Visibility Processing

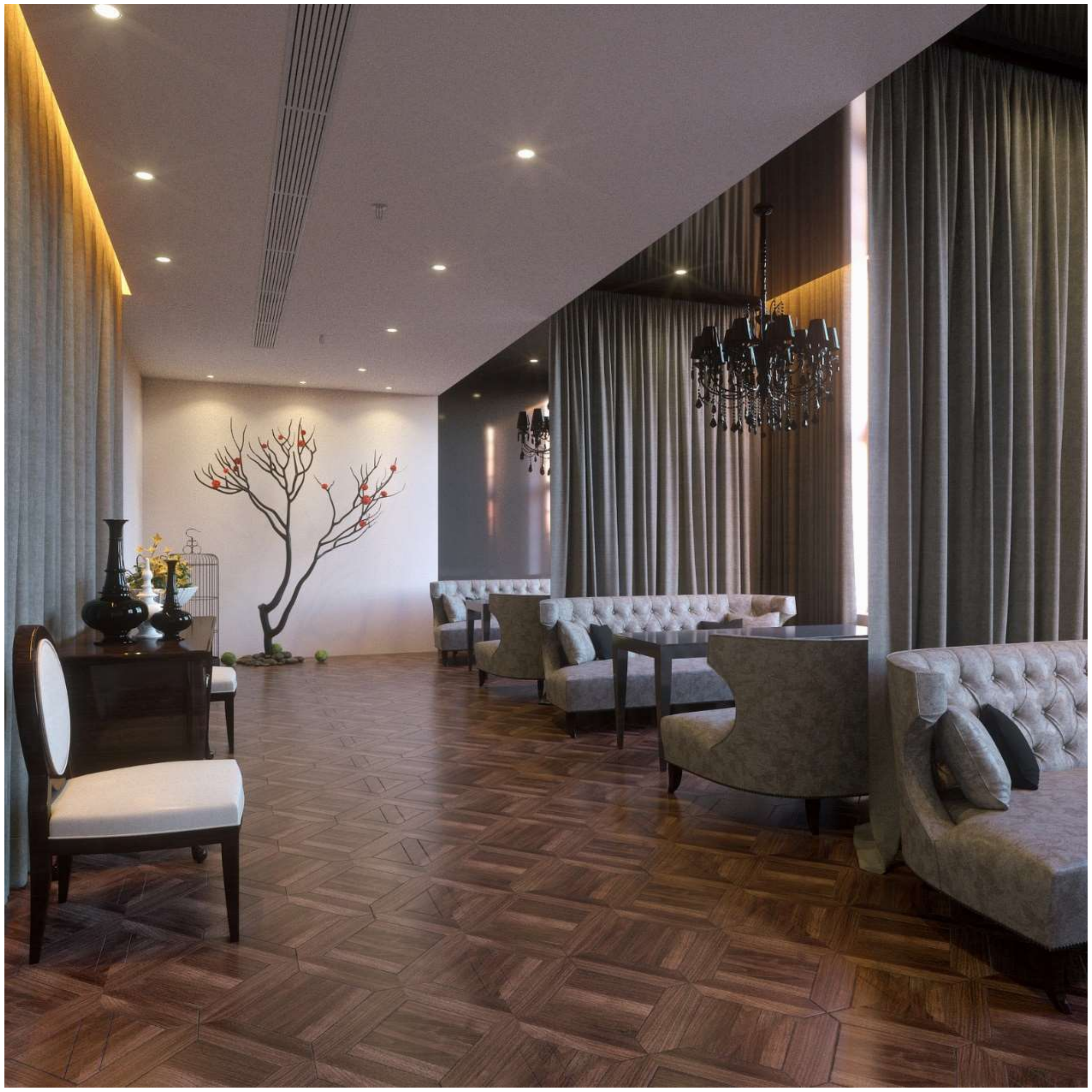
CSCI4120 Principle of Computer Game Software

Outline

- Visible surface determination
- Indoor consideration
- Outdoor consideration

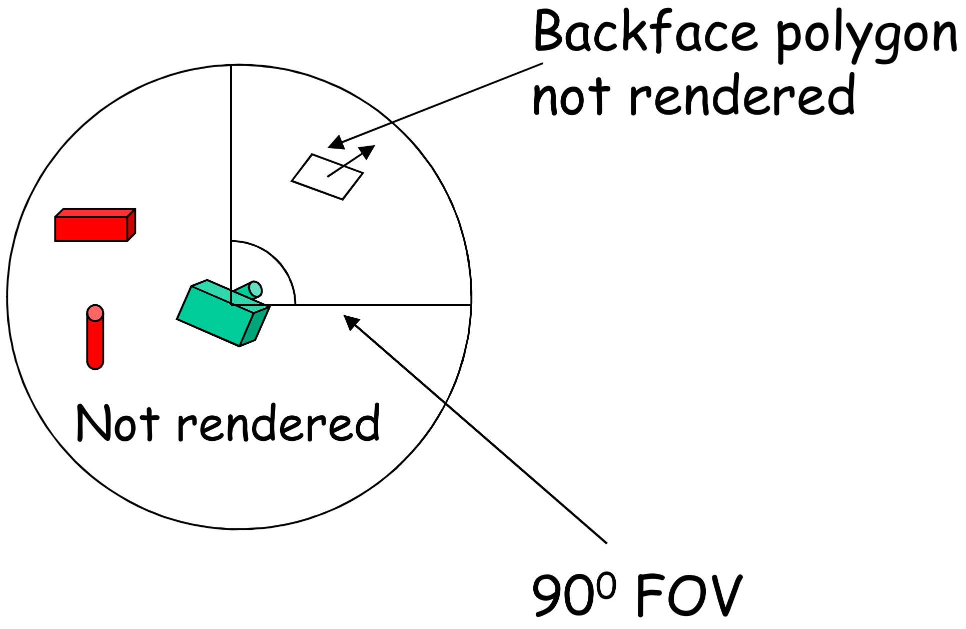
Indoor Rendering

- Seems like not a very tough problem
- (1600x1600) This still shot takes around 20 min on GTX680
- Major consumer of rendering effort is lighting calculation



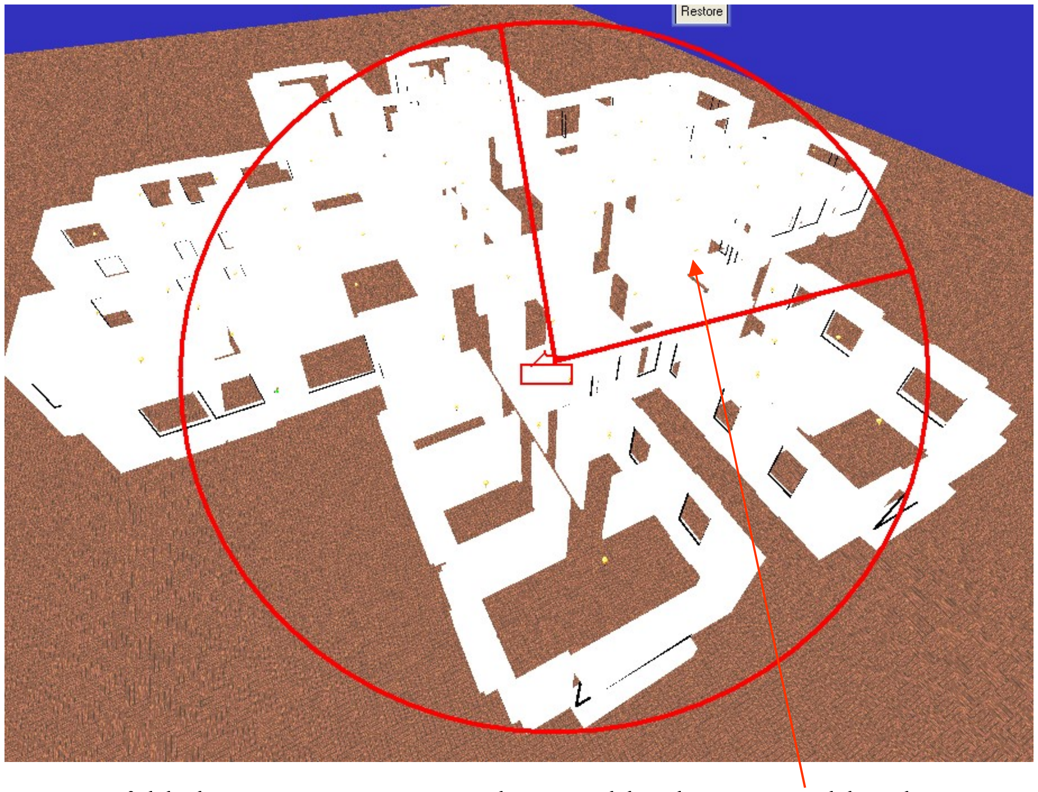
Rendering

- Clipping : Camera with 90^0 will clip around three quarters of the global geometry
- Culling: ½ geometry is back facing, culled away



Indoor Rendering

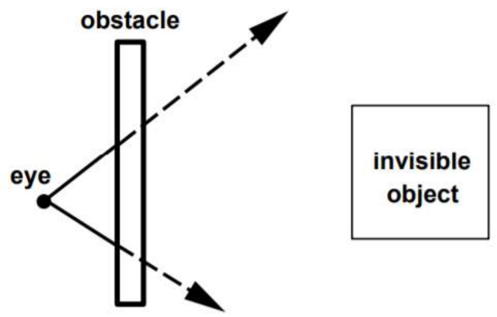
- The surviving 1/8 geometry will be rendered i.e. within the viewing frustum.
- We still want to render *only* those *visible* polygons i.e. those camera can see



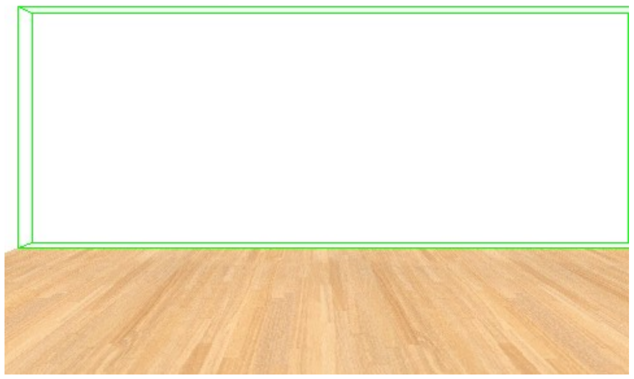
Still have to render all the walls here

Indoor rendering

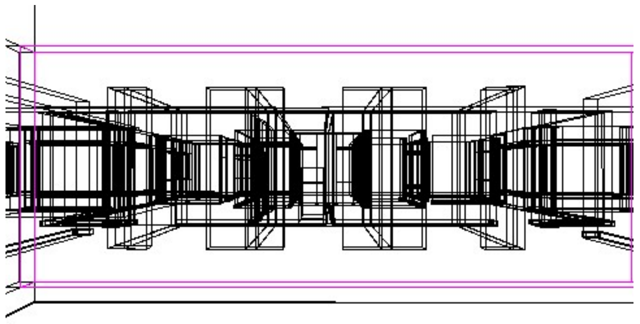
- Clipping & culling can now mostly performed by hardware



- Indoor rendering would be faster if *occlusion culling* can be performed by taking advantage of walls which are occluders



A camera facing a wall



Need not render those polygons behind the wall

Idea of Occlusion Test

- $\mathcal{O}$(num_triangles^2) if simply testing each triangle against all the others
- $\mathcal{O}$(num_object^2) if testing each object against all the others
- Can be further reduced by partitioning the set into potential occluder and occluded triangle
- Viewing distance can be used to further reject the outlier

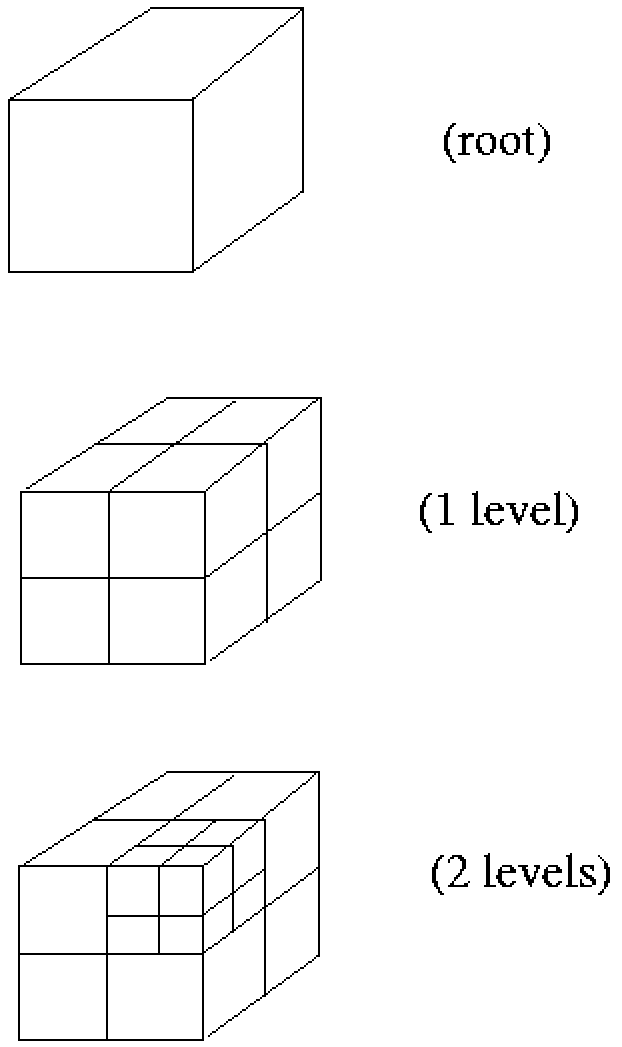
Indoor rendering

- Most games take place inside building
- Use spatial subdivision hierarchies to help answer this question:
  if the viewing frustum intersect with an object/mesh/polygon?

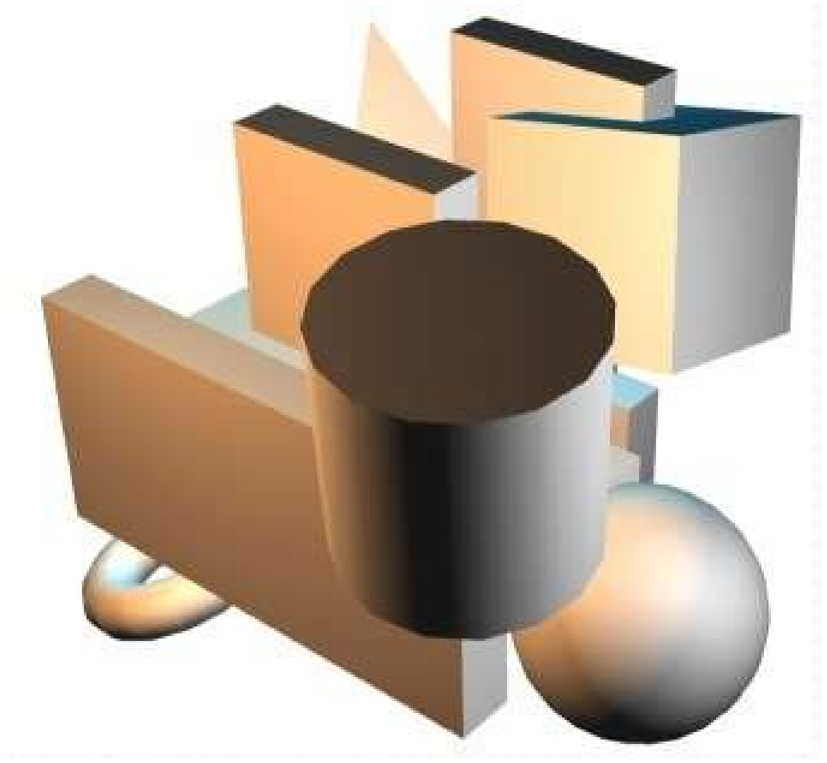
- Popular approaches(CPU based):
  1. Quadtree/Octree
  2. Binary Space Partitioning(BSP) with potential visibility set(PVS)
  3. Portal

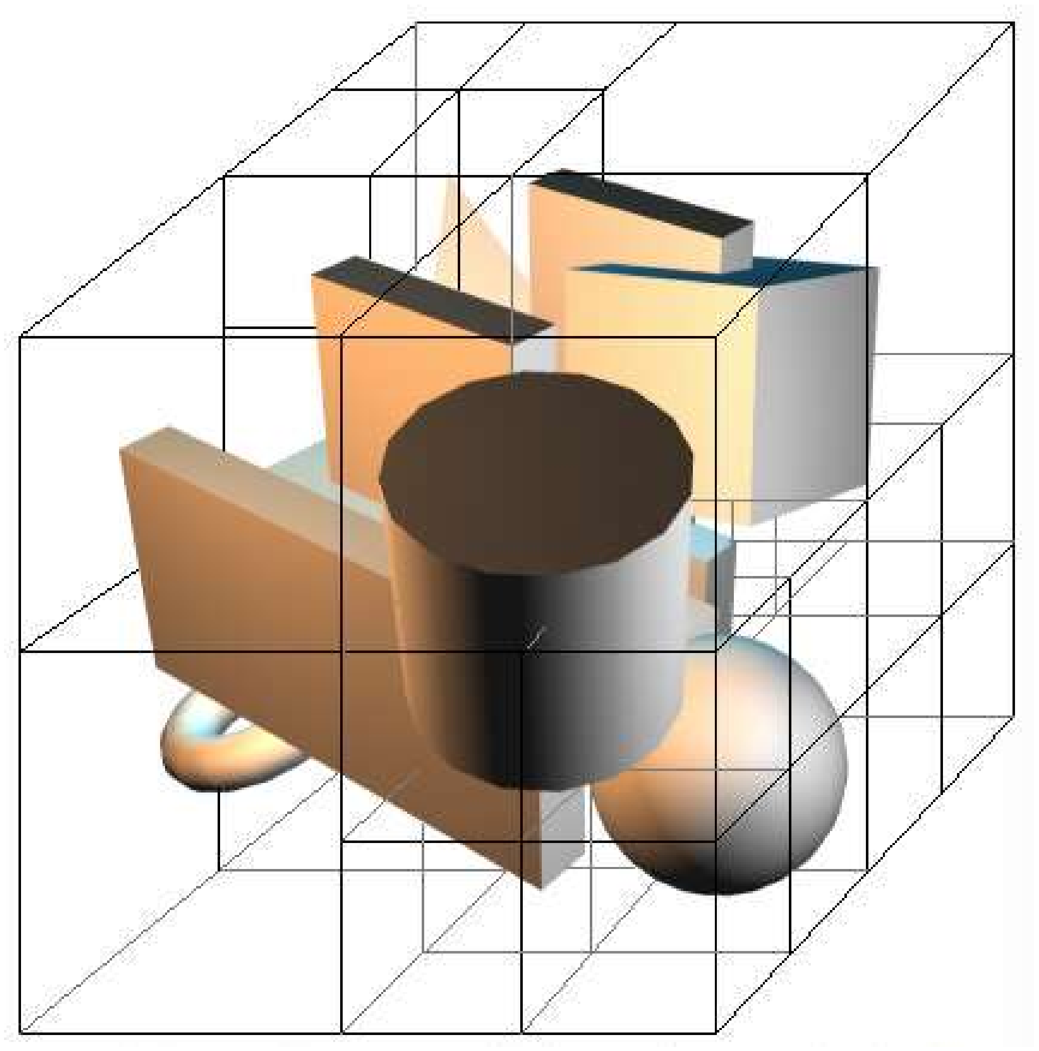
Octree

- Divide the space into 8 sub-regions
- Each node representing a region will thus have 8 children
- Any sub-node can thus be decomposed further, which depends on some measures e.g. no. of polygons



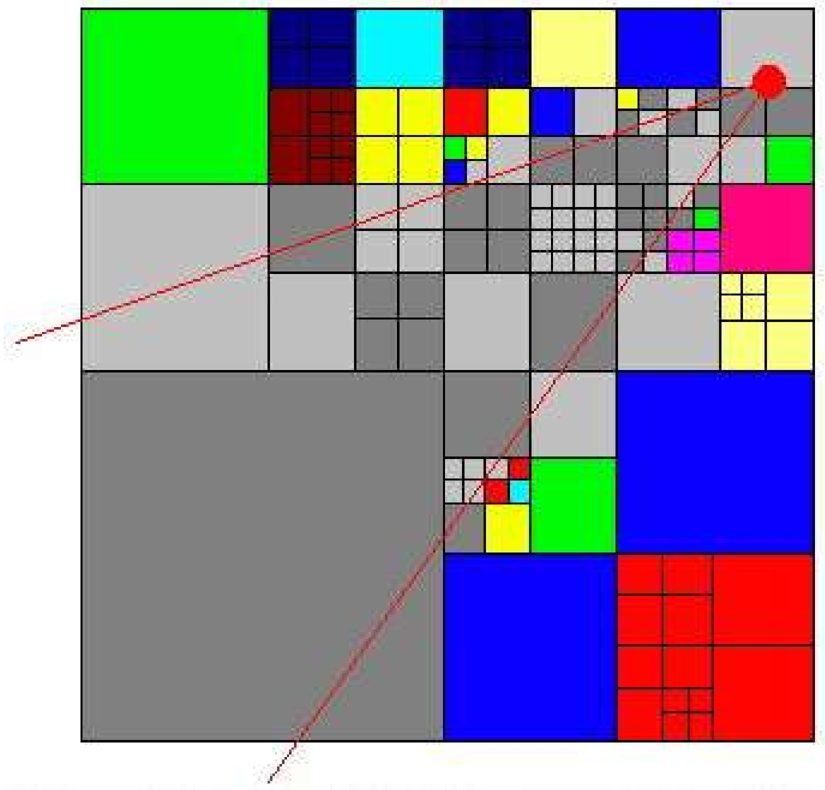
Octree





Octree

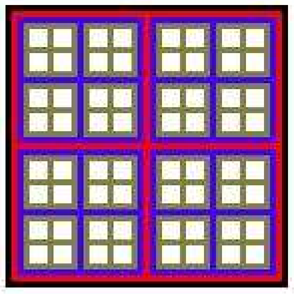
- Easy to code – using recursion
- Quickly reject lots of polygons (*colored on right*)
- Useful in collision detection and shadow as well



An octree from the top and a viewing frustrum

Quadtree

- Single layer version of *Octree*
- Having 4 child nodes for every parent node
- Concept similar to *Octree*
- Suitable for terrain rendering



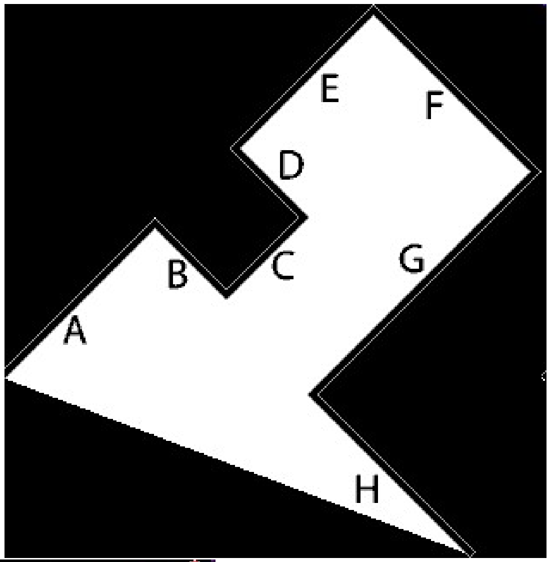
Binary Space Partitioning

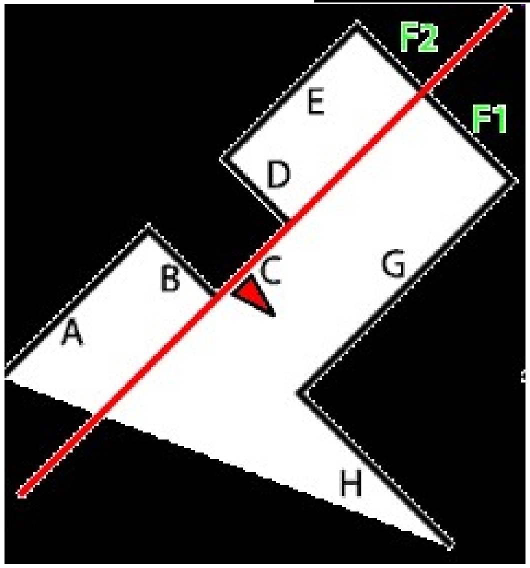
- Not suitable for modern rendering, but useful in other areas eg. collision detection
- Advantage – view point *independent* polygon *ordering*
- 2 phases:

1. Tree generation (*next slide*)
2. Tree traversal – given viewpoint, will return the list of triangles ordered by Z-value by tree traversal i.e. *back-to-front*

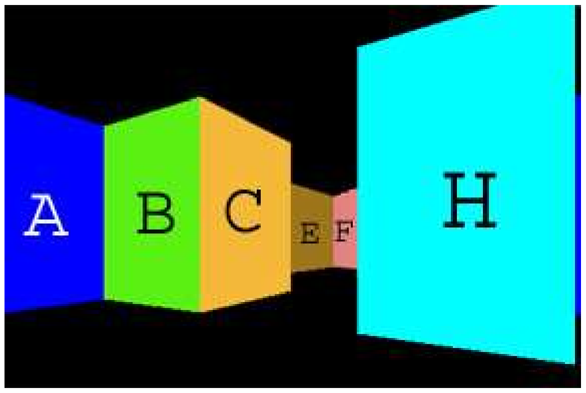
BSP Tree Generation

1. *Select* a partition *hyperplane*.
2. *Partition* all polygons with the initial hyperplane, storing them in the front and back polygon list accordingly.
3. *Recurs* with each of the front and back polygon list.

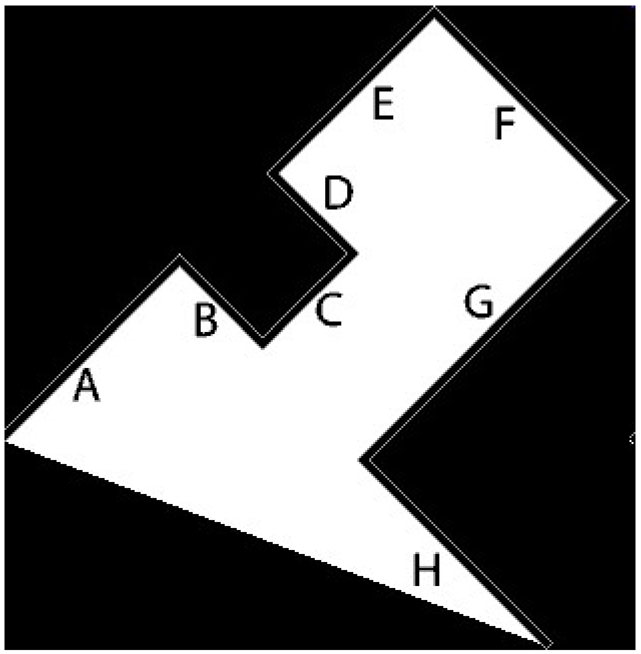




BSP Tree example



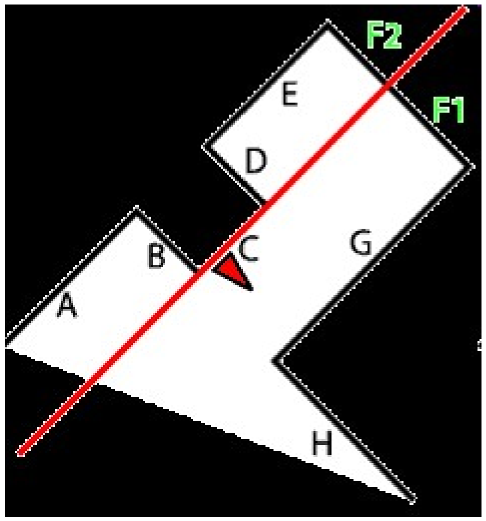
Camera view



Top down view

BSP tree example

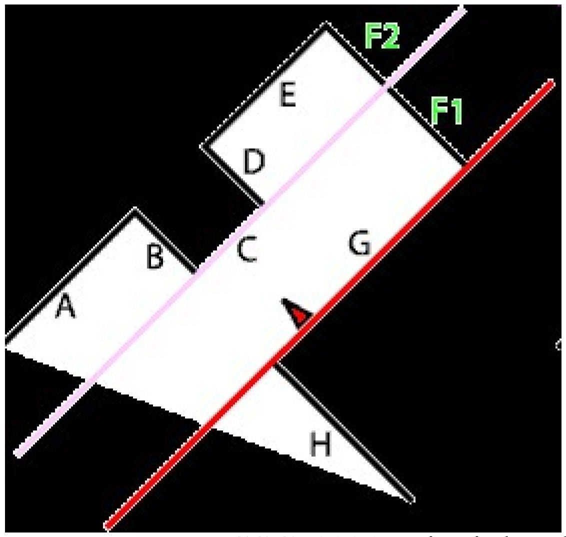
First split plane



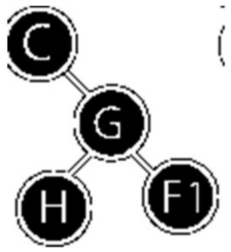
in front of C: { G, H }
in back of C: { A, B, D, E }
C splits: { F => F1, F2}

First (root) node

Second split plane



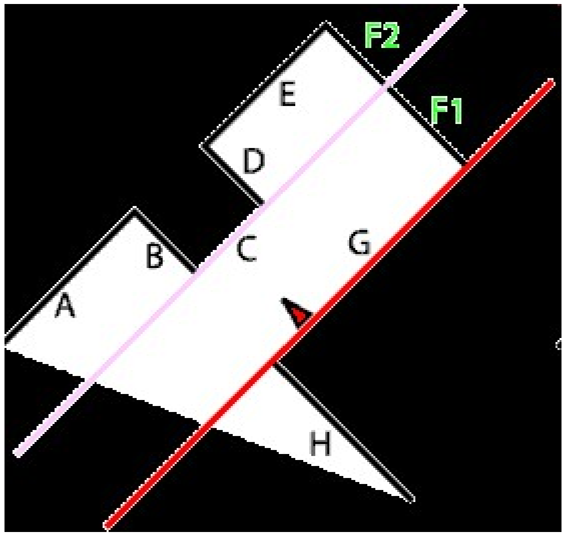
BSP tree



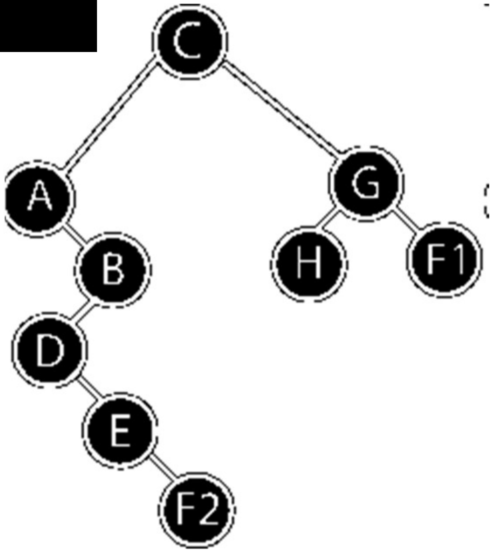
BSP tree example

- Consider {A,B,D,E,F2}, assuming split plane A is chosen, final BSP tree may like right below .

- Note special situation of same split plane E, we are making it pseudo in front of A here!



Final BSP tree



BSP tree

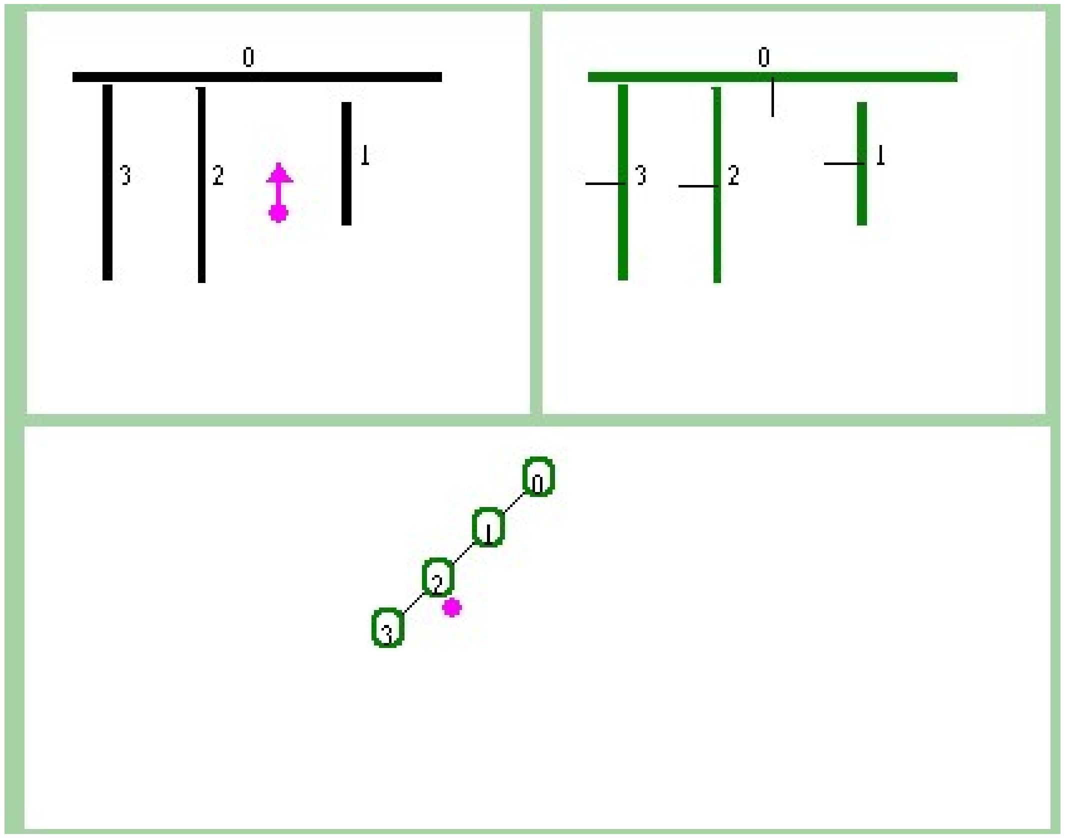
- Choice of partitioning plane is critical in :
  1. *Balance* of tree
  2. Doubling up of polygons in case of any *split*
- Determine the best plane in each partitioning during compilation phase
- Set up cost functions such as

$$score(p) = abs(front\_faces - back\_faces) + f(no\_of\_splits)$$

e.g. $f(no\_of\_splits) = \mathrm{no\_of\_splits} * 8$

Balance of tree

- If the choice of partition plane on right is *0,1,2,3* the BSP tree created will be unbalanced.



BSP tree Rendering

Camera position

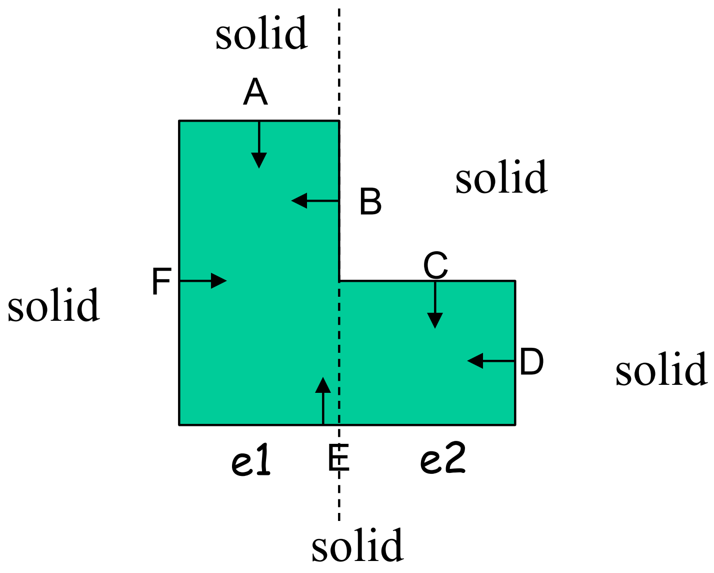
```
void drawBSPTreeBacktoFront(node)
{
    if (node->isLeaf) return;
    int result = classifyPoint(pos, node->splitter);
    if (result==R3P_FRONT)
    {
            if (node->back!=NULL) drawBSPTreeBacktoFront( node->back);
            // draw this polygon
            drawPolygon(node->polygon);
            if (node->front!=NULL) drawBSPTreeBacktoFront(node->front);
            return;
    }
    if (node->front!=NULL) drawBSPTreeBacktoFront( node->front);
    if (node->back!=NULL) drawBSPTreeBacktoFront( node->back);
    return;
}
```

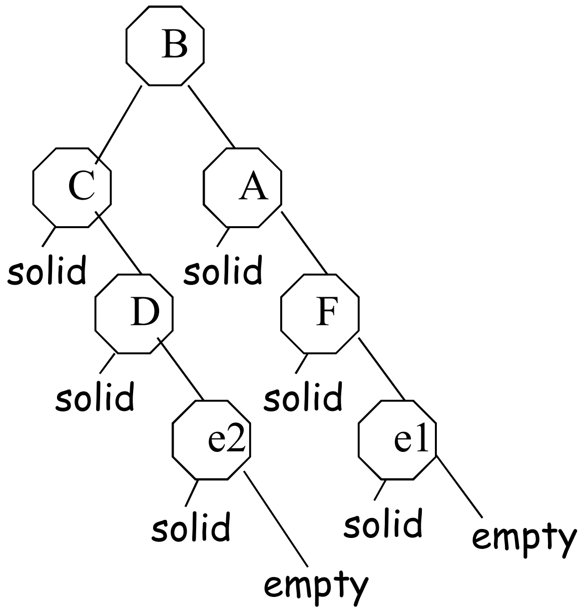
Advantage of BSP

- Can emanate polygons in Z-sort order in linear time
- Rendering back to front is useful when display card has no hardware Z-buffer
- With little change, can render in front to back
- Provide collision detection and lighting calculation information

Solid Leafy BSP Tree

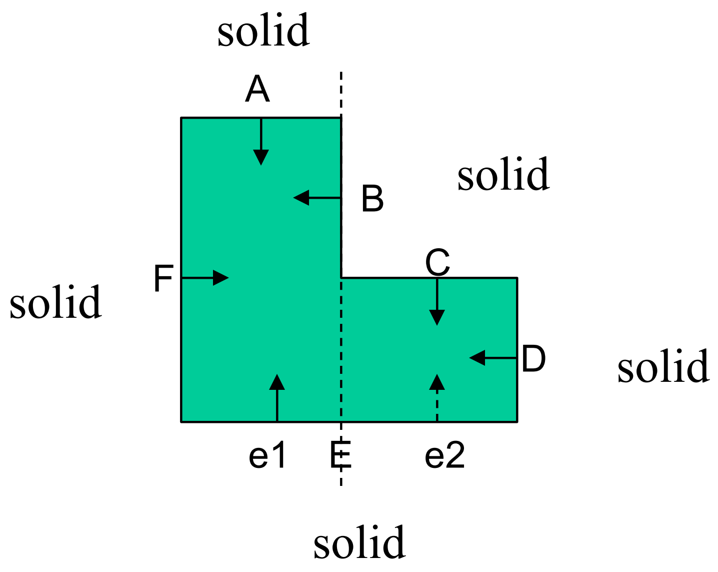
- A solid BSP tree can provide additional information of solid space for use in *collision detection*

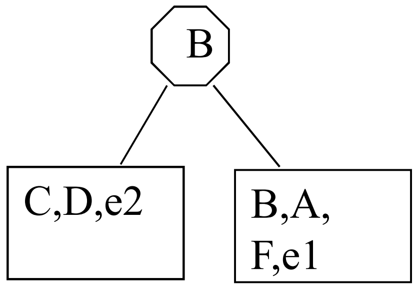




Solid Leafy BSP Tree

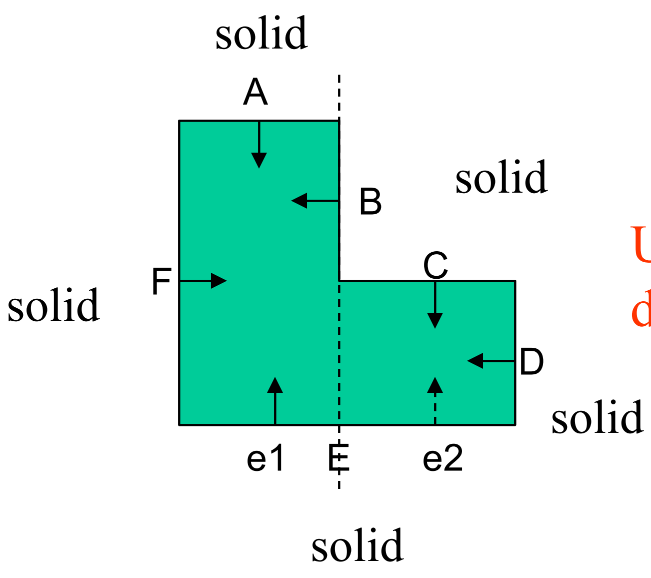
- A leaf in the tree contains the polygons forming the convex hull of an empty space
- Polygons inside each cell may be rendered in any order with back face culling

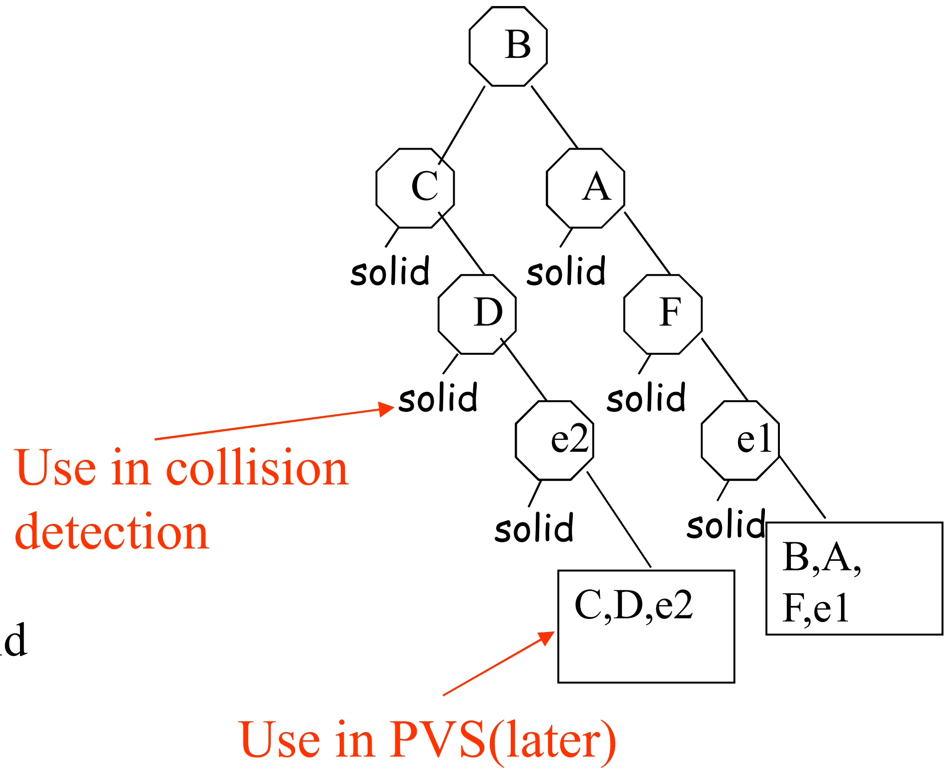




Solid Leafy BSP Tree

- Combining the two to have all the advantages



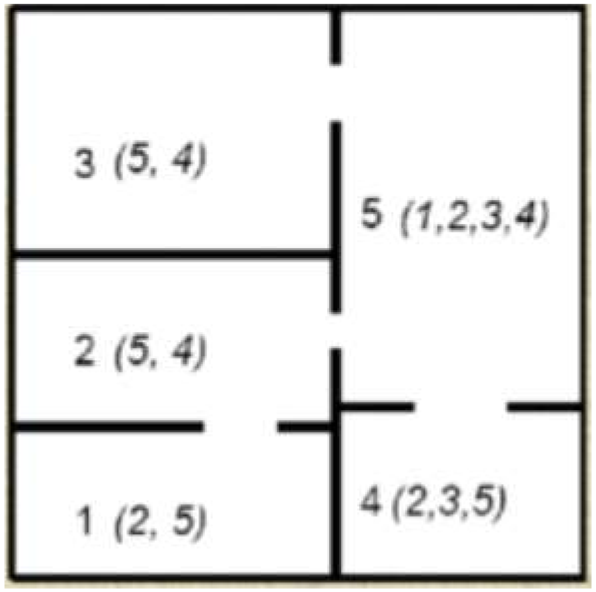


Problems of BSP

- Indoor environment – high complexity and high occlusion => BSP cannot take advantage
- BSP did not cope well with complex interior models – single BSP tree for many rooms results in many polygons drawn without being seen on screen (*overdraw*)
- Seek ways to prune away unseen polygons

Potential Visibility Set(PVS)

- An array that store for each *leaf* the visibility information of all other leafs from that leaf e.g. assume 128 leaves level, if PVS[8] of leaf 100 equals to 1, then cell 8 is visible to cell 100.



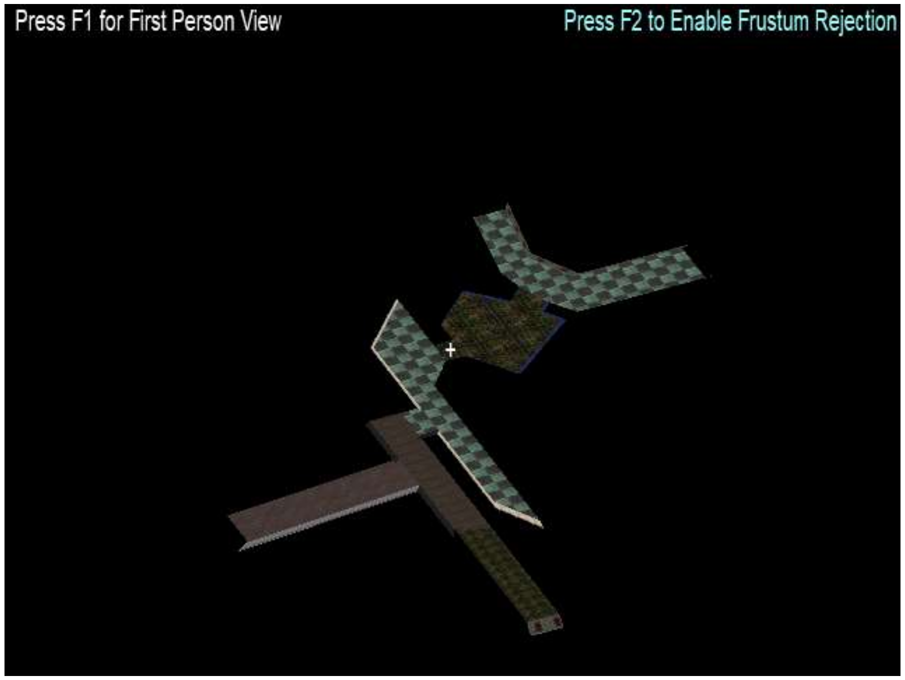
- Unseen polygons can then be skipped before any processing need be done by looking up the PVS

- Much overdraw can then be avoided

Rendering with Solid leaf PVS

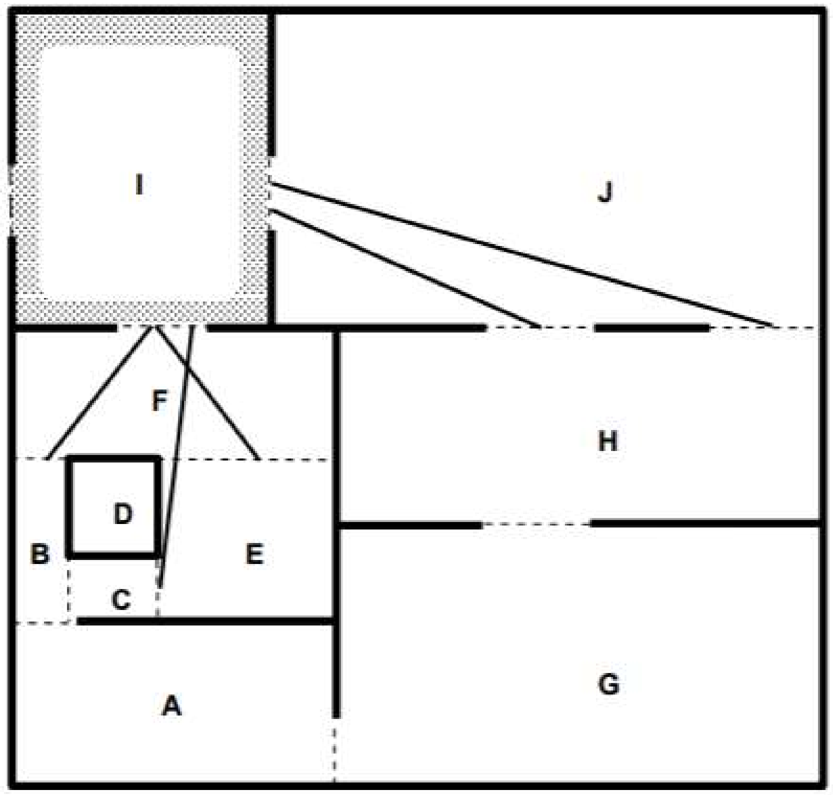
- For a given camera position
  - Traverse the BSP tree to find which leaf the camera in
  - Loop through that leaf's PVS data
  - Render leaf that is visible from within the current leaf i.e. *PVS[] is 1*, with Z-buffer on (no depth sort information)

- Still cannot take full advantage of modern hardware acceleration features


Press F1 for First Person View
Press F2 to Enable Frustum Rejection

PVS calculation

- The gates/portal(different from portal rendering) between leafs are checked against visibility
- A stab tree is created for each leaf
- Cell to cell visibility is determined by *depth first search* (DFS) on the tree
- The concepts still useful in modern day occlusion handling



https://people.csail.mit.edu/teller/pubs/siggraph91.pdf

Portal Stabbing Algorithm

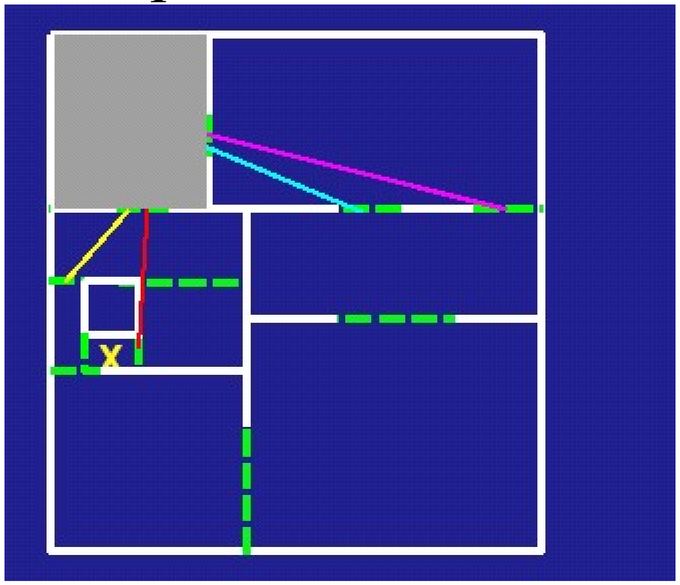
- Assume existence of Stabbing_Line(P) where P : Portal sequence
  determine either a stabbing line for P or no such lines
- All cells visible from C find with recursive routine below

```
Find_Visible_Cells (cell $C$, portal sequence $\mathbf{P}$, visible cell set $\mathcal{V}$)
    $\mathcal{V} = \mathcal{V} \cup C$
    for each neighbor $N$ of $C$
        for each portal $p$ connecting $C$ and $N$
            orient $p$ from $C$ to $N$
            $\mathbf{P}' = \mathbf{P}$ concatenate $p$
            if Stabbing_Line ($\mathbf{P}'$) exists then
                Find_Visible_Cells ($N, \mathbf{P}', \mathcal{V}$)
```

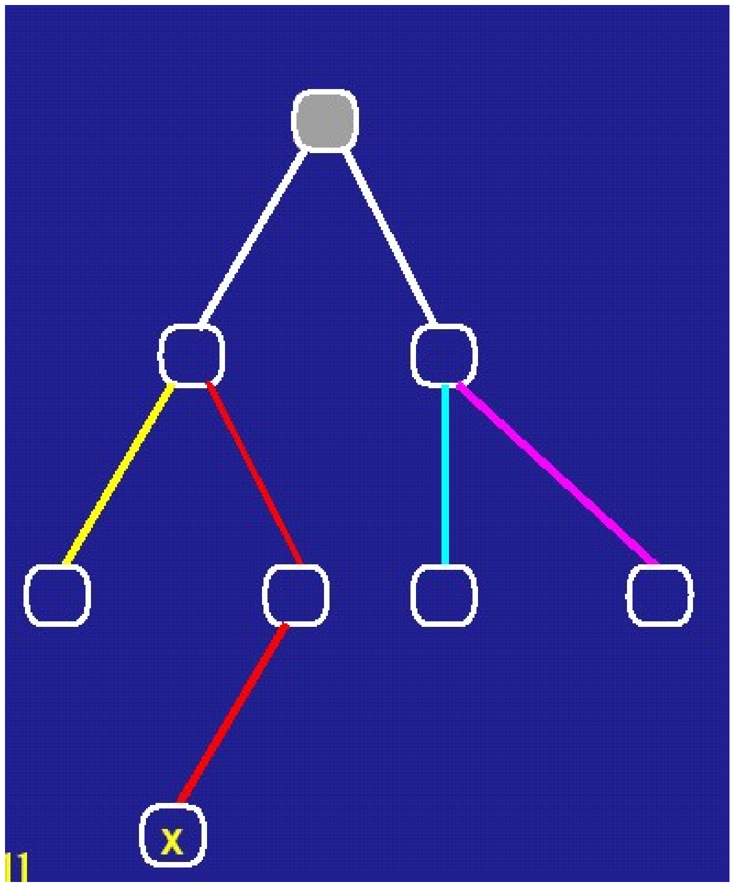
Portal Stabbing

- Tree node = leaf cell
- Tree edge = stabbed portal

Top view of level

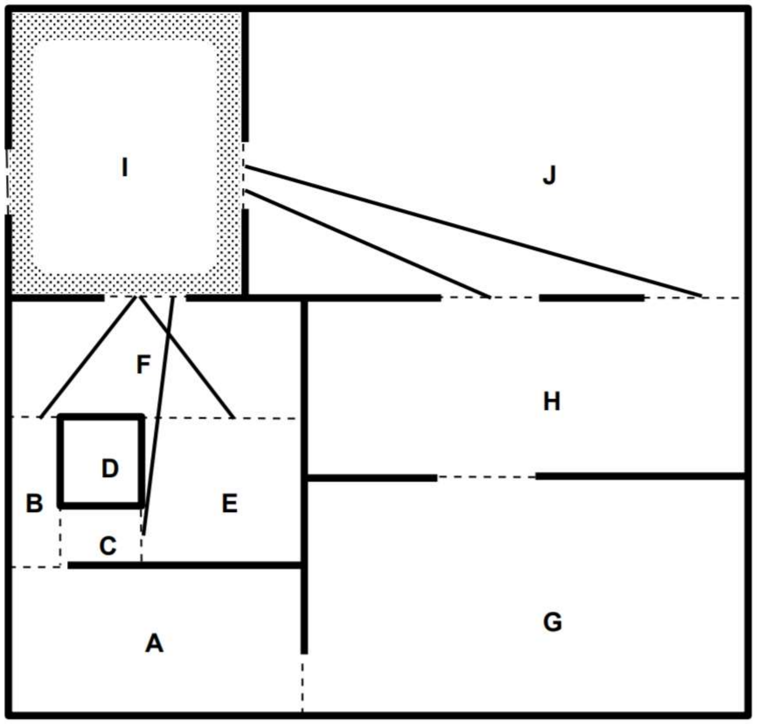


Stab tree



Portal Stabbing

- Find_Visible_Cells(cell I, P=*empty*, $\mathbb{V}=\emptyset$)



Invocation stack

$Find\_Visible\_Cells\ (I, \mathbf{P} = [\,], \mathcal{V} = \emptyset)$

$\quad Find\_Visible\_Cells\ (F, \mathbf{P} = [I/F], \mathcal{V} = \{I\})$

$\quad\quad Find\_Visible\_Cells\ (B, \mathbf{P} = [I/F,\ F/B], \mathcal{V} = \{I, F\})$

$\quad\quad Find\_Visible\_Cells\ (E, \mathbf{P} = [I/F,\ F/E], \mathcal{V} = \{I, F, B\})$

$\quad\quad\quad Find\_Visible\_Cells\ (C, \mathbf{P} = [I/F,\ F/E,\ E/C], \mathcal{V} = \{I, F, B, E\})$

$\quad Find\_Visible\_Cells\ (J, \mathbf{P} = [I/J], \mathcal{V} = \{I, F, B, E, C\})$

$\quad\quad Find\_Visible\_Cells\ (H, \mathbf{P} = [I/J,\ J/H_1], \mathcal{V} = \{I, F, B, E, C, J\})$

$\quad\quad Find\_Visible\_Cells\ (H, \mathbf{P} = [I/J,\ J/H_2], \mathcal{V} = \{I, F, B, E, C, J, H\})$

Portal Rendering

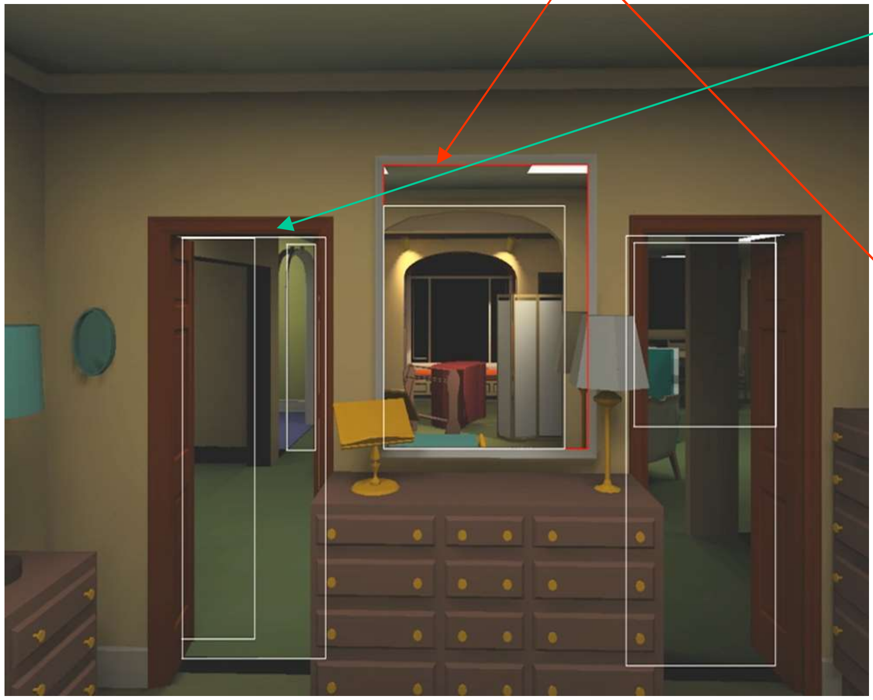
- One of popular rendering algorithms
- Level is designed with connected rooms (sectors)
- Each room is connecting to the other through a portal
- Each room should store with its bounding volume

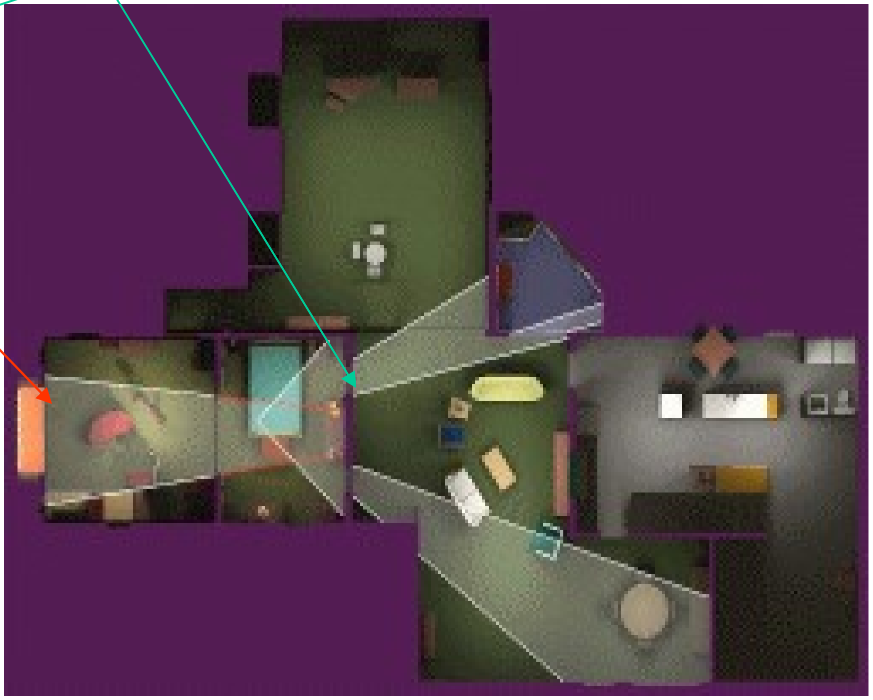
- Portal rendering need not pre-compute visibility
- Can take advantage of display hardware acceleration

Portal

cull boxes for portals (white) and mirrors (red).

Overhead view, showing portal culling frustums active





Portal Rendering

- Pseudo code:

```
detect room of current camera position
render the room
locate any portals
for each portal
    clip geometry through the portal;
    Render survived geometry into portal region;
    recurse through any portals in this region;
```

Advantage – Immediate reduction of overall geometry into single or small number of rooms

Optical Effects using Portal

- Reflections & translucency easily implemented

```
switch (portal_type)
case REGULAR:
    paint destination room;
case REFLECTIVE:
    calculate virtual camera using support plane of portal
    Invert viewing matrices
    Paint destination room
    Paint portal using alpha blending
case TRANSLUCENCY:
    Paint destination room
    Paint portal using alpha blend
```

Hybrid Approaches

- BSP require *static* scene geometry in general
- Portal techniques has problem in handling room with *huge* number of triangles
- Can combine *octree* with *portal* : portal connecting room walls only, with interior content in octree

Hybrid Approaches

- Quadtree-BSP useful in large areas to be explored

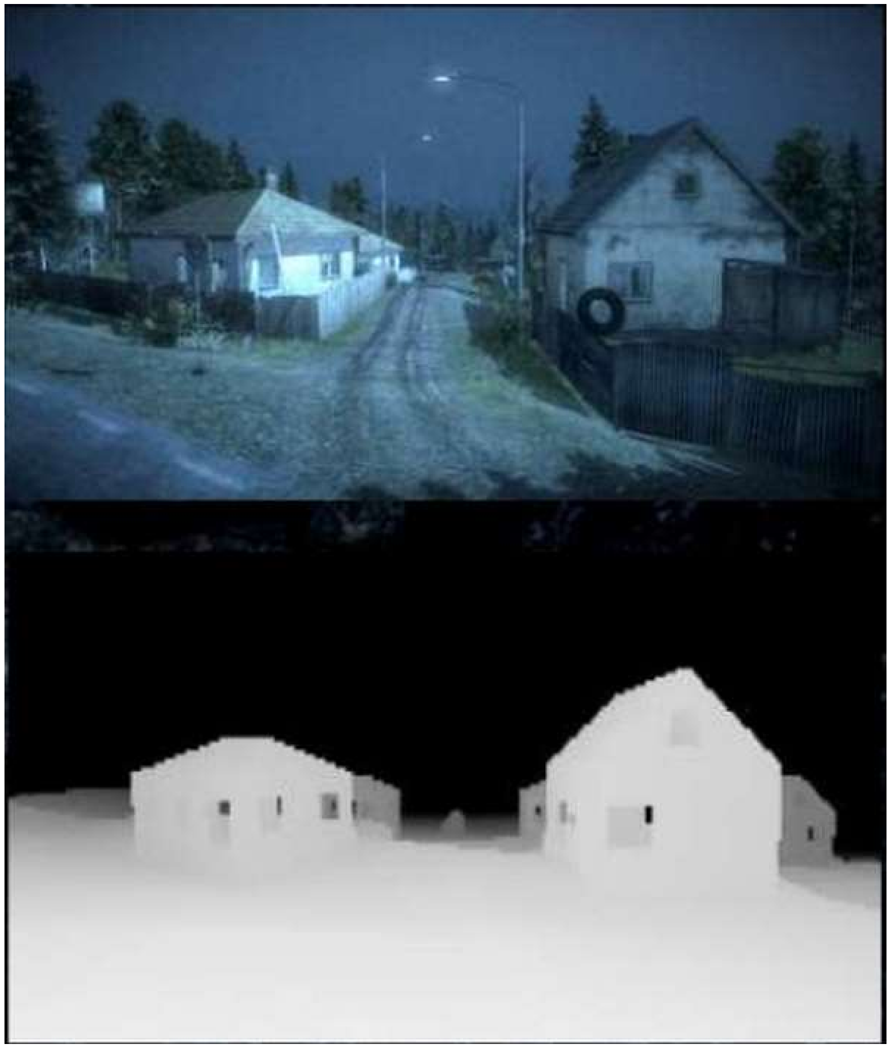
- BSP break a level down to triangle level, resulting in large tree & long traversal

- Quadtree detect quickly where we are
- BSP refine the location details to help in collision detection

Hardware Assisted Occlusion Tests

- New generations cards support hardware occlusion test
- Sending the bounding box to the hardware to check whether update in Z-buffer results

```
For each object
    activate occlusion query
    send bounding box
    deactivate occlusion query
    if pixels were modified
        render object
```



Similar technique used in Frostbite

Hardware Assisted Occlusion Tests

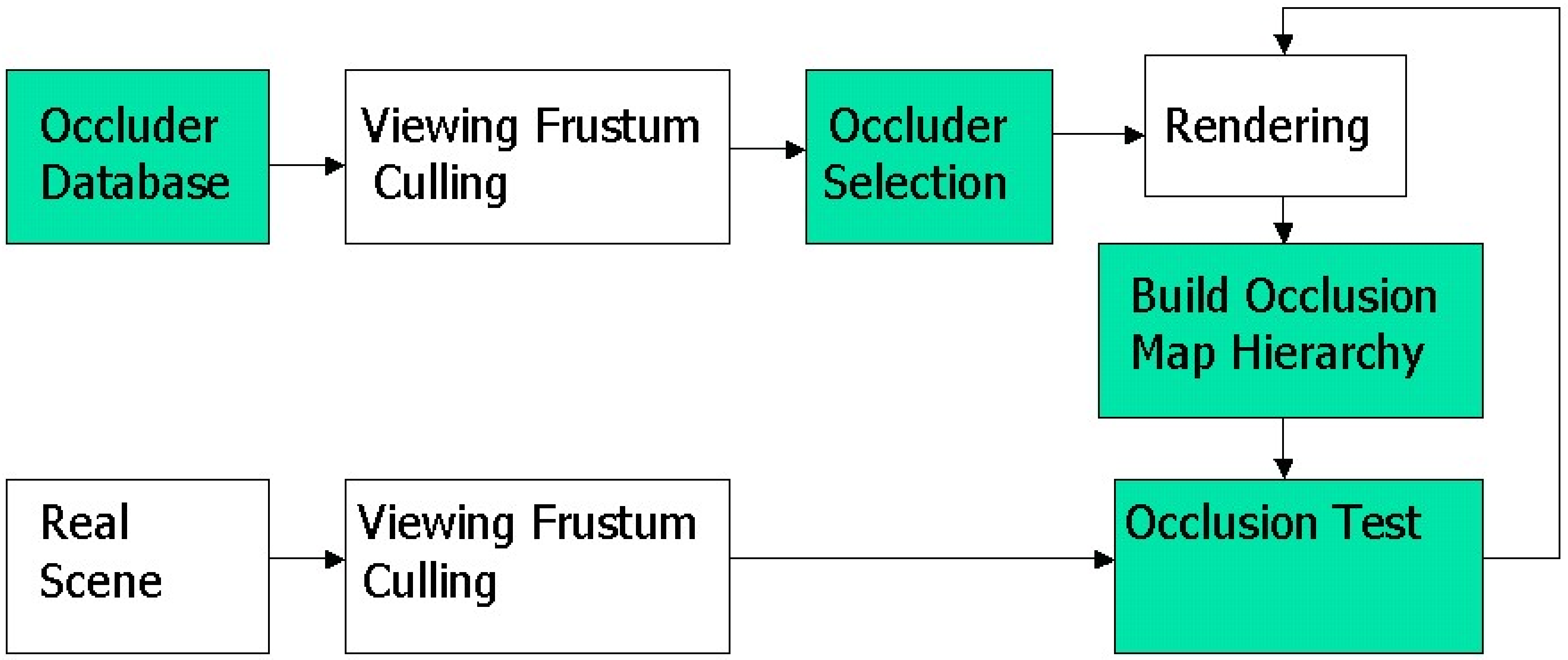
- speed up using front to back testing
- Further speed up by hierarchical culling(Quadtree)

```
Sort the four nodes by distance to viewer
for each node
    if subnode is not empty
        if subnode is not clipped
            activate occlusion query
            paint bounding box
            deactivate occlusion query
            if pixels were modified
                paint object
```

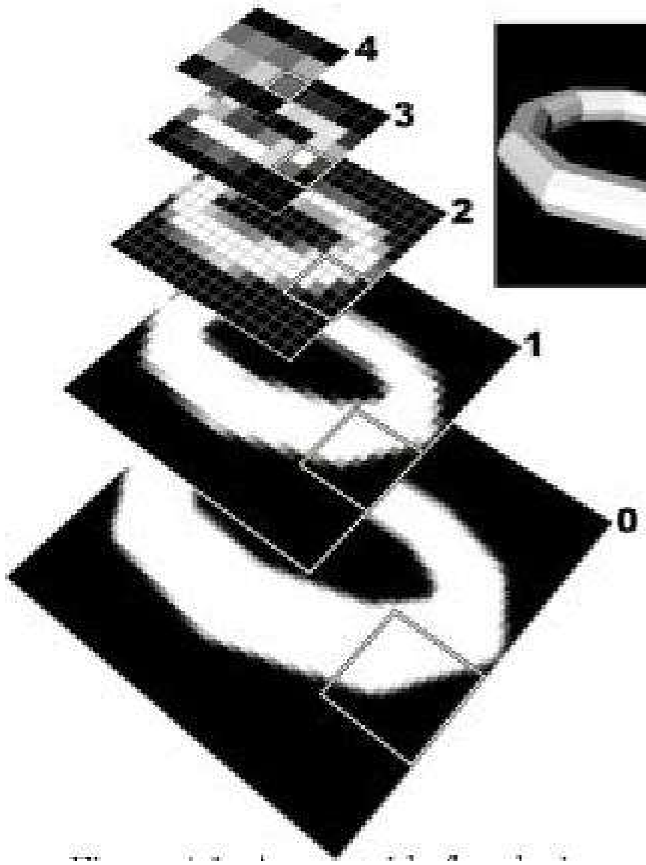
Hierarchical Occlusion Map (HOM)

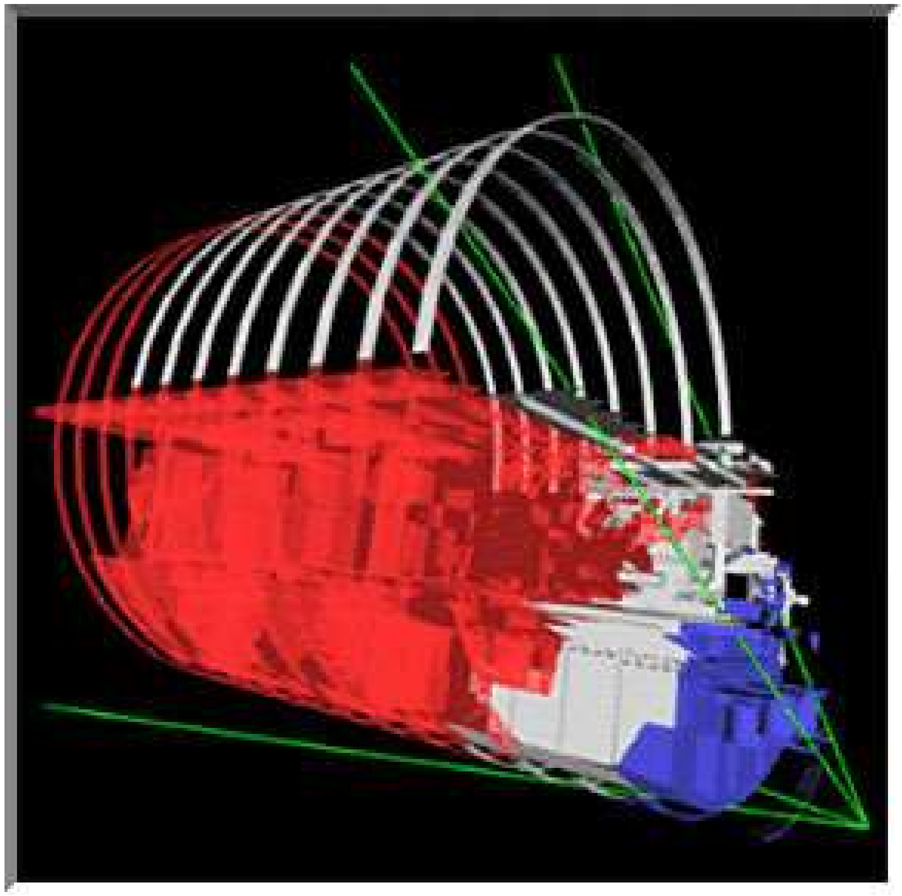
- Run-time visibility determination
- Choose a set of graphics objects from the scene as occluders
- Use the occluder to define occlusion map(hierarchically)
- Compare the rest of the scene against the occlusion map

Algorithm


Occluder Database
Viewing Frustum Culling
Occluder Selection
Rendering
Build Occlusion Map Hierarchy
Real Scene
Viewing Frustum Culling
Occlusion Test

HOM





- Blue: Occluders
- Red: Occludees

Summary

- Modern display card can process millions of triangle per frame at 30 FPS, affecting the visibility algorithm being used
- Cost of pruning individual polygon is getting too high when compare with efforts needed to render them
- Modern game engines typically use various visibility algorithms together to render a scene

Summary

- Clipping, culling and occlusion handling is vital in indoor scene rendering
- Hardware assisted culling will also affect the techniques chosen
- Most problems are now being handled, e.g. *lighting, movable geometry, destructible geometry* etc, albeit not under a single framework
- To a certain extent, the indoor rendering technologies are quite stable right now

Further reading

- Core Techniques and Algorithms in Game Programming – *Ch.13*
- 3D Games – Realtime Rendering and Software Technology Volume 1 – *Ch.9*
- BSP being used in Unreal engine http://wiki.beyondunreal.com/Legacy:BSP_Tree
- Portal Rendering http://www.flipcode.com/archives/Building_a_3D_Portal_Engine-Issue_01_Introduction.shtml

Outdoor rendering

- Talking about large viewing distance
- Clipping & culling still be used
- Few occluders in outdoor scene
- Level of details is the key to efficient rendering

Terrain Rendering

- A smooth switching from low resolution to high is expected for exploring in these area
- Large data set is also expected
- *Triangle budget* i.e. a fixed number of triangles, will be assumed for each frame

Data Structures

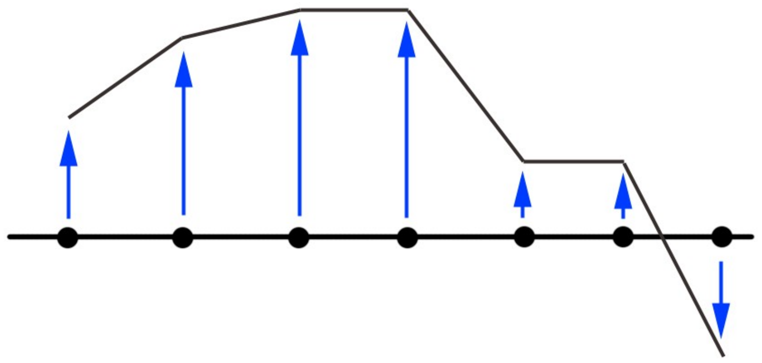
- Mostly represented in height field i.e. 2D array of heights e.g. 256x256 bitmap with 8 bit color representing 1x1 km of 512 m max height have scale (4,4,2)

- Can convert easily to *quadtree* by placing mesh grid on top of height field

- Grids are also suitable for triangle strips & indexed primitives rendering

Height field representation

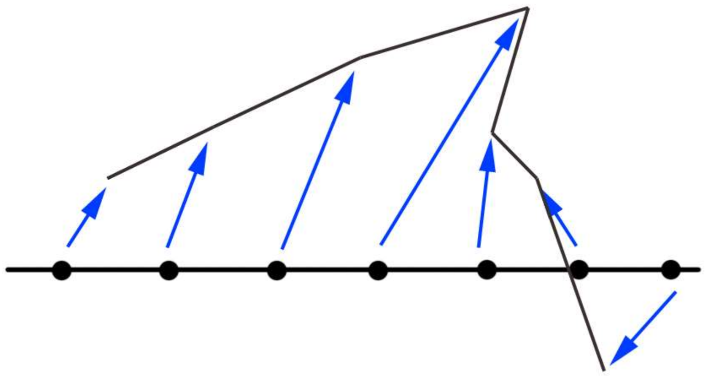
- A typical height field representation

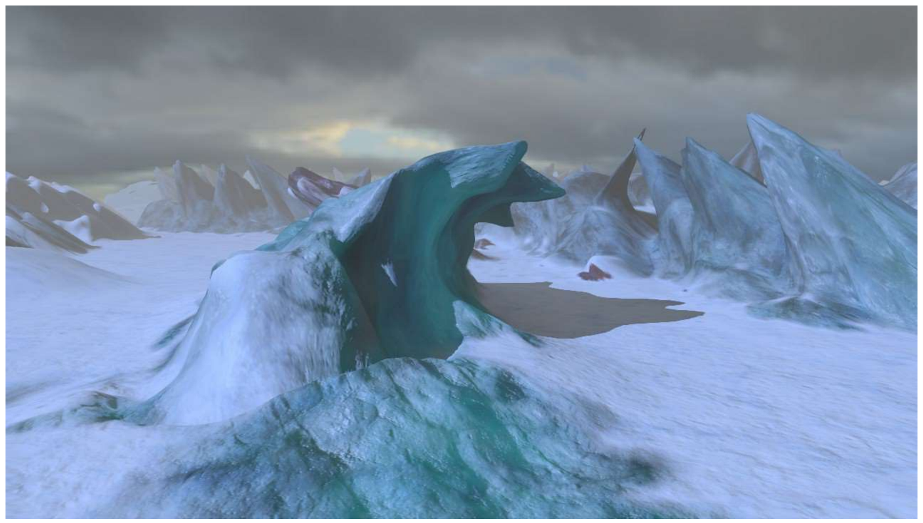


- Disadvantage:
  - hole structure e.g. arch is not possible with this representation alone
  - Not well suited for LOD modeling

Vector Field

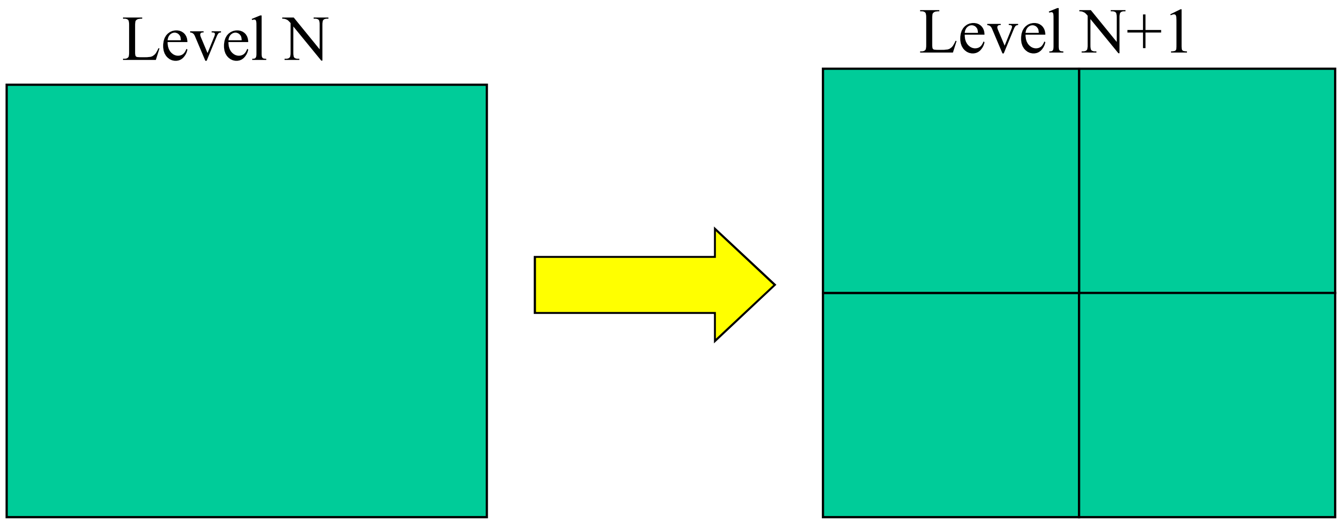
- Enable more different terrain structures such as Overhangs / vertical faces





Quadtrees

- 4-ary trees that subdivide each node into four subnodes



Quadtrees

- Can be adaptive that expand nodes only when more details is found

- *Metric of details* can be the variance between real data and a quad at this place

$$\sigma 2=\frac{1}{N}\sum_{i=1}^{N}(x_i-\overline{x})2$$

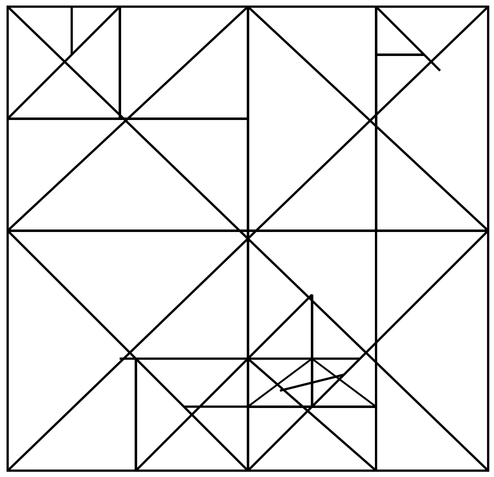
- Larger variance imply more detail

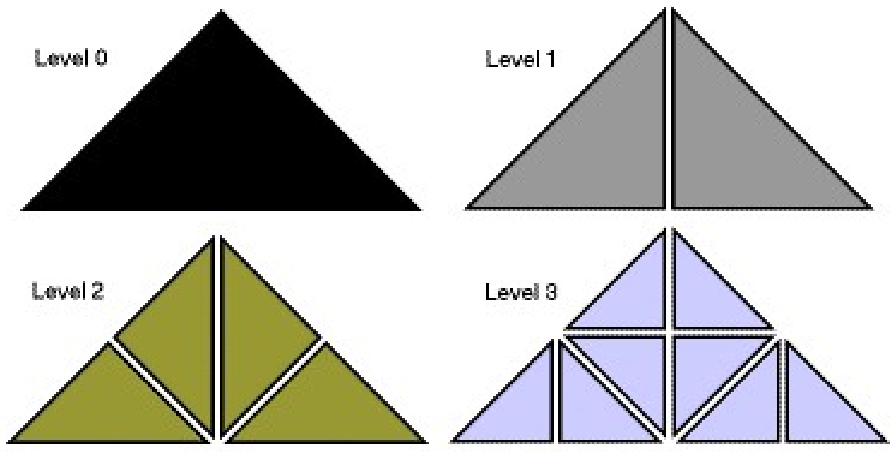
Quadtrees

- The terrain geometry is broken down into rectangular block of geormetry eg
- during runtime
  - Renderer select the maximum depth (of quadtree) based on distance to player and the detail in the mesh
  - Select coarser representation for distant elements & maximum detail for nearby objects

Binary Triangle Trees(BTT)

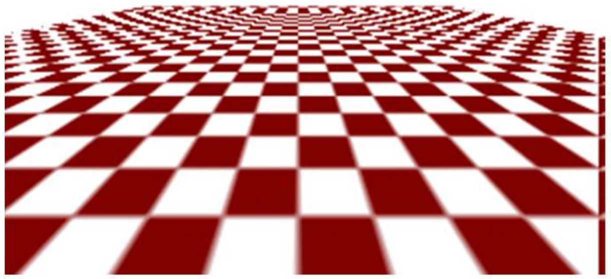
- Adaptive structure that grows where more details are present
- Different in that triangles used instead of quads – divided into two new triangles



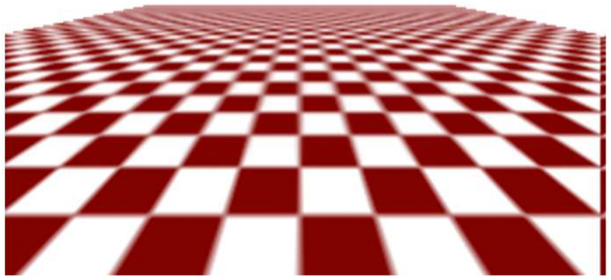


Geomipmapping

- **Mipmapping** : texture mapping technique aimed at improving visual quality of *distant* textured primitives
- Texture mipmap
  - Pre-computing a series of scaled-down texture maps – mipmaps ($\frac{1}{2}, \frac{1}{4} ..$ of original size)
  - Selecting appropriate mipmap texture by distance to viewer



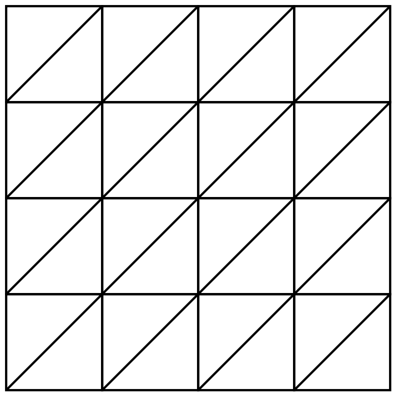
No mipmap



mipmap

Geomipmapping

- *Geomipmaps* – mipmaps on terrain geometry
- Same concept with *seams handling* needed
- Terrain size must be a power of 2
- A terrain block contains 4x4 mesh



Terrain block
(32 triangles total, 5x5 vertices)

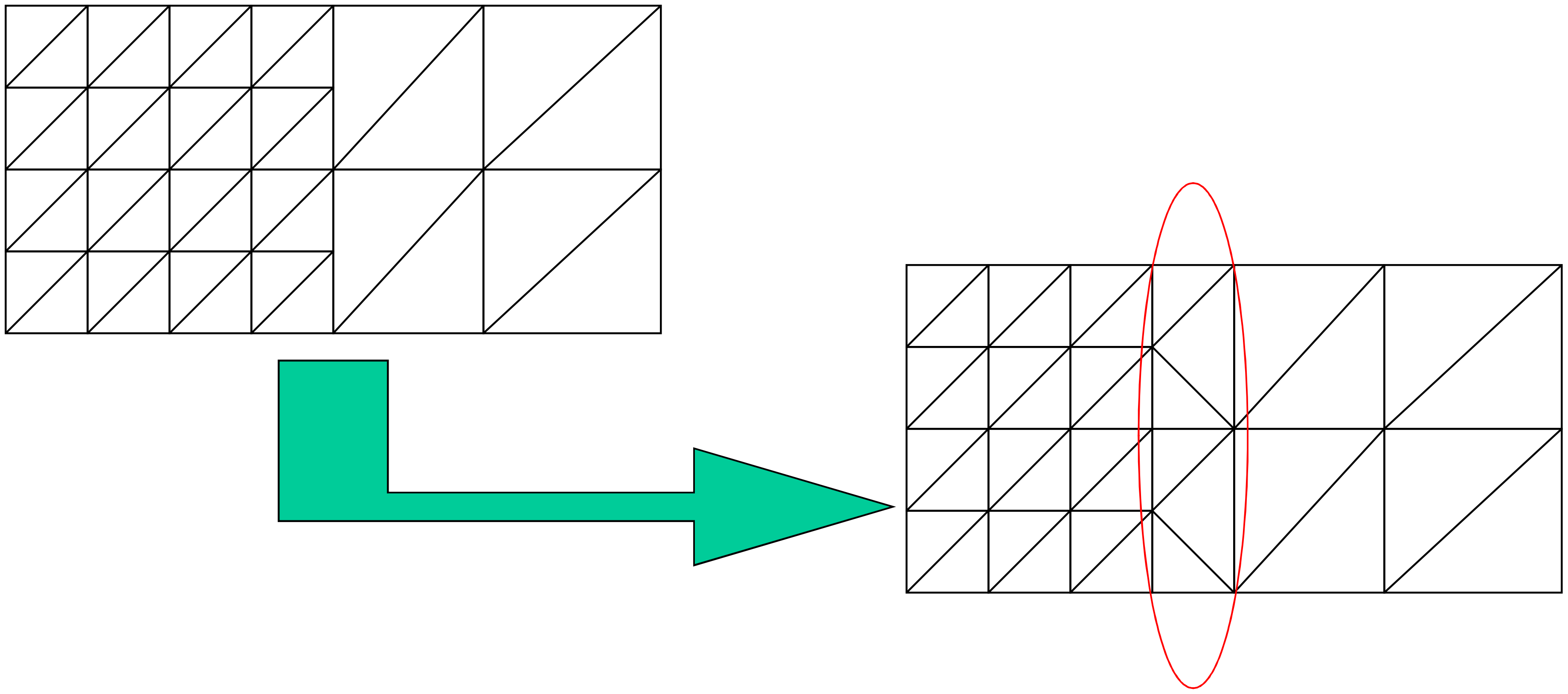
Geomipmapping

- Each node contains its bounding box
- 4 descendants => 4 sub-quadrants
- Until a node contains exactly a terrain block
- Depth traverse the tree & perform hierarchical clipping
- Distant triangles must still be rendered!
- When reach the leaves, decide which geo-mipmap level to render
- Metric: quantity of detail & distance to viewer

- Two issues:
  1. Gaps between two blocks with different resolution
  2. Change in details easily perceived

Seamless gap between blocks with different resolution

- Alter the connectivity of higher detail mesh



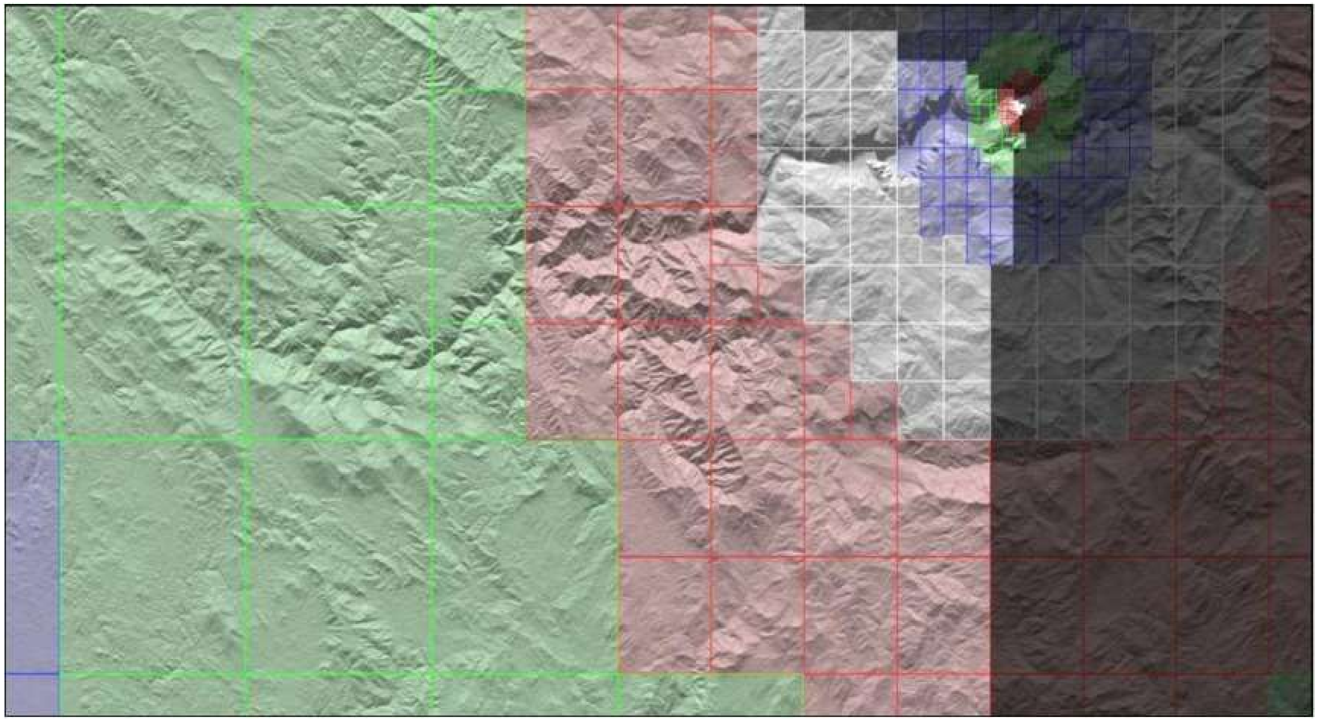
Sudden Pops(change in resolution)

- In recent approach CDLOD (Continuous distance-dependent LOD by Strugar 2010), no stitching geometry is used

- Continuous morphing between LOD levels
- higher level mesh is completely transformed into lower level before switching occurs

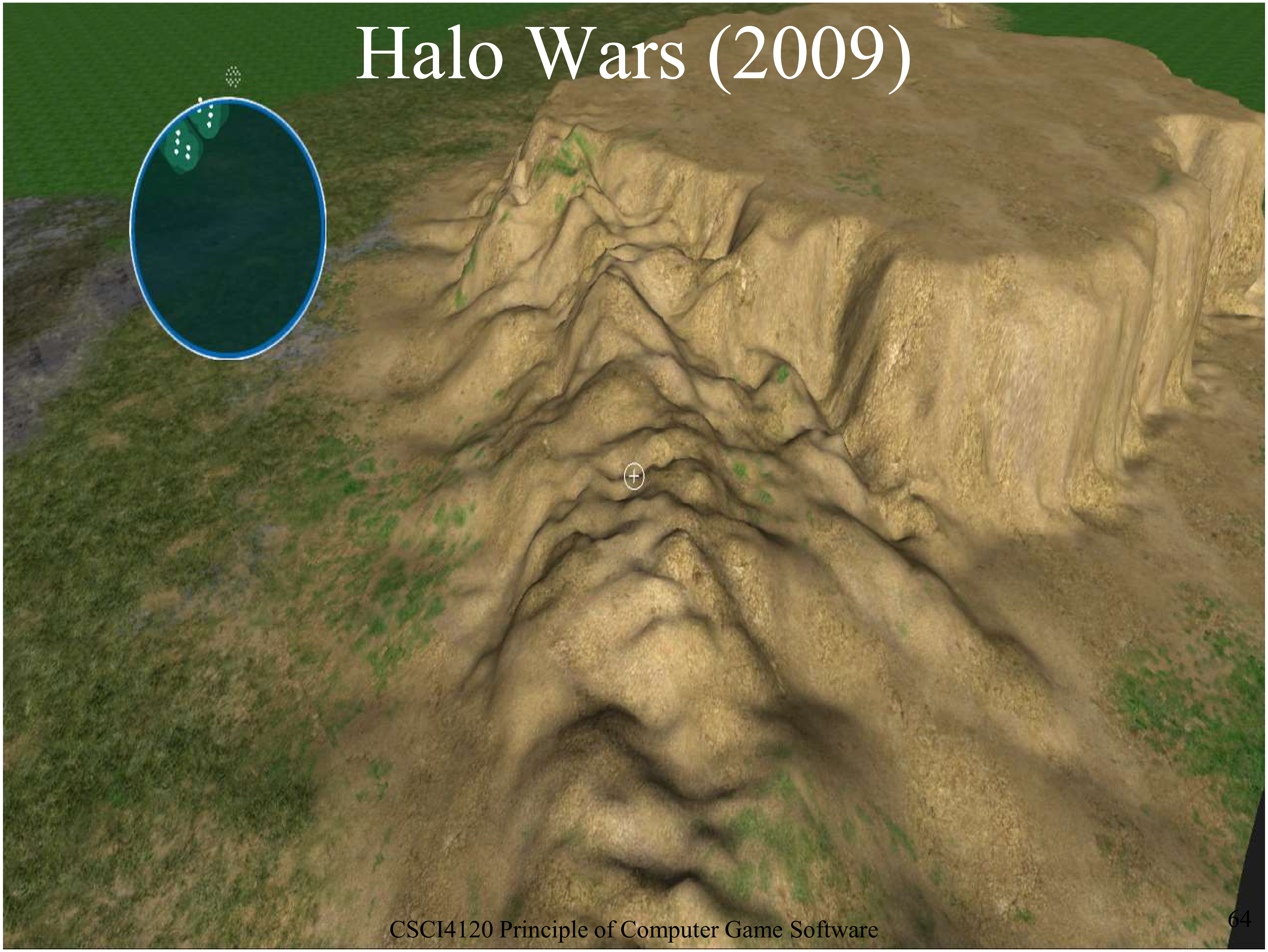
Sudden Pops(change in resolution)

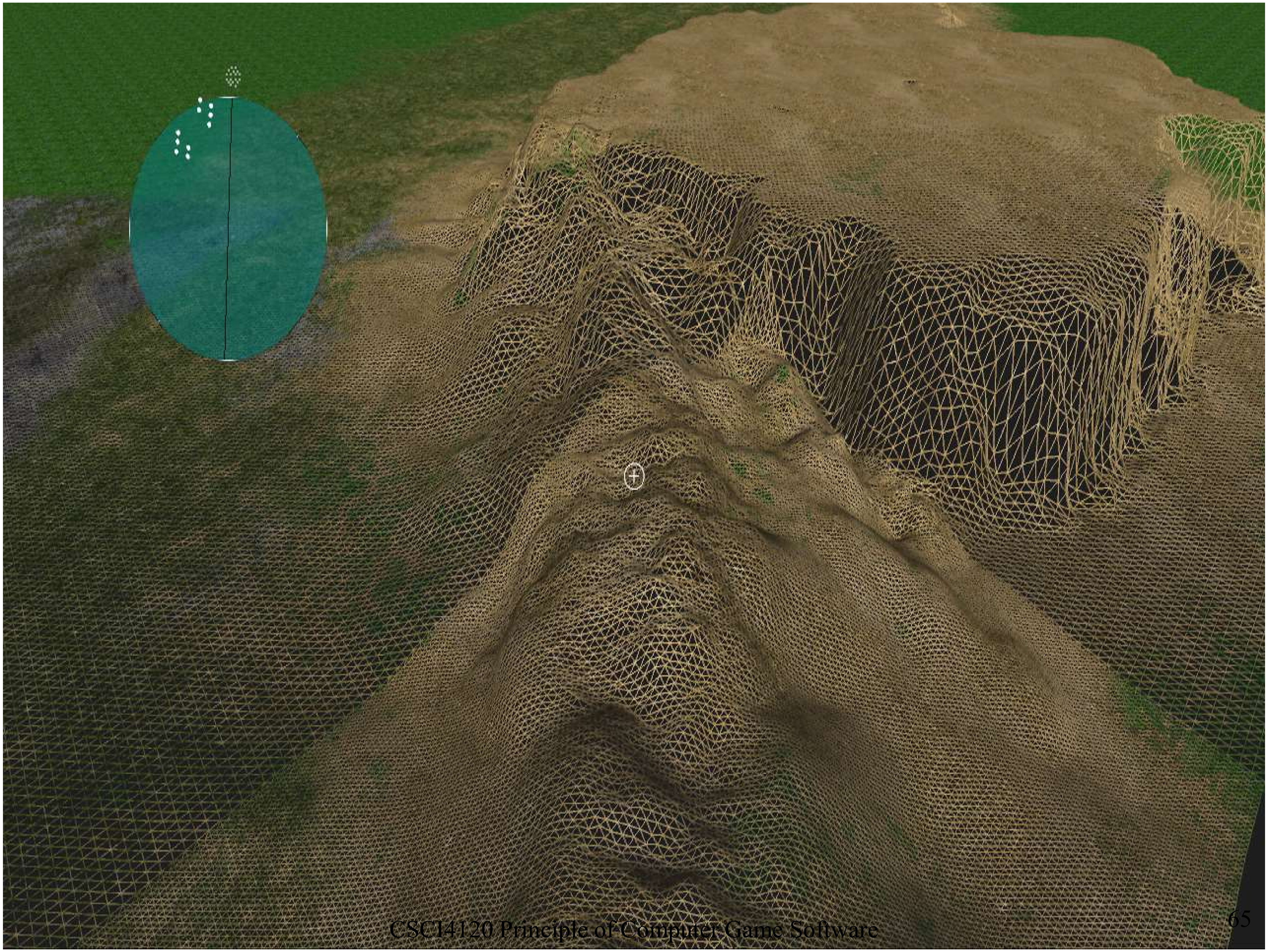
- lesser popping artifacts
- allows for simpler rendering and thus hardware tessellation



LOD quadtree selection, dark region means frustum culled

Halo Wars (2009)





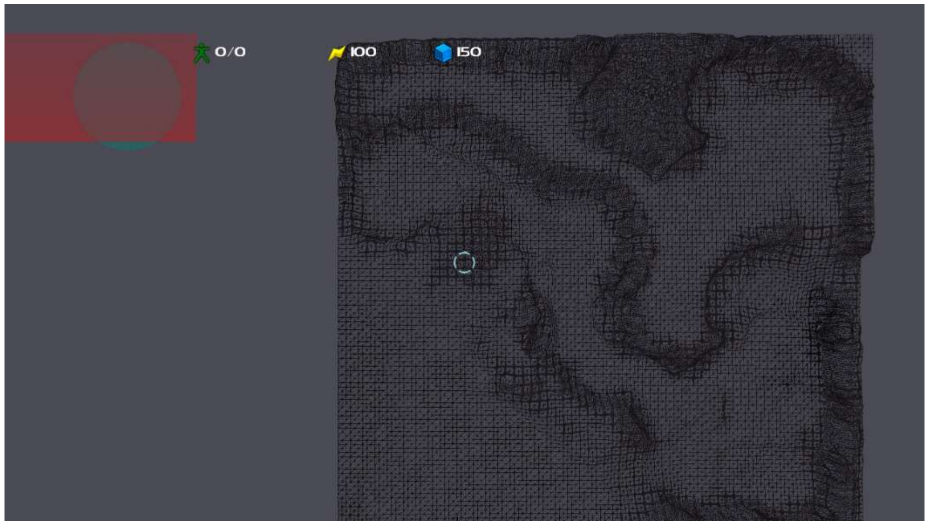
GPU

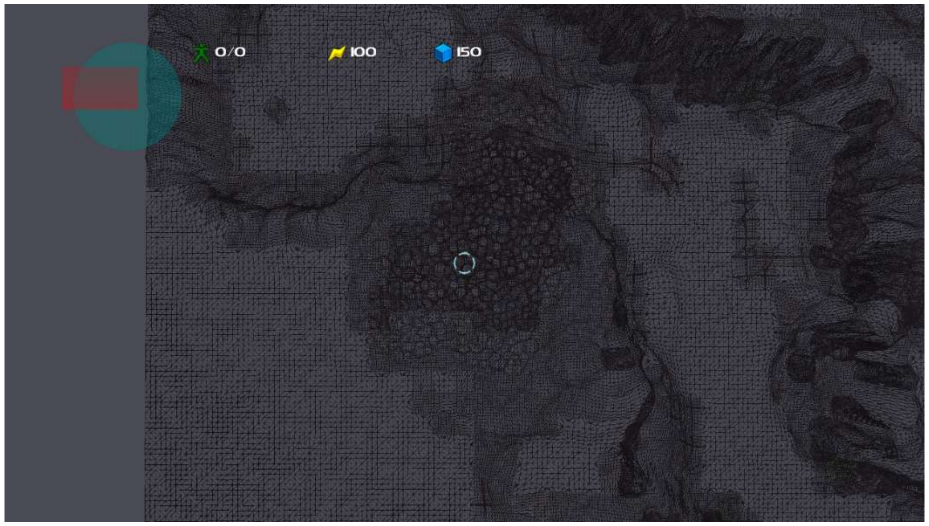
- New generation GPU provides huge processing power
- Favor loading meshes to accelerator memory
- Prevent mesh from changing at the benefit of efficient rendering
- As GPU’s computation power is now exceeding that of CPU, contemporary rendering engine will leverage on GPU-centric approach

GPU-centric approach

- Store terrain geometry directly on GPU
- Quadtree – its leaf nodes are terrain blocks, being stored in main memory

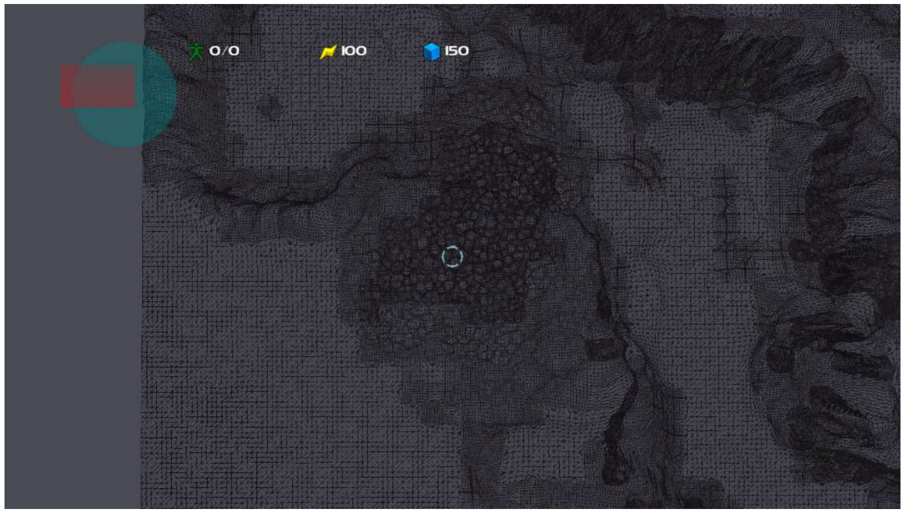
- Use hardware quad patch tessellation
  - Subdivide terrain to 16x16 patches

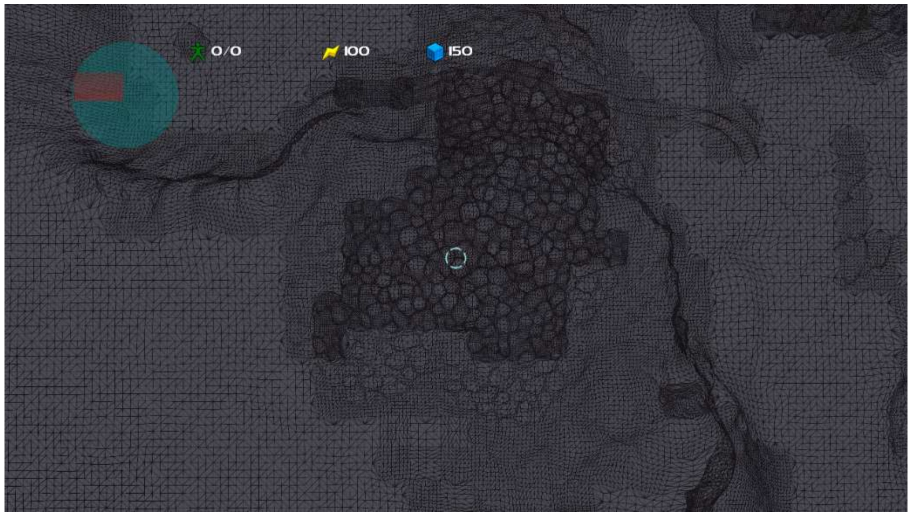




GPU-centric approach

- Project the patch into screen-space at runtime
- Determine visible contribution in screen area by
  - Average normals for each terrain block computed
  - The dot product of this normal with view vector determine whether the block render or not
  - Interpolate between set tessellation steps





GPU centric approach

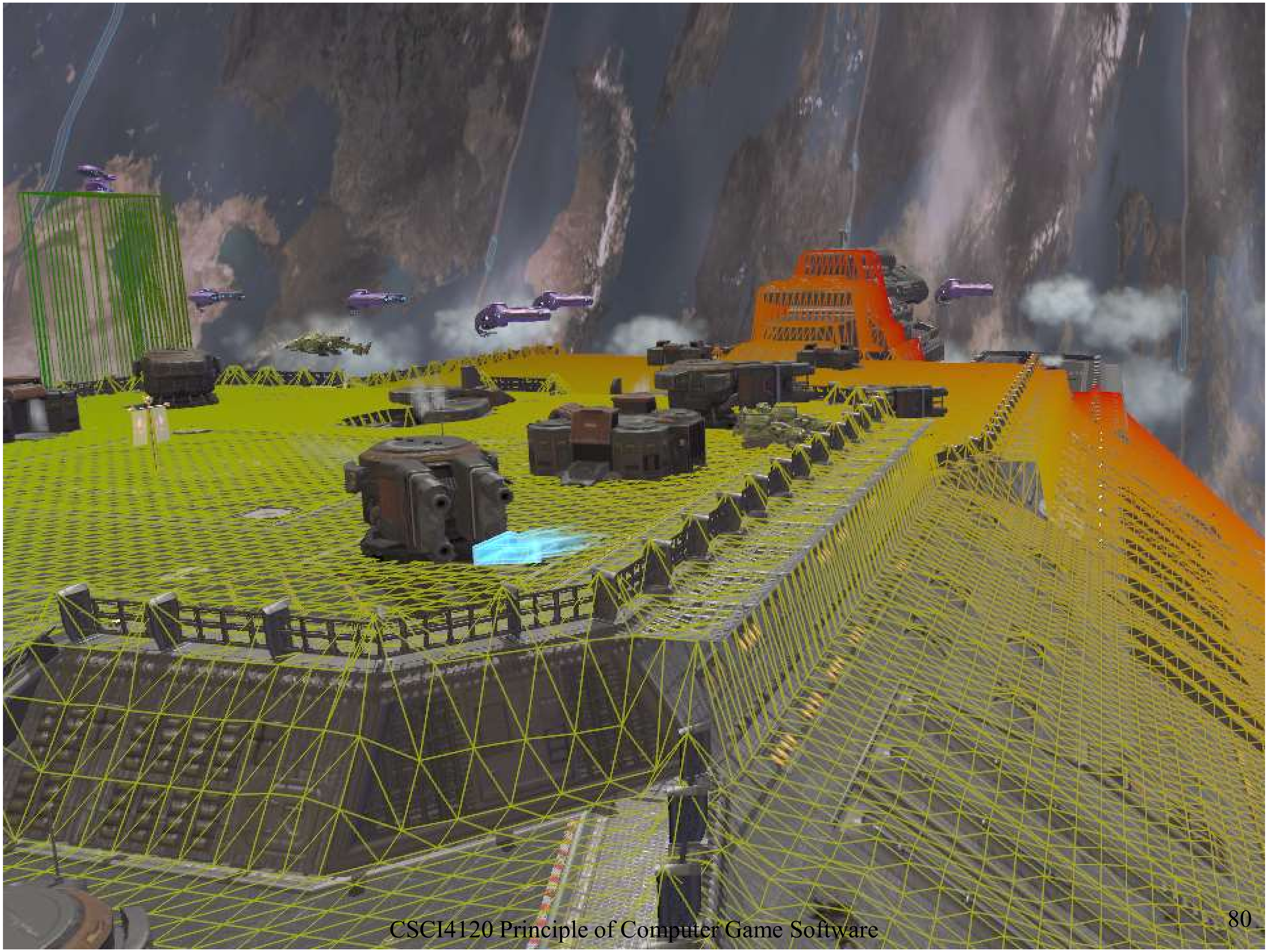
- Problems : New terrain was too difficult for the game play module to use
  - GPU specific & compressed
- Opted to create a lower-resolution version
  - Height field only
  - Acted just like older terrain for legacy systems
- Solution : create a CPU side terrain which is much lower resolution than the GPU side version..
- Data organization optimized for height field ray cast & physics

Outdoor Scene Graphs

- Outdoor scenarios – cities, forests ..
- Huge loading on renderer
- To render these details
  - Use *instance* based engines
  - LOD analysis
  - Fast routine to discard an object

Outdoor Scene Graphs

- Set up primitive list holding fundamental building blocks, similar to sprite list in 2D side scroller
- Each object with LOD support
- Spatial indexing to provide which objects within frustum :
- Regular grid with bucket holding lists of objects <*primitiveid*, *position*>
- Can help collision detection as well



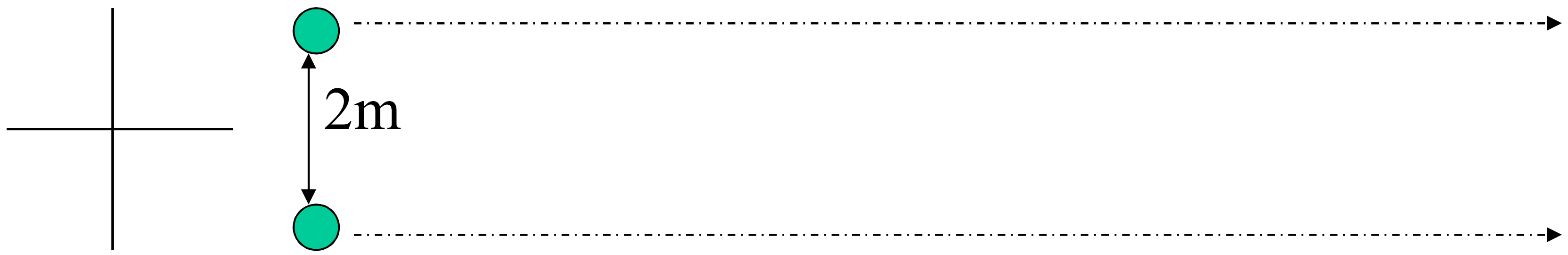
Vast Area

- A number of issues arousing from implementing great regions for player to navigate

  – Precision issue
  – Enormous textures for great details
  – Memory management issues aroused
  – Content streaming to provide a seamless world

Precision Issue

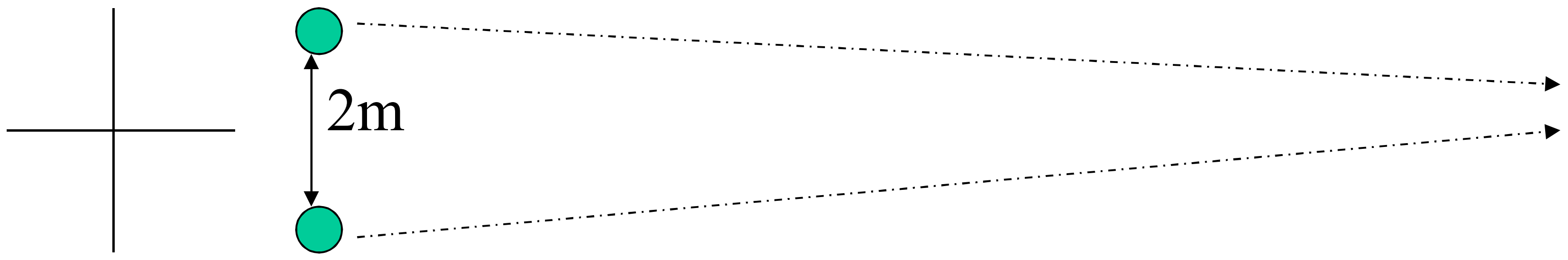
- Assume two characters walking in formation 2 m apart



- separation distance(2m) is overwhelmed by the distance from the origin at certain point “two characters seem at the same location”

Precision Issue

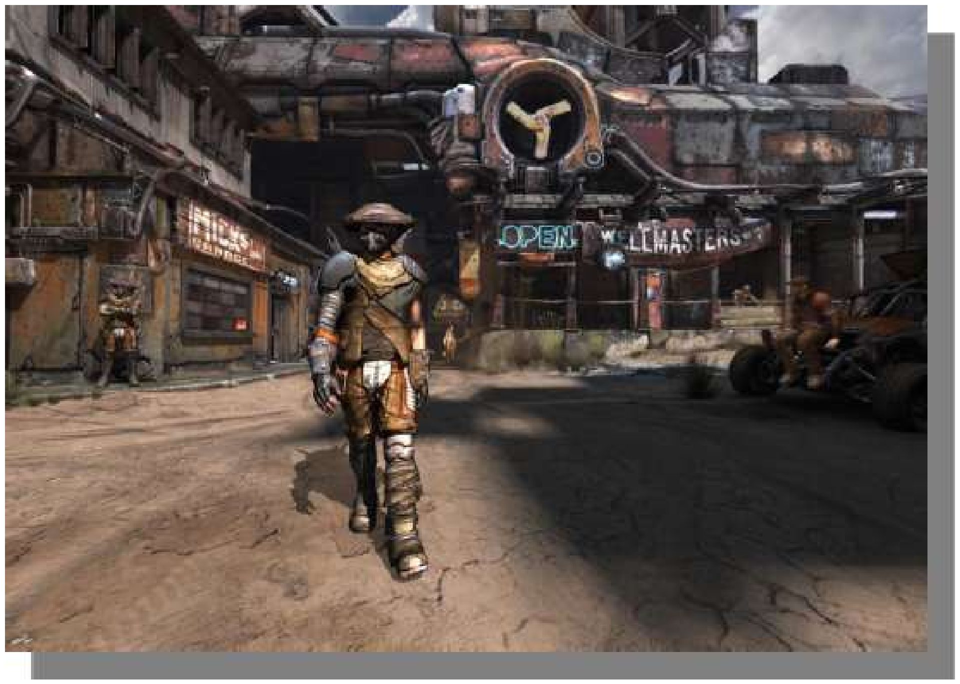
- Reason : With floating point, the further you get from the origin, the more precision you lose

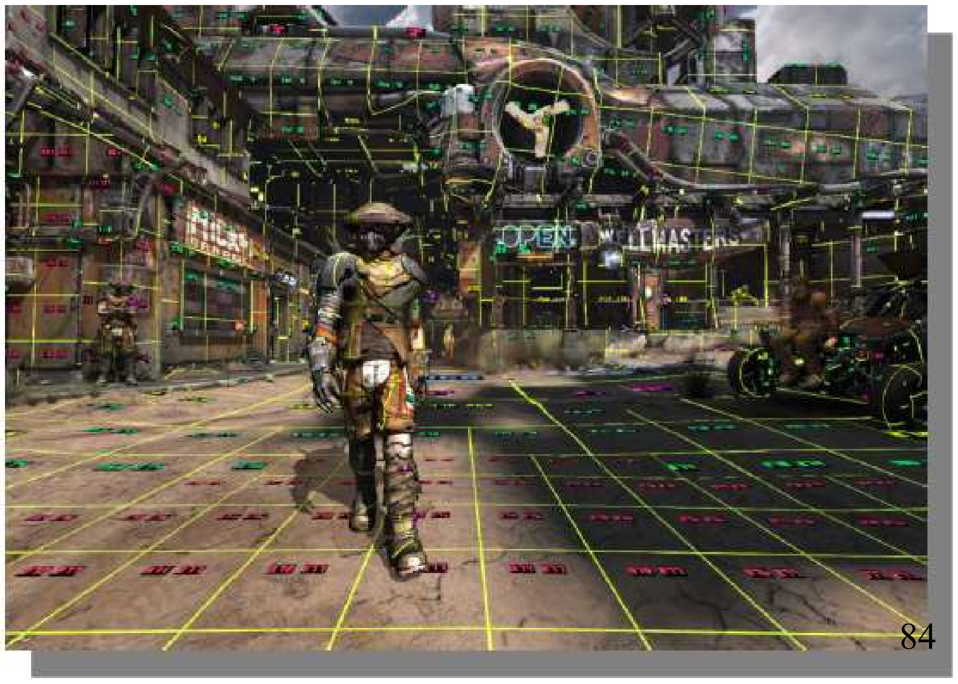


- Solution:
  segment the continuous world into a set of independent coordinate spaces
  switch among them periodically to reset the precision

Virtual Textures

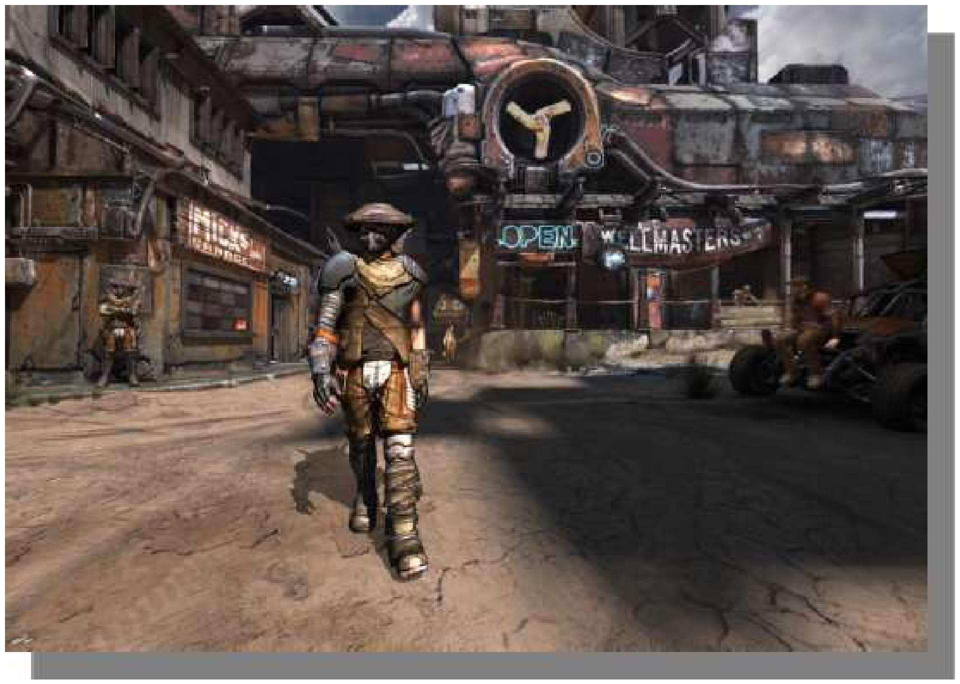
- A single very large virtual textures for static terrain
  - Very Large = 128k x 128k texels (1024 pages on a side)
  - Allowing large amount of details
  - As the player moves around the game, different sections of the texture are loaded into memory, scaled and applied to the 3D models of the terrain.

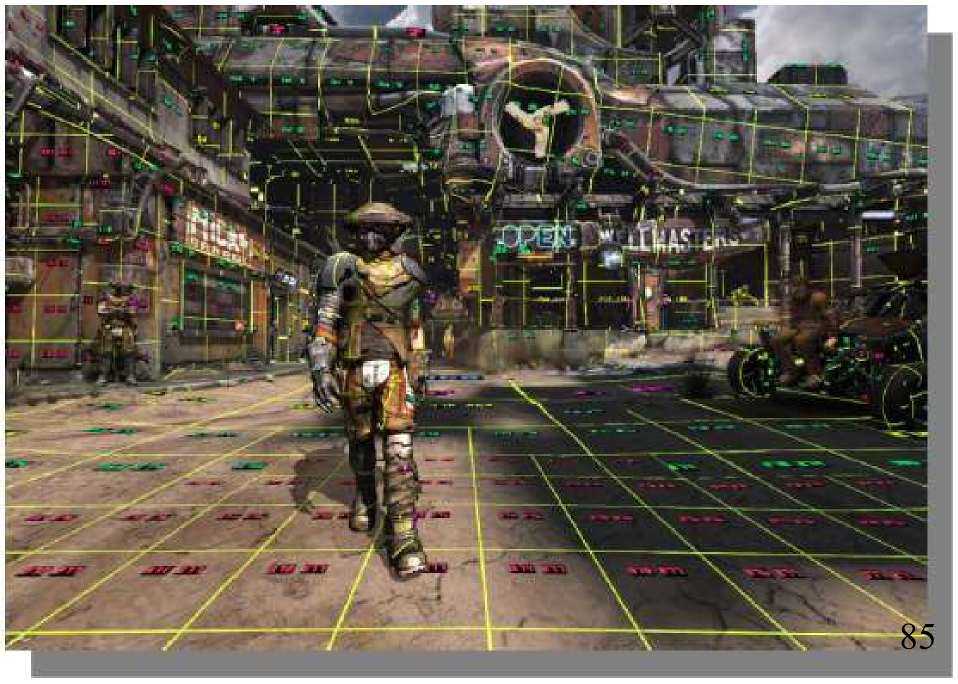




Virtual Textures

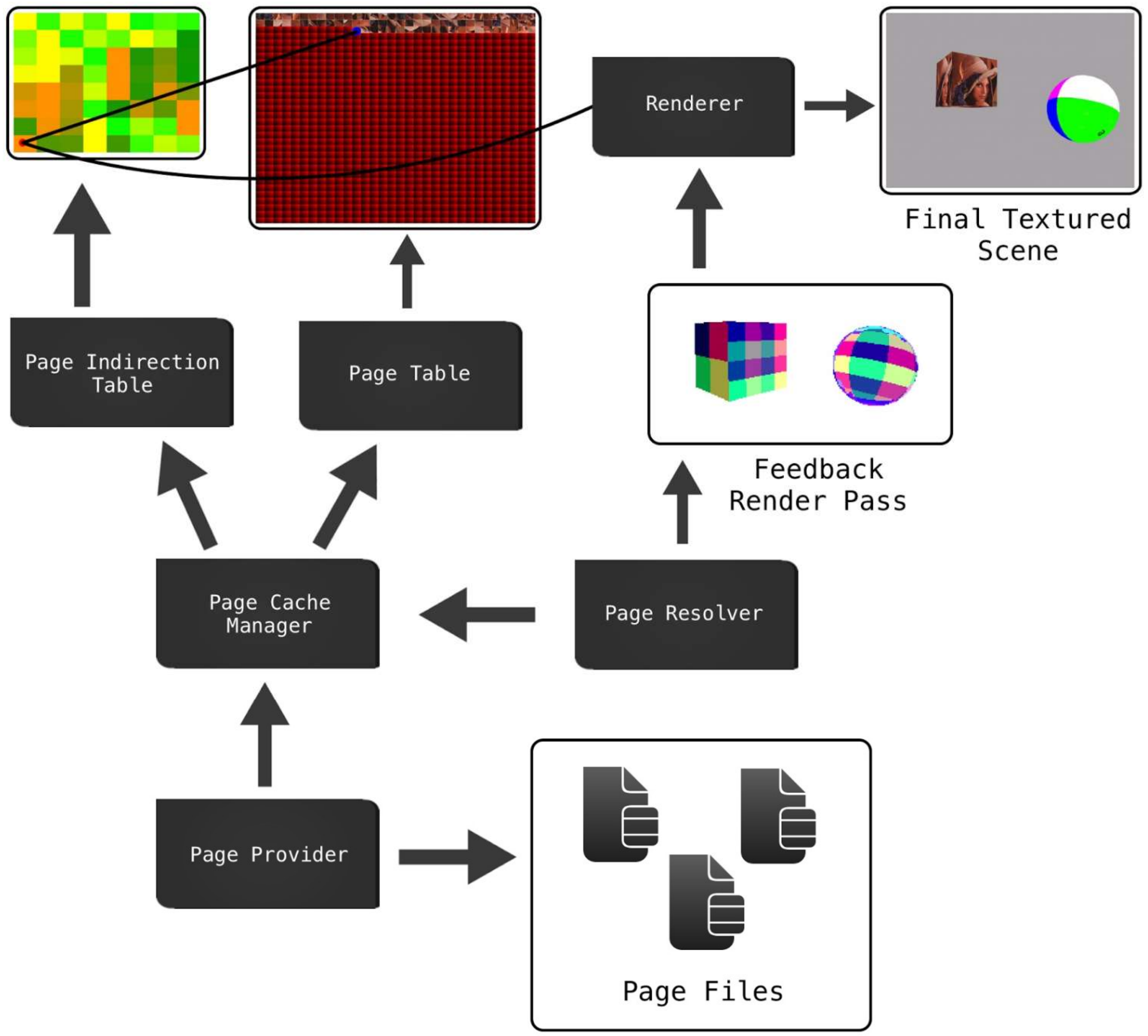
- A unique parameterization of the whole terrain mesh
- And map this to a 2D (great) virtual texture space
- Artists can then coloring everywhere in the world with differently (no repeating texture!)





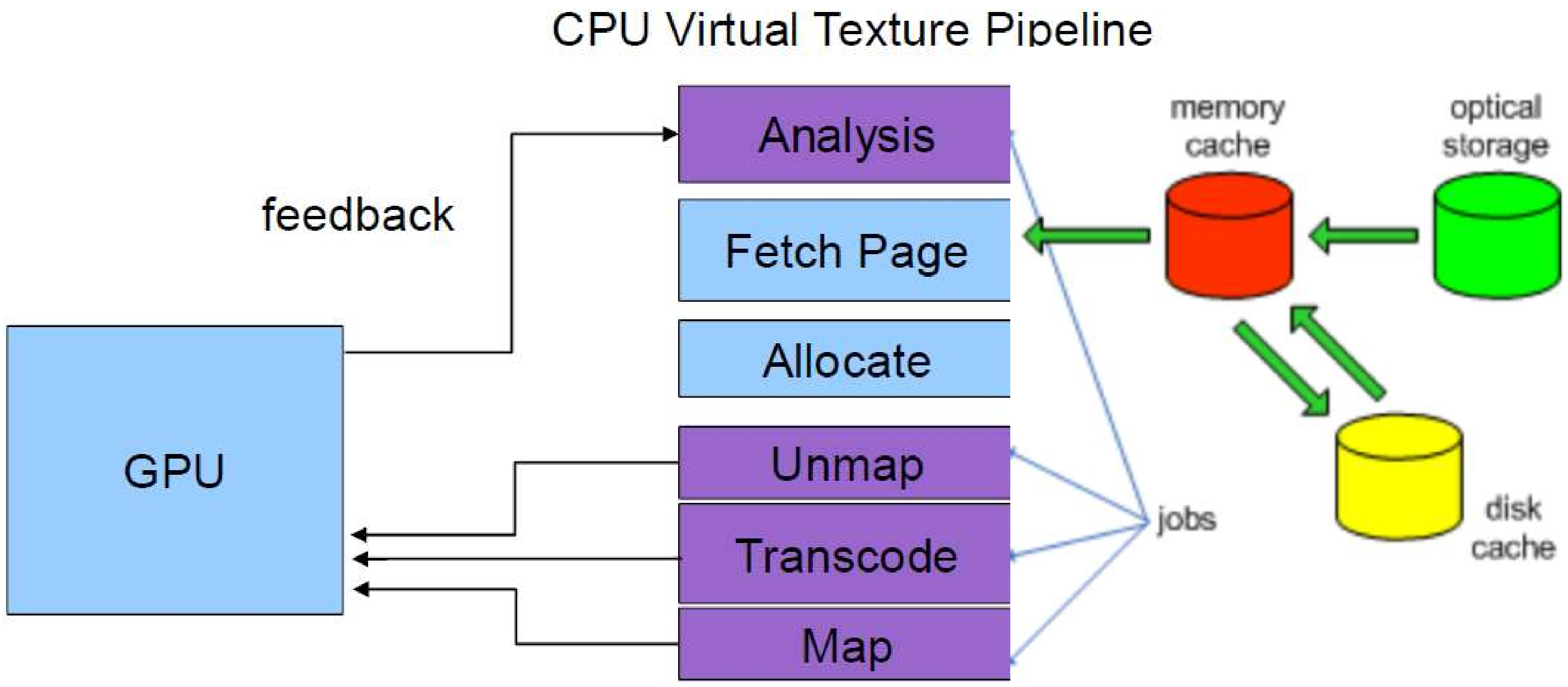
Virtual Textures

- Results in computation intensive complex system with dependencies that need to take advantage of modern parallel (multi-core) systems



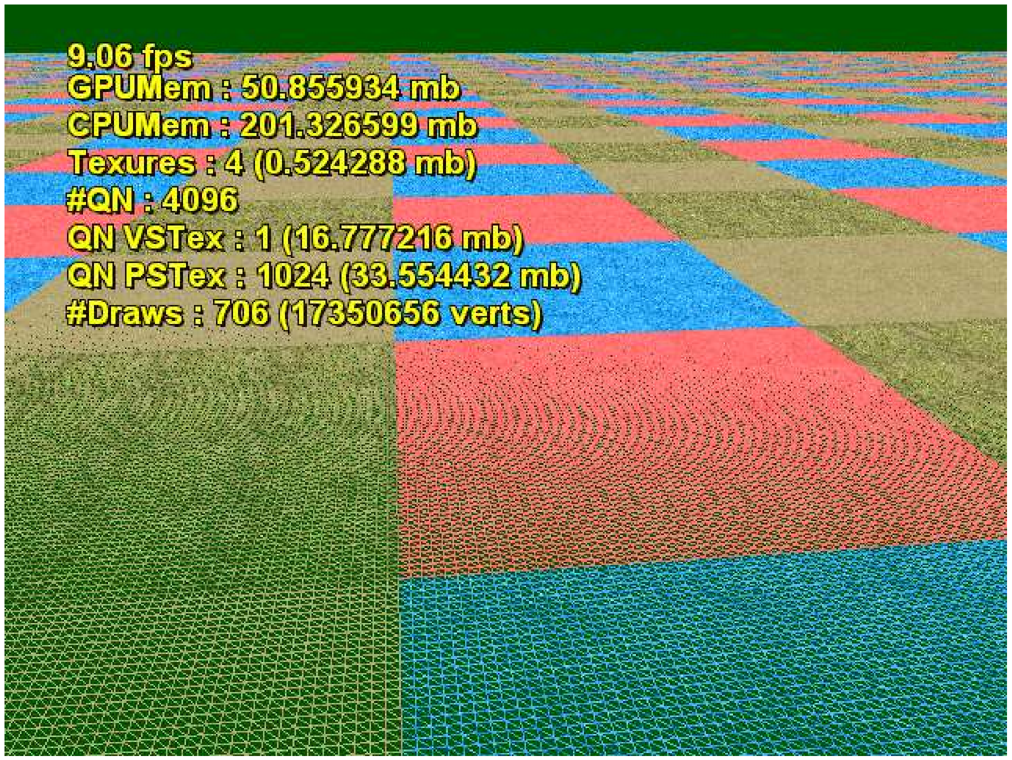
Virtual Textures

- Memory requests cannot always be satisfied due to I/O delay
- Low resolution version kept in cache



Memory Requirement

- For terrain system in Halo Wars
  Terrain was 1280 x 1280 vertices
- 80mb of uncompressed data alone without textures
  - 40mb position, 40mb normals
- Need efficient memory compression
- Using DXT5A textures & DXT3A
- Result : 14.063mb



```
9.06 fps
GPUMem : 50.855934 mb
CPUMem : 201.326599 mb
Texures : 4 (0.524288 mb)
#QN : 4096
QN VSTex : 1 (16.777216 mb)
QN PSTex : 1024 (33.554432 mb)
#Draws : 706 (17350656 verts)
```

Issues in VT

- Zooming in/out is hard to handle, especially when camera is moving
  - More texture pages requested than system can stream in
- Doesn't handle transparency nicely

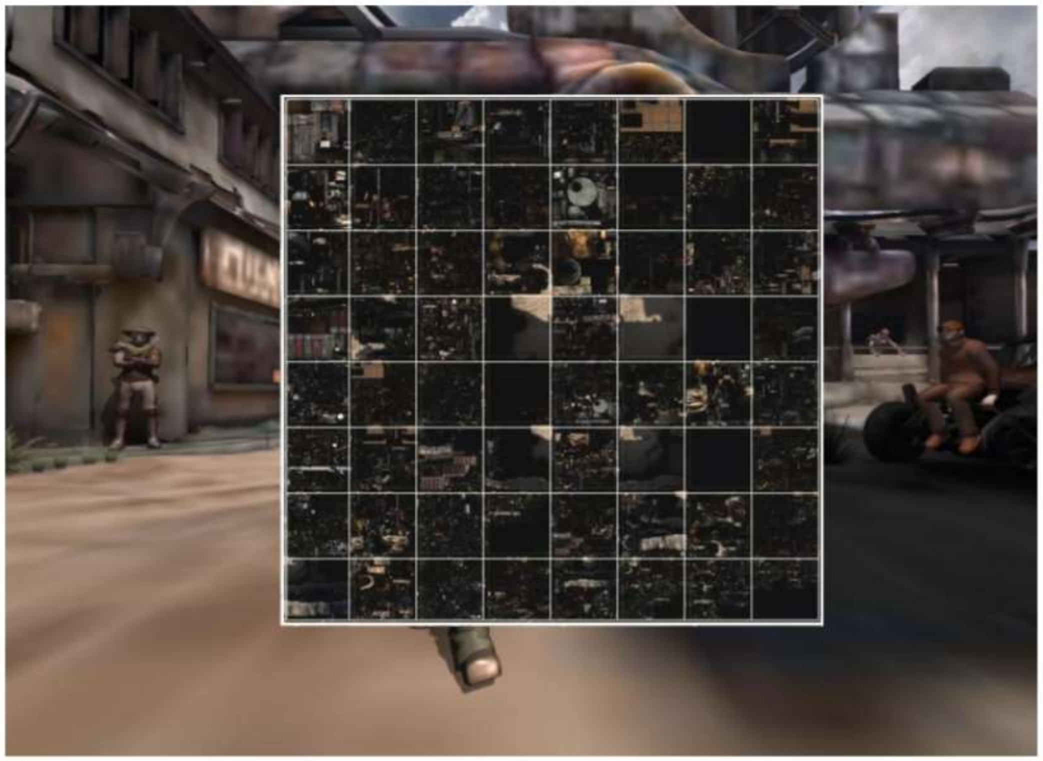


Figure 9: Physical texture with 8 x 8 pages. Oversubscribed system with everything blurry.

http://www.mrelusive.com/publications/papers/Software-Virtual-Textures.pdf

Content Streaming

- optimized loading of large chunks of raw content on-demand according to a visibility prediction scheme

- dynamically background-loading and unloading the complex object-oriented data associated with levels

Red Dead Redemption – with huge Connected world

https://thumbs.gfycat.com/InexperiencedMadKiskadee-mobile.mp4

Content Streaming

- Several key points of the content streaming system
- Asynchronous reading from files
  - A “job manager” to manage the file reading threads
  - Better to read from single resources file for efficiency

- A mechanism to allow game program to know a resource is ready i.e. callback

- An algorithm to determine which resource need to be loaded – streaming content manager
  - Reference counting system to control resource load/unload
  - Volume based region subdivision to decide whether a region/sector need be loaded into memory

Summary

- Clipping, culling and occlusion handling is vital in indoor scene rendering
- Hardware assisted culling will also affect the techniques chosen
- Increasing the details in scene viewing is still the goal of real time rendering engine

Summary

- Outdoor – focus on details handling
- Front-to-back terrain render follow by other objects render is quick and easy solution
- But more elegant solution needed

Reference

- Virtual texture
  http://silverspaceship.com/src/svt/
- Halo terrain presentation Youtube video
  https://www.youtube.com/watch?v=In1wzUDopLM
- Planetary Annihilation tech talk on planet terrain
  http://mavorsrants.blogspot.com/2013/02/planetary-annihilation-engine.html
- Seamless world
  https://www.gamedevs.org/uploads/the-continuous-world-of-dungeon-siege.pdf