dy = -1
   else if(sector < 3) { dx = 5 - sector * 2; dy = -1; }
        // Sector 3:  3 <= sector < 4   dx = -1,  -1 <= dy < 1
   else                { dx = -1;             dy = sector * 2 - 7; }
}
```

Raycasting (cont.)

- Converting square position to ray number:

```
int SquareToRayNumber(const GridPos& vp, const GridPos& p)
{
    // Compute square position p relative to viewer position vp
    const int dx = p.x - vp.x;
    const int dy = p.y - vp.y;

    // Compute ray number
    int ray;

    // Sector 0:  -1 <= dx/dy < 1,  0 <= ray < RAYS_PER_SECTOR
    if((dx >= -dy) && (dx < dy))
          ray = 0*RAYS_PER_SECTOR + (dx * RAYS_PER_SECTOR / dy + RAYS_PER_SECTOR) / 2;
    // Sector 1:  1 <= dy/dx < -1,  RAYS_PER_SECTOR <= ray < 2*RAYS_PER_SECTOR
    else if((dx >= dy)  && (dx > -dy))
          ray = 1*RAYS_PER_SECTOR + (RAYS_PER_SECTOR - dy * RAYS_PER_SECTOR / dx) / 2;
    // Sector 2:  -1 <= dx/dy < 1,  2*RAYS_PER_SECTOR <= ray < 3*RAYS_PER_SECTOR
    else if((dx <= -dy) && (dx > dy))
          ray = 2*RAYS_PER_SECTOR + (dx * RAYS_PER_SECTOR / dy + RAYS_PER_SECTOR) / 2;
    // Sector 3: 1 <= dy/dx < -1,  3*RAYS_PER_SECTOR <= ray < 4*RAYS_PER_SECTOR
    else
          ray = 3*RAYS_PER_SECTOR + (RAYS_PER_SECTOR - dy * RAYS_PER_SECTOR / dx) / 2;

    return ray;
}
```


Raycasting (cont.)

- Computing a ray depth map

```
// Compute the ray depth for all rays from a viewing given position
typedef int RayDepthMap[NUM_RAYS];
void ComputeRayDepthMap(const GridPos& vp, RayDepthMap& raydepthmap)
{
    // Go around the outer ring in clockwise direction
    for(int ray = 0; ray < NUM_RAYS; ray++)
    {
          // Compute ray step direction from ray number
          float dx, dy;
          RayDirectionFromNumber(ray, dx, dy);

          // Cast a ray, and store distance to the first thing hit.
          // If nothing hit, store a distance larger than anything can be
          GridPos p;
          if(Raycast(vp, dx, dy, p))
                raydepthmap[ray] = Distance(vp, p);
          else
                raydepthmap[ray] = INT_MAX;
    }
}
```


Raycasting (cont.)

- Casting a ray

```
bool Raycast(const GridPos& vp, const float dx, const float dy, GridPos& p)
{
    for(Coord s = 1; s <= MAX_VISIBLE_DISTANCE; s++)
    {
          // Compute nearest square to ray sample
          p.x = vp.x + (int)floor(s * dx + 0.5);
          p.y = vp.y + (int)floor(s * dy + 0.5);

          // If the ray falls off the end of the world, return nothing hit
          if((p.x < 0) || (p.x >= WORLD_SIZE) || (p.y < 0) || (p.y >= WORLD_SIZE))
                return false;
          // If the square is a wall, return a hit
          if(world[p.x][p.y].wall)
                return true;
    }
    // Gone beyond the visible range, return no hit
    return false;
}
```


Raycasting (cont.)

- Determining if a square is visible:

```
bool IsSquareVisible(const GridPos& vp, const GridPos& p, const RayDepthMap&
    raydepthmap)
{
    const Coord dx = p.x - vp.x;
    const Coord dy = p.y - vp.y;

    // Any squares beyond range are invisible
    if((abs(dx) > MAX_VISIBLE_DISTANCE) || (abs(dy) > MAX_VISIBLE_DISTANCE))
          return false;

    // A square is always visible from itself
    if((dx == 0) && (dy == 0))
          return true;

    // Check if the distance to the square is no more than the ray distance
    int ray = SquareToRayNumber(vp, p);
    return Distance(vp, p) <= raydepthmap[ray];
}
```

Raycasting (cont.)

- Rendering

```
void RenderWorld(void)
{
   // Compute the ray depth map for the current view
   // Really only needs to be done when the player moves
   // to a different square
   RayDepthMap raydepthmap;
   ComputeRayDepthMap(player, raydepthmap);

   // Go through all squares in the visible range and find which
   // are visible from the viewing position
   GridPos p;
   const Coord minx = max(player.x-MAX_VISIBLE_DISTANCE, 1);
   const Coord miny = max(player.y-MAX_VISIBLE_DISTANCE, 1);
   const Coord maxx = min(player.x+MAX_VISIBLE_DISTANCE+1, WORLD_SIZE-1);
   const Coord maxy = min(player.y+MAX_VISIBLE_DISTANCE+1, WORLD_SIZE-1);
```


Raycasting (cont.)

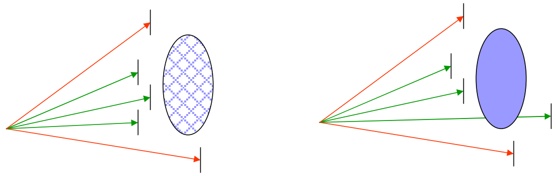
```
   for(p.x = minx; p.x < maxx; p.x++)
   {
         for(p.y = miny; p.y < maxy; p.y++)
         {
                  if(IsSquareVisible(vp, p, raydepthmap))
                           RenderSquare(world[p.x][p.y]);
         }
   }
}
```


Raycasting (cont.)

- Potential problems:
  - False hits at oblique angles
  - Visibility may change as player moves across square
- Possible solution: check neighbouring squares
  - If any neighbours are visible by raycasting, consider square as visible

Raycasting (cont.)

- Can also be used to cull large objects
  - Determine which rays intersect the object
  - If ray depth of any of those rays is larger than distance between viewer grid square and object, object is visible. Otherwise, cull object.



Raycasting (cont.)

- Possible improvements:
  - Interpolate from view position to outer ring in `Raycast()` instead of stepping (see online code)
  - Only cast rays within the field of view
  - Use Manhatten distance:

    `abs(p.x – vp.x) + abs(p.y – vp.y)`

    - Real distance not needed, only relative order
    - Manhatten distance gives order close to L2 order
  - Use Bresenham line drawing algorithm instead of rounding position to the nearest square

Raycasting (cont.)

- Used in the Cube engine to compute visibility at each frame

`http://wouter.fov120.com/cube/`