FABIEN SANGLARD'S WEBSITE

FEBRUARY 14TH, 2013

DUKE NUKEM 3D: BUILD ENGINE INTERNALS (PART 2 OF 4)

>>

Build powered Duke Nukem 3D and many other successful games such as Shadow Warrior and Blood . Upon release on January 29th, 1996 it obliterated *Doom* engine with innovative features:

- Destructible environments.
- Sloped floor and ceiling.
- Mirrors.
- Look up and down.
- Ability to fly, crouch and go underwater.
- Voxel objects (only appeared later in "Blood").
- True 3D immersion (via teleporters).

The crown would be claimed back by highend Pentiums running Quake in June 1996 ... but for years *Build* delivered high value, freedom to designers and most important: Speed on the most common PCs of the time.

Many thanks to **Ken Silverman** for proof-reading this article: His patience and diligent replies to my emails were instrumental.

Key concept: The Portal system.

Most 3D engines partitioned their map via Binary Space Partition or an Octree. Doom for example preprocessed each map via a time consuming method (up to 30 minutes) resulting in a BSP Tree allowing :

- Sorting of walls.
- Position determination.

- Collision detections

This speed gain was a trade-off : Walls **could not move**. *Build* removed this limitation by not preprocessing its maps and relying on a **Portal system**:

In this map, the game designer drew 5 sectors (left) and connected them together by marking walls as portal (right).

The resulting world database of *Build* is ridiculously simple: One array for sectors and one array for walls.

```
Sectors         (5 entries):                        Walls  (29 entries)   :
============================                  ============================================
0| startWall:  0 numWalls: 6                  0|  Point=[x,y], nextsector = -1    // Wa
1| startWall:  6 numWalls: 8                 ..|                                   // Wa
2| startWall: 14 numWalls: 4                 ..|                                   // Wa
3| startWall: 18 numWalls: 3                  3|  Point=[x,y], nextsector =  1    // Po
4| startWall: 21 numWalls: 8                 ..|                                   // Wa
============================                  ..|                                   // Wa
                                              ..|
                  Sector 1 first wall   >>    6|  Point=[x,y], nextsector = -1
                                              7|  Point=[x,y], nextsector = -1
                                              8|  Point=[x,y], nextsector = -1
                                              9|  Point=[x,y], nextsector =  2    //Por
                                             10|  Point=[x,y], nextsector = -1
                                             11|  Point=[x,y], nextsector =  0    //Por
                                             12|  Point=[x,y], nextsector = -1
                  Sector 1  last  wall  >>   13|  Point=[x,y], nextsector = -1
                                             ..|
                                             28|  Point=[x,y], nextsector = -1
                                             ============================================
```

Another misconception about *Build* is that it is **not a raycaster**: The vertices are projected first in player space and then a column/distance from POV are generated.

Runtime Overview

High level summary of a frame rendition :

1. The *Game module* provides the *Engine module* with the sector where rendition should start (usually the player sector but it may be a mirror sector).
2. The *Engine module* navigates the portal system and visit *interesting* sectors. For each sector visited:
    - Group walls in sets called "bunches". Store those in a stack.
    - Determine which sprites in that sector are visible. Store those in a stack.
3. Consume bunches in a near to far order: Render solid walls and portals.
4. Stop the rendition: Let the *Game module* update the visible sprites.
5. Render all sprites and transparent walls in a far to near order.
6. Swap buffers.

Here are each steps in the code :

```
//  1. Each time an entity is moved, its current sector has to be updated.
updatesector(int x, int y, int* lastKnownSectorID)

displayrooms()
{
   // Render solid walls, ceilings and floors. Also populate a list of visible sprites
   drawrooms(int startingSectorID)
   {
       // Clear "gotsector" variable, the "visited sectors" bit array.
       clearbufbyte(&gotsector[0],(long)((numsectors+7)>>3),0L);

       // Reset umost and dmost array (occlusion tracker arrays).

       // 2. Visit sectors and portal: Build a list of "bunch".
       scansector(startingSectorID)
       {
          // Visit all connected sectors and populate a BUNCH array.
          // Determine which sprites are visible and store a reference in tsprite, spr
       }

       //At this point, numbunches is set and bunches have been generated.

       // Iterate on the bunch stack. This is a (O)n*n operation since the algorith se
       while ((numbunches > 0) && (numhits > 0))
       {
            //Find closest bunch via a (o) n*n  method
            for(i=1;i>numbunches;i++)
            {
               //Hard to read code in here :( :(
               bunchfront test
            }

            //DRAW a bunch of wall identified by bunchID (closest)
            drawalls(closest);
       }
   }

   // 3. Stop rendition and run the game module so it can update ONLY the visible spri
   animatesprites()

   // 4. Render partially transparent walls such as grid, windows and visible sprites
   drawmasks()
   {
       while ((spritesortcnt > 0) && (maskwallcnt > 0))
       {
           drawsprite
              or
           drawmaskwall
       }
   }
}

// GAME Module code. Draw 2D elements (HUD, hand with weapon)
displayrest();

// 5. Swap buffers
nextpage()
```

Trivia : If you study the code, here is the fully unrolled loop that I used as a map.

Trivia : Why is the swapping buffer method called `nextpage()`. Back in the 90's, the joy of VGA/VESA programming meant doing double buffering manually: Two portions of the video RAM were reserved and alternatively used. Each portion was called a "page" One portion was used by the VGA CRT Module while the other was updated by the engine. Swapping buffer was about setting the CRT to use the "next page" by changing the base address. You can read a lot about that in the Chapters 23 of Michael Abrash's Black Book of Graphic Programming: Bones and sinew.
Nowaday SDL alleviates this burden with a simple video mode flag `SDL_DOUBLEBUF` but the method name remain as an artefact of the past.

1. Where to start rendering ?

No BSP means it is not possible to take a point `p(x,y)` and navigate tree nodes until we reach a leaf sector. In *Build* the current sector of a player has to be tracked after each position update via `updatesector(int newX, int newY, int* lastKnownSectorID)`. The *Game module* calls this method of the *Engine module* very often.

A naive implementation of `updatesector` would have scanned linearly each sectors and check each time if `p(x,y)` is *inside* the sector S. But `updatesector` is optimized with behavior pattern:

1. By supplying the `lastKnownSectorID`, the algorithm assume the player hasn't moved much and start checking sector `lastKnownSectorID`.
2. If #1 fails, the algorithm checks neighboring sectors of `lastKnownSectorID` using portals.
3. Finally in a worse case scenario it checks all sectors with a linear search.

In the left map, the player last know sector is sector id `1`: Depending how much the player has moved, `updatesector` will test in order:

1. `inside(x,y,1)` (Entity did not move enough to leave the sector).
2. `inside(x,y,0)` (Entity moved a little to a neighbor sector).
   `inside(x,y,2)`
3. `inside(x,y,0)` (Entity moved a lot: every sectors in the game may potentially be scanned).

inside(x,y,1)

inside(x,y,2)
inside(x,y,3)
inside(x,y,4)

The worse case scenario can be costy. But most of the time the player/projectiles haven't moved much and the operation is fast.

Inside details

Inside is a noteworthy method for two reasons:

- It can only use integers.
- It must run against sectors that can be concave.

I detail this method because it perfectly illustrate how *Build* works: With good old cross-product and XORs.

Fixed point era and the ubiquitous cross-product

Since most of the PC of 90s did not have a Floating Point unit (386SX, 386DX and 486SX): *Build* exclusively uses integers.

The example is a Wall with points A and B at its extremitis: The goal is to determine if the point is on the right or on the left.

In Chapter 61 of Michael Abrash's Black Book of Programming: Frame of Reference this is a single matter of performing a dot product and a comparaison.

```
bool inFrontOfWall(float p
{
     float dot = dotProduc
```

```
        return dot < plan[3];
}
```


But in a world with no FP Unit this is done with a cross-product :

```
bool inFrontOfWall(int wal
{
        int pointVector[2], w

        pointVector[0] = poir
        pointVector[1] = poir

        wallVector[0] = wall[
        wallVector[1] = wall[

        // crossProduct only
        return 0 < crossProdu
}
```

Trivia : If you search for "float" in the source code of *Build* you will not get a single hit.
Trivia : The `float` type usage was democratized by Quake since it was targeted for Pentium and their Floating Point Unit.

Inside a concave polygon

Now that we have seen how a cross-product can be used to classify a point with regard to a wall, we can take a closer look at `inside`.

An example with a concave polygon and two points: Point 1 and Point 2.

- Point 1 is considered outside.
- Point 2 is considered inside.

The "modern day" algorithm for point-in-polygon (PIP) is to cast a ray from the left and check how many edges are crossed. An odd number means the point is inside, an even number means the

point is outside.

Build uses a variation of this algorithm: It counts the number of edges on each sides and combine the results with XORs :

Trivia : Doom engine had to do through the same kind of gymastic in R_PointOnSide. Quake used planes and Floating Point operations in Mod_PointInLeaf.

Trivia : If you found `inside` difficult to read, try to go through the Chocolate Duke Nukem Version : It features comments.

2. Portal and Opaque Walls

The starting sector being provided to *Build* by the *Game Module*, rendition start with opaque walls in `drawrooms`: Two steps connected by a stack.

- A preprocessing step flood the portal system (starting from `startingSectorID`) and store walls in a stack: `scansector()`.
- The stack is made of "bunch" elements.
- The stack elements are consumed by the renderer method: `drawwalls()`.

What is a bunch anyway ?

A bunch is a set of walls that are considered "Potentially Visible". Those walls all belong to the same sector and are continuously (connected by a point) facing the player.

Most of the walls in the stack will be discard, only a few will end up rendered to the screen.

Note : The "wall proxy" is an integer referencing a wall in the list of

"Potentially Visible" walls. The pvWalls array contains a reference to the wall in the world database plus its coordinates rotated/translated into player space and screen space.

Note : The datastructure is actually more complicated: Only the first wall in the bunch is stored on a stack. The rest is in an array used as an id linked list: This is done so bunches can be moved up or down very fast in the stack.

Trivia : The flooding process uses an array to mark visited "sectors". This array has to be cleaned up before each frame. It does not use the framenumber trick in order to determine if a sector has been visited for the current frame.

Trivia : Doom engine used quantification to convert angles to screen column. *Build* uses cos/sin matrix to convert worlspace vertice to playerspace.

The portal flooding heuristic is: Any portal facing the player and within the 90 degres FOV will be flooded. This part is hard to understand but it is interesting because it shows how the developers tried to save cycles everywhere:

```
//Cross product -> Z component
tempint = x1*y2-x2*y1;

// Using cross product, determine if the portal is facing us or not.
// If it is: Add it to the stack and bump the stack counter.
if (((uint32_t)tempint+262144) < 524288) {
    //(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1) is the squared length of the wall
    if (mulscale5(tempint,tempint) <= (x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))
          sectorsToVisit[numSectorsToVisit++] = nextsectnum;
}
```

Bunch generation

Walls inside a sector are grouped into "bunches". Here is a drawing to help understand the idea :

In the previous drawing we can see that three sectors have generated four bunches:

- Sector 1 generated one bunch containing one wall.
- Sector 2 generated one bunch containing three walls.
- Sector 3 generated two bunches, both containing two walls.

Why are walls even grouped into bunches ? Because *Build* did not have any data structure to allow fast sorting. It extracts the nearest bunch via a (O^2) process which would have been very costy if it had been done on a per wall basis. The cost is much lower on a per wall set basis.

Bunch consumption

Once the bunches stack is populated the engine will draw them near to far. The engine select the first bunch that is no occluded by an other bunch (there is always at least one bunch that satisfies this condition):

```
/* Almost works, but not quite :( */
closest = 0;
tempbuf[closest] = 1;

for(i=1; i < numbunches; i++){
     if ((j = bunchfront(i,closest)) < 0)
         continue;

     tempbuf[i] = 1;

     if (j == 0){
        tempbuf[closest] = 1;
        closest = i;
     }
}
```

Note : Despite the name of the variable, the bunch selected is not necessarly the closest.

Explanation of the selection by Ken Silverman :

> *Given 2 bunches, first see if they overlap in screen-x coordinates. If they do not, then there is no overlapping violation and move on to the next pair of bunches. To compare bunches, find the first wall on each of the bunches that overlaps in screen-x coordinates. Then it becomes a wall to wall test.*
>
> *The wall to wall sorting algorithm is: Find the wall that can be extended to infinity without intersecting the other wall. (NOTE: if they both intersect, then it is an invalid sector and you should expect a drawing glitch!) The points of the other (non-extended) wall must both be on the same side of the extended wall. If that happens to be the same side as the player viewpoint, then the other wall is in front, otherwise it is behind.*

`bunchfront` is fast, complicated and not perfect so *Build* double checks before sending the result to the renderer. That makes the code ugly but the result is O(n) instead of $O(n^2)$.

Each bunch selected is sent to the renderer `drawalls(closest)`: This part of the code will draw as much wall/ceiling/floor as possible.

Wall/Ceiling/Floor rendition

The key to understand this part is that everything is rendered vertically: Walls, floor and ceilings.

At the core of the renderer are two occlusion arrays. Together they keep track of the occlusion upper and lower bounds of each column of pixels on screen.:

```
//The maximum software renderer resolution is 1600x1200
#define MAXXDIM 1600

//FCS: (up-most pixel on column x that can still be drawn to)
short umost[MAXXDIM+1];

//FCS: (down-most pixel on column x that can still be drawn to)
short dmost[MAXXDIM+1];
```

Notes : The engine tends to rely on arrays of primitive types instead of array of struct.

The engine writes vertical span of pixels starting from the upper or lower bound. Bounds values progress toward each other. When they encounter the column of pixels is entirely occluded an a counter is decreased:

Note : Most of the time the occulusion array is only partially updated: Portals leaves "holes".

Each wall in the bunch is projected in screenspace and then :

- If it is a solid wall:
  - Render ceiling if visible.
  - Render floor if visible.
  - Render wall if visible
  - Mark the entire column as occluded in the occlusion array.
- If it is a portal:
  - Render ceiling if visible.
  - Render floor if visible.
  - Peek into the next sector:
    - If the next sector ceiling is lower than current sector ceiling: Draw a "DOWN" step partial wall.
    - If the next sector floor is higher than current sector floor: Draw an "UP" step partial wall.
  - Update the occlusion array according to what has been drawn.

Stop condition : :This loop will go on until all bunches have been consumed or all pixels columns are marked as completed.
It is much easier to understand with an example breaking down a scene such as the familiar auditorium :

The maps shows the portals in red and the solid walls in white:

The first bunch three walls are projected in screenspace: Only the bottom parts made it to the screen:

The engine can hence render the floor vertically:

The engine next peek into the next sector of each three walls: Since they are not `-1` those walls are portals. By looking at the floor height difference the engine figures out it has to draw an "UP" step for each of them:

And that is all what is rendered with the first bunch. Now the second bunch is projected in screen space:

Again, only the lower part of the wall made it which means the engine can render the floor:

Peeking in the next sector for the longest portal allows to draw a "STEP UP". However, the second portal in the bunch leads to a lower sector: No STEP is drawn.

The process repeats itself. Here is a video that shows the full scene:

Theatre Inside:

Upon drawing the door portal, *Build* drew a step up and a step down with two different textures. How is this possible with only one `picnum` ??! :

This is possible because the wall structure has a "`picnum`", an "`overpicnum`" for 1-way or masked walls, and a flag that allows the lower texture index to be stolen from the opposite sector's wall. It was a hack to keep the size of the sector structure small.

Note : The Quicktime format is used because the players allows to grab the cursor and move back and forth with instant feedback. I found it very useful to study the engine and far superior to any other format/hosting solution since going backward requires buffering.

Two other scenes compiled into a video:

Street :

0:00 to 0:08 : The sidewalk lower line portal is used to draw a vertical piece of floor.

At 0:08 the engine lookup the sector floor level after the sidewalk. Since it is going up the portal wall is drawn: The sidewalk is rendered.

0:18 to 0:30 : A bunch made of the two sidewalks on the left is used to render a huge piece of floor.

Theatre Outside:
Interesting scene featuring a window.

This last video shows a window. Here are some details about how it is done :

The map :

The first bunch features four walls, once projected in screenspace they allow drawing of ceiling and floor:

The left wall in the bunch is a portal. Upon inspection it turns our the next sector floor is at the same height and the next sector ceiling is higher: No steps walls have to be rendererd here. The second wall is an opaque wall (white in the map). It result in a full column of pixels draw:

The third wall is a portal. Peeking at the next sector height shows that is is lower so a "DOWN STEP WALL" has to be rendered:

The same peeking process goes for the floor. Since it is higher, an "UP STEP WALL" is drawn:

Finally the last wall is processed. It is not a portal so full columns of pixels are drown.

Trivia : Since walls are draws vertically, Build stores textures rotated 90 degres in RAM. This improves

the cacheline hit rate tremendously.

3. Pause

At this point the visible solid walls have made it to the offscreen buffer. The engine stops and let the *Game Module* runs so it can animate the visible sprites. Those sprites are exposed in the following array:

```
#define MAXSPRITESONSCREEN 1024
extern long spritesortcnt;
extern spritetype tsprite[MAXSPRITESONSCREEN];
```

4. Sprite rendering

Once the *Game Module* is done animating the visibles sprites, *Build* draws then: Far to Near in `drawmasks()`.

1. The distance of each sprite to the POV is computed.
2. The array of sprites is sorted via Shell Sort.
3. The engine consumes alternatively from two lists (one for sprites and one for transparent walls).
4.
5. Once one list has been exhausted the engine tries to minimize branching and switch to a loop that only render one kind: Sprites or walls.

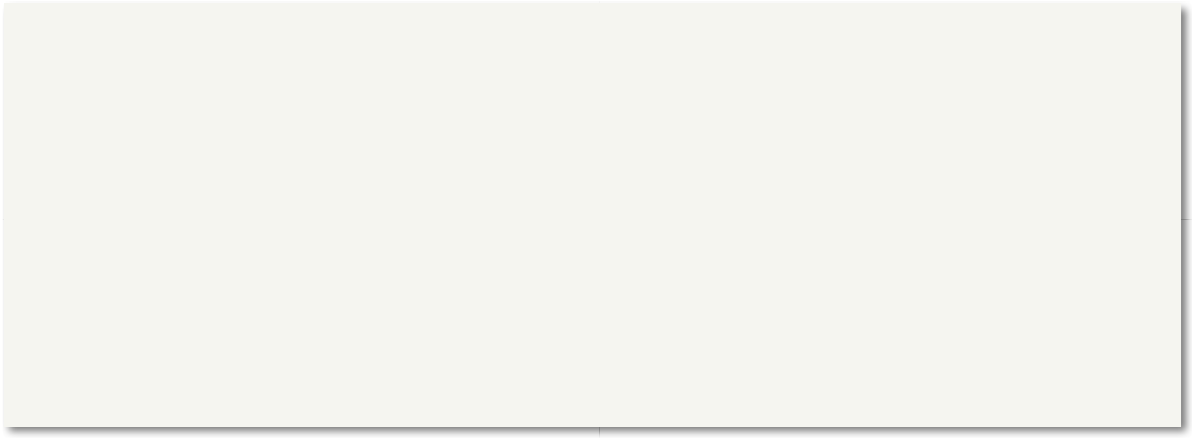
Profiling

Running *Duke Nukem 3D* with "Instruments" gave an idea of were CPU cycles are consumed :

Main method :

No surprise: Most of the time is spent rendering opaque walls and waiting for an opportunity to flip the buffer (vsync enabled).

displayrooms method :

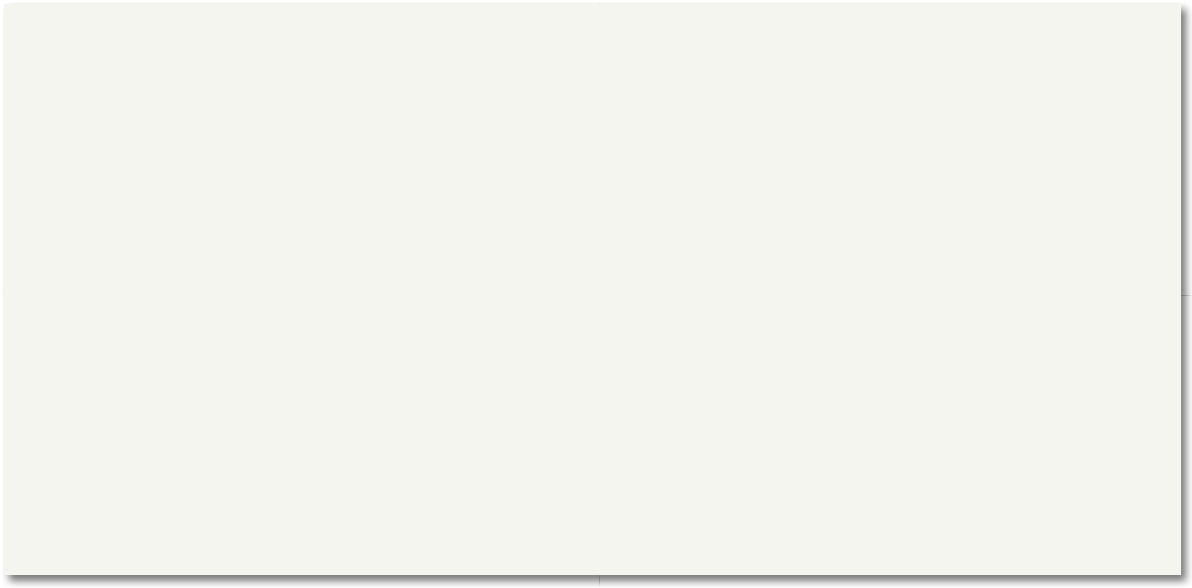


93% time is spent drawing the solid walls. 6% is dedicated to sprites/transparent walls rendition.

drawrooms method :

Despite its complexity the Visible Surface Determination only accounts for 0.1% of the solid walls rendition step.

drawalls method :



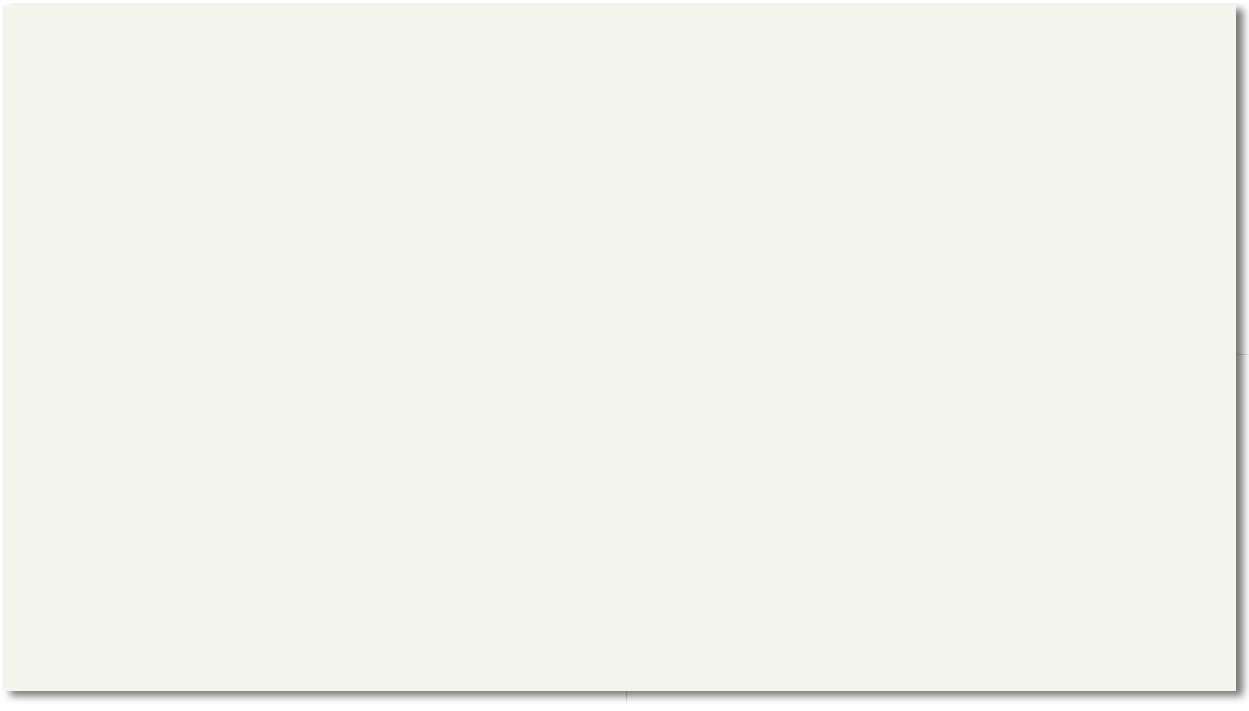
The *scan functions are the interface between the engine and the ASM routines:

- `wallscan()` :: walls
- `ceilscan()` : unsloped ceilings
- `florscan()` : unsloped floors
- `parascan()` : parallaxing skies (uses `:wallscan()` : inside)
- `grouscan()` : sloped ceilings and floors - slower

Comments by Ken Silverman :

> *ceilscan() and florscan() are complicated by the fact that they convert a list of vertical spans to a list of horizontal spans. This is what all those "while" loops in there are for. The reason I do this is because it is much faster to texture-map unsloped ceilings and floors in the horizontal direction. This is a critical optimization in the engine that you seem to have overlooked. I have seen similar while loops in the Doom source code as well.*

Ken sent an evaldraw script, span_v2h.kc that shows how ceilscan() and florscan() convert a list of vertical spans to a list of horizontal spans.:



displayrest method :

`displayrest` is from the *Game Module*. The major cost is once again drawing (the weapon). The status bar is not drawn each frame, it is only patched when the ammo counter is updated.

VGA and VESA Trivia

Most people ran *Build* in 320x240 thanks to VGA's X-Mode. But the engine also supported Super-VGA resolution thanks to the VESA standard. The Vanilla source code is packed with VESA code that allowed portable resolution detection and rendition.

The nostalgics can read about VESA Programming in this good tutorial. Today all what remains in the code is method names such as `getvalidvesamodes()`.

Sound engine

Duke3D had a very impressive sound system engine at the time. It was even able to mix sound effect in order to simulate reverberation. Read more about it in reverb.c.

Legacy

Build was very hard to read. I listed all the things making it difficult in the next page:

Duke Nukem 3D and Build engine: Source code issues and legacy

Recommended Readings

- If you want to read about another game that was portal based, I recommend this excellent Post-Mortem by Sean Barrett about Thief: The Dark Project.
- Many interviews of Ken Silverman :
  - 21 November 2005 from classicdosgames.com (mirror).
  - 2005 from strifestreams.com (mirror).
  - February 27, 2006 from 3drealms.com (mirror).
  - March 25, 2010 misterdai.yougeezer.co.uk (mirror).
  - May 01, 2012 videogamepotpourri.blogspot.ca (mirror).

Comments