A Brief Summary of DOOM-style Rendering

Robert Forsman and Bernd Kreimeier

July 24, 1996

Abstract

This article serves as a brief summary and first introduction to the idea underlying DOOM-style rendering. It discusses storing the scene geometry using LineDefs, SideDefs and Sectors, and sketches how to render a scene described by a WAD file. Its purpose is to introduce the main concepts of restricted geometry rendering. A list of references suggests further reading.

Introduction

DOOM by id Software combined a couple of well-known and already used techniques with newly invented ones in a unique way. While upcoming 3D hardware acceleration allows for arbitrary scene geometry, restrictions of one kind or another are still valuable and sometimes necessary ways to trade versatility for performance or algorithmic simplicity.

The main restriction of DOOM-style rendering is a static environment that can be represented as a 2D projection. DOOM used additional restrictions, e.g. did not allow for sloped floors and slanted walls, but these are not as important as the restriction to a world that is, in principle, only two-dimensional in structure while obviously three-dimensional in visual appearance.

To discuss this in further detail, we fill first briefly discuss the representation of the scene geometry, as used in DOOM. A full summary can be found in [2].

The worlds of DOOM

The DOOM scene geometry is stored in several lookups for Vertices, LineDefs and Sectors. During BSP building, additional descriptions are generated. In this document we refer to SubSectors and LineSegs, but ignore Nodes. Placement of objects requires yet another lookup.

Vertices

Vertices are two-dimensional. This means that multiple vertices of the 3D scene implied are represented by one projected vertex in the XY plane. The Z coordinate is implicit, because it has to be obtained from the Sectors.

LineDefs and SideDefs

LineDefs are edges in the XY plane, defined by a start and an end vertex. Thus LineDefs have a direction, and a right and a left side. Every LineDef has one or two SideDefs. If a LineDef only has one SideDef, then this must be the right SideDef. Consider the following figure Fig. 1. This LineDef only has one SideDef and it is the right SideDef (if you stand on the first point and face the last point, the Sector is on your right). Note that Sectors are in turn referenced by SideDefs.

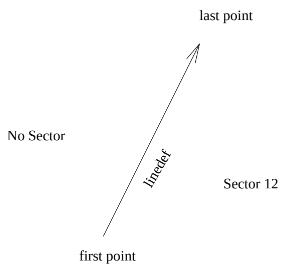


Fig. 1

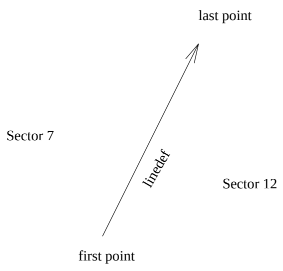


Fig. 2

In Fig. 2, the LineDef has two SideDefs. A Sector 7 is on the left and a Sector 12 is on the right side of the LineDef.

The LineDef represents the geometry of the scene. It could be viewed as a projection of one, two or three rectangular polygons along z, to the XY plane. The polygons are implicitly defined by the data in the LineDef, its SideDefs and adjacent Sectors. In consequence, DOOM describes up to three 3D rectangles and eight 3D vertices by two 2D vertices and four z heights.

Sectors

A Sector is an 2D area completely surrounded by LineDefs, and referenced by the LineDefs' SideDefs. As an example, consider the following Sectors:

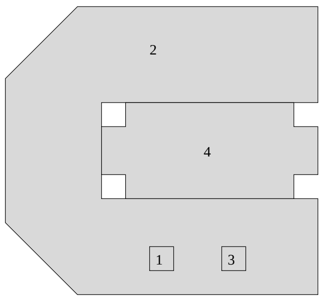
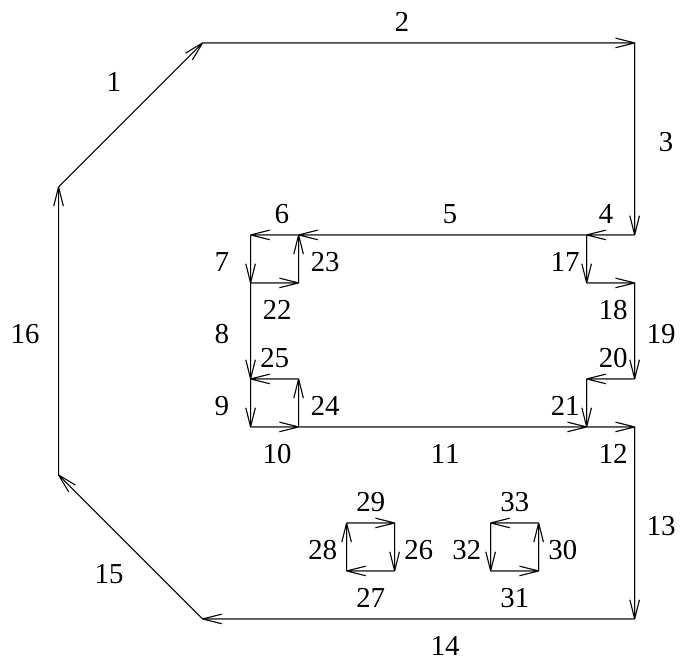


Fig. 3

You will notice that this area is reminiscent of the starting area in E1M1. There are 4 pillars that aren't sectors, and there are also two sectors (1,3) that are **inside** sector 2.

These Sectors are not necessarily convex polygons, but could be concave, or even have holes. In addition, as mentioned below, LineDefs could have two SideDefs referencing the same Sector, i.e. have the same Sector adjacent on both sides.

Let's look at the numbered list of linedefs that are used to define the sectors:



And here's a table of sectors referred to by sidedefs of a linedef. Pay special attention to the difference between linedefs 26-29 and linedefs 30-33.

LineDef number	left Sector	right Sector		LineDef number	left Sector	right Sector
1	-	2		17	-	4
2	-	2		18	-	4
3	-	2		19	-	4
4	-	2		20	-	4
5	4	2		21	-	4
6	-	2		22	-	4
7	-	2		23	-	4
8	4	2		24	-	4
9	-	2		25	-	4
10	-	2		26	2	1
11	4	2		27	2	1
12	-	2		28	2	1
13	-	2		29	2	1
14	-	2		30	3	2
15	-	2		31	3	2
16	-	2		32	3	2
				33	3	2

Note, however, that there can be two-sided LineDefs with SideDefs referencing the same Sector on both sides: the partly transparent grate texture walls on E1M1 or E1M3 are a prime example.

Upper, Lower, Middle Textures

While the LineDefs and Sectors completely define the scene geometry, the SideDefs (as well as the Sectors) determine the visual appearance of the world. DOOM-style rendering relies heavily on texture mapping to give the world detail and a solid appearance. For each side of a LineDef, described by a SideDef, there are three possible surfaces to be textured. These are rectangular polygons always perpendicular to the XY plane, and are named *upper*, *lower* and *middle* (or *normal*) texture. For each, texture coordinate alignment in both directions can be controlled explicitly by offsets, and implicitly by pegging attributes representing some natural alignment given by the scene geometry. The dimensions of the three surfaces are given by the LineDef and the adjacent Sectors: the length is always the LineDef's length, and the height is given by floor and ceiling heights of Sectors on one or both sides. For details see [2] or the WebView3D description [3]. Note that WebView3D does not allow for differently textured upper and lower textures, and that there are no middle textures on two sided LineDefs in WebView3D.

Floor and Ceiling Textures

The Sectors define 2D areas which are textured as well, the floor, and the ceiling. There is only a fixed, natural texture alignment given by a fixed size world coordinate aligned XY grid.

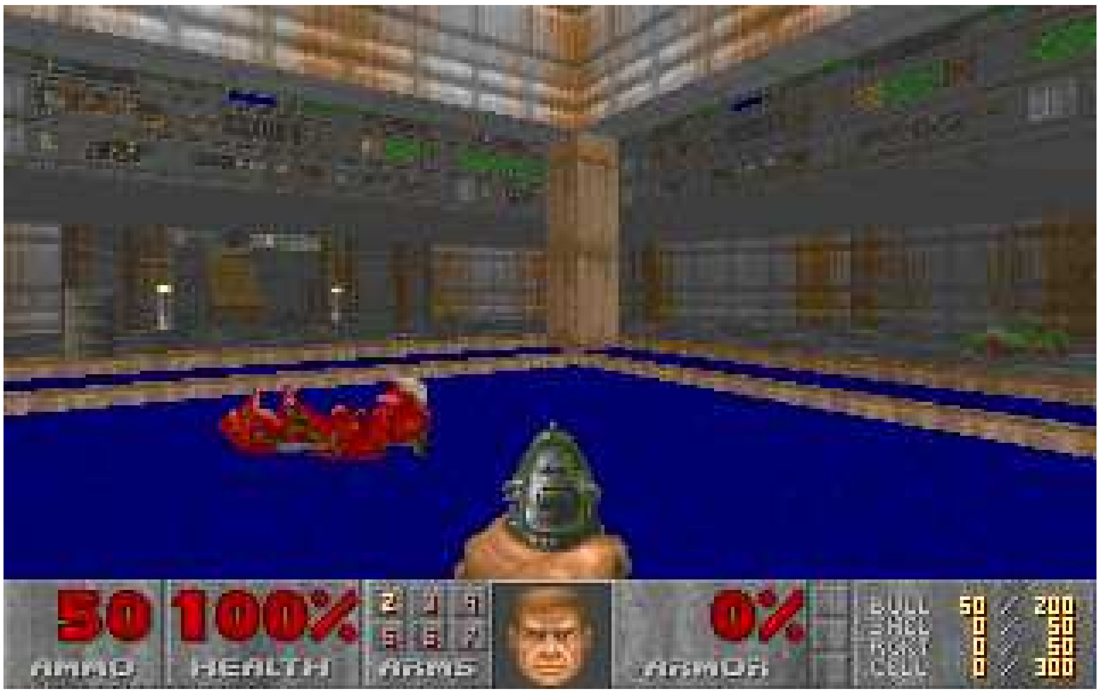
Viewing the world of DOOM

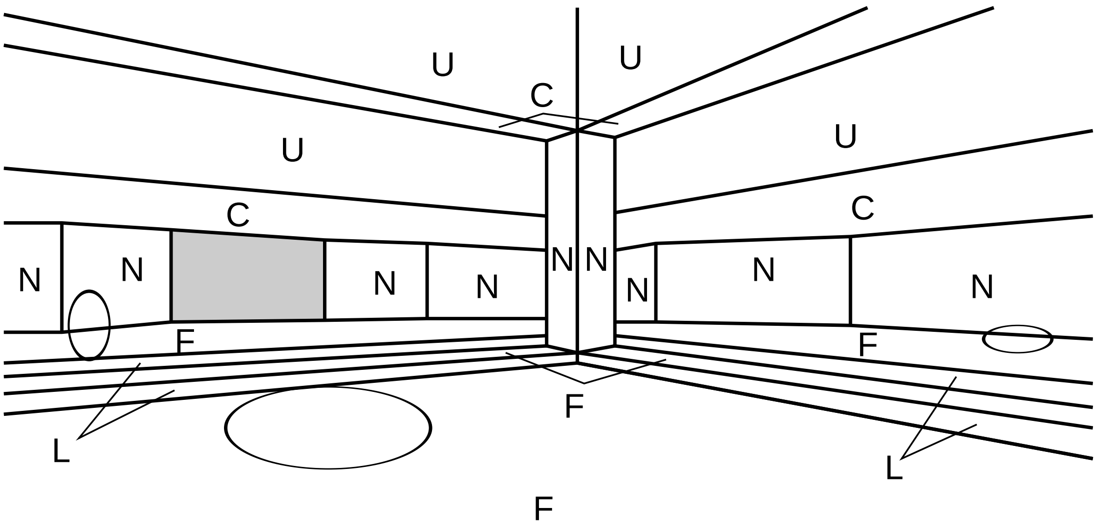
For reasons discussed later, a renderer benefits from the restriction to a 2D representable scene geometry only if accompanied by a restriction of possible directions of view. DOOM-style rendering is often called 4.5 degree of freedom rendering. This means that the POV has three degrees of freedom of movement in X,Y,Z, i.e. arbitrary translation is possible (neglecting collision detection, of course), but that only one full degree of freedom in terms of rotation is available: around the Z axis, changing the azimuth angle. A second rotation for limited looking up and down can be faked by shearing, but a roll rotation around the view axis is prohibited.

Note that these restrictions are partly due to the restricted geometry, as we will see below, while others are simply related to special case texture mapping. In addition, restricted view rendering allows for using a very straightforward cylindrical mapping for view position independent background, i.e. *sky* textures.

A Sample Scenery

Here is a sample from the well-known E1M1 map, right at the start. Witness overlays in the screen plane (status bar, weapon sprites), and partly transparent billboard objects (the barrel, guts, a dead player sprite).





Now consider the screenshot and its schematic in the terms coined above: F is a floor. C is a ceiling. N is a normal or middle texture. U is an upper texture. L is a lower texture. Some billboards are marked as ellipsoids.

A few obvious observations: floors and ceilings could never be adjacent. There are no visible floors above the middle row of the screen, and no visible ceilings below the middle row of the screen. Ceilings might be separated by uppers, floors are separated by lowers. A middle always separates a floor and a ceiling. Taking occlusion into account, a lower or upper from a pillar might obscure anything behind. An opaque middle obscures everything behind by definition. Note that single-sided LineDefs have to carry an opaque middle on their right SideDef. Partly transparent textures are only allowed on middle textures of double sided LineDefs.

DOOM-style Rendering Basics

Binary Space Partition

This introduction does not discuss BSP, as there is exhaustive material available, be it in books [1], on the web [6] or FTP [7]. The 2D BSP used by DOOM partitions possibly non-convex Sectors, creating SubSectors that are guaranteed to be convex planar 2D polygons. In the process, it uses LineDefs as partition lines, and splits other LineDefs to LineSeg segments. During BSP traversal, SubSectors are processed in front to back order, and for each SubSector the texture and surface data are obtained by referencing the LineDef and the SideDef and the Sector, starting with each LineSeg found in the SubSector. All the discussion above is still valid, except that the width of the actual surface rendered is given by the length of the LineSeg, not the LineDef's length.

Spanning Scanline

There is an alternative rendering approach that does not use a BSP, which has been used e.g. by Chris Laurel's "wt". This is in principle an edge-sorted rasterizer rotated by 90 degrees, as, for reasons discussed below, the scanlines are not screen rows, but screen columns in this case. Sorting is done for each frame, but is based on sorting 2D LineDefs instead of sorting upper, lower, middle surfaces separately.

Special case Texture Mapping

In the PCGPE [7] you will find many examples of affine texture mapping. This approach does not give perspective correct texture mapping in general. As the latter requires a very expensive division per pixel, it is a common idea to restrict to those cases in which affine texture mapping happens to be correct. This is discussed in detail in [4], and the basic idea is that affine texture mapping yields correct results as long as we are proceeding along a slice of constant z depth in screen space. Neglecting free direction texture mapping, this enforces familiar restrictions on scene geometry and view: the walls have to be perpendicular to the XY plane (thus each wall slice of constant z distance maps to a screen column), and floors and ceilings have to be in the XY plane (thus each screen row has constant z distance as well on a floor or ceiling).

Obviously, rendering floors and ceilings is more expensive (we are proceeding along a diagonal in texture space in most cases), and it is more complicated with respect to clipping, as we will see. Sean Barrett [4] gives an inner loop in assembly and mentions the necessity to store floor/ceiling textures in a particular way, a technique sometimes called texture interleaving or pixmap interleaving - the basic idea being to have each texture row starting on a 256 byte offset, which allows for fast computation of the current pixel's address.

Within this document, the most important observation is that rendering walls with perspective correct texture mapping could be done as fast as it gets, as long as the view is not allowed to roll or tilt, and as long as we do not have to deal with sloped or slanted surfaces with textures. In this case, texture mapping a wall slice to a screen column is equivalent to scaling the texture. Note that limited looking up/down is done by shifting the vertical slices within the screen columns up or down (including clipping), an operation sometimes called shearing that distorts the view but does not require changing the rendering approach.

Per column rendering

We learned so far that wall rendering is best done per screen column. One problem is to determine the start and end column indices from the projection of the vertices, a problem that has to be solved in 2D only. A fast approach as used by DOOM is described in [5]. So we know how to determine which screen columns we have to handle for a given LineSeg or LineDef.

How is a screen column done, then? Both the BSP and the scanline approach have in common that, somewhere down in the rendering loop, operations are performed per screen column (with 3D rendering, you would prefer rendering per screen row, for obvious reasons). From the schematic we learned that within the DOOM world, we start with a ceiling or an upper texture or a middle texture in the topmost row, and with a floor or a lower or a middle texture in the bottom screen row. No matter what sequence exactly, we know that a middle will fill the column finally.

The most important aspect of front to back rendering is to stop processing the world as soon as the frame is done anyway, i.e. all other surfaces will not be visible. So how do we clip our surfaces efficiently?

Floating Horizons

The BSP approach processes surfaces in several columns, while the scanline approach by its very nature deals with each screen volume separately, until it is done. For our discussion this only means that the BSP approach needs to have the same lookup used per column in as many copies as there are screen columns. Anything else is identical.

It is obvious that, given the restrictions to the scene geometry described above, the view will grow from top down, and bottom up during front to back rendering. The part of the screen row that is still empty is always one single slice in the middle of the column, between the top slice already filled by ceilings and upper textures, and the bottom slice filled by already drawn floors and lower textures. As soon as a middle texture is drawn, we are down. However, there is no guarantee that any middle texture is visible, it might have been obscured by an upper or lower texture already, or even a floor or ceiling.

To handle which part of the scenery obscures other parts further away, we introduce to clipping horizons within the screen column: two indices that indicate the topmost pixel (from the bottom) and the bottom pixel (from the top) that has already been drawn. Prior to texture mapping, we simply clip the vertical slice of the wall we want to render against these floating horizons. As soon as the top horizon is identical to or below the bottom horizon, we are finished with this column. Using a BSP, we increment a counter and check against the total number of columns, to see if the frame is complete. With a scanline approach, we proceed to the next screen column until the leftmost (or rightmost, depending) is done.

Partial Transparency

Obviously, the simple clipping approach used above does not work with partly transparent wall textures or billboard objects (sprites), for the same reason we cannot handle multiple floors and ceilings: in both cases, our single empty slice in the screen column would have to be split into two or more separated slices.

It is possible to use a span buffer, essentially a specialized list of already drawn slices for each column, which has to be done if you want to add multiple floor and ceiling extensions, or include polygon objects in the front to back rendering pass. DOOM-style rendering sticks to the restrictions, and the simple clipping. Partly transparent walls and sprites are clipped against the floating horizons as soon as they are encountered, without updating them, and put on a stack. They are processed separately in a back to front rendering pass as soon as the world has been done by a front to back pass.

Floors and Ceilings

The acute reader will by now have recognized that this descriptions is missing something essential - namely the rendering of floors and ceilings. As has been explained above, floors and ceilings have to be rendered in horizontal slices to get correct results by affine mapping. But our clipping is done by column, not by row.

It happens that this is the most difficult thing in writing a DOOM-style renderer. Chris Laurel's "wt" used a floodfill approach that did not work with partly transparent billboards, and restricted to flat floors and ceilings that could be done in vertical slices, just like walls, in the final release. Philip Stephens' BSP-based version of WebView3D used perspective correct texture mapping with vertical slices, which trades speed for algorithmic simplicity. The Difference Engine renderer never did floor and ceilings textures in the scanline version. There is a working flat shading source that does floors and ceilings in horizontal slices using the BSP, written by Jake Hill, that should serve as a demonstration of how it could be done. The basic idea is to fill all rows in the space given by the current and the old top (or bottom) clipping index across the space spanned by the current LineSeg's projection to the screen.

Summary and Comment

DOOM-style rendering comprises a neat example of how different techniques and restrictions fit together to a streamlined design. Restrictions on degree of freedoms affect both scene geometry representation and world to view transformation overhead, as well as clipping and texture mapping requirements. While there are many examples of extending a DOOM-style approach to include lots of special cases hacks to soften the restrictions, or fake more general scene representation, the approach really does work well by being strict.

On the other hand, certain restrictions are serious design flaws in retrospect - the 2D collision detection used by DOOM is a prime example. Neglecting the current height of objects introduced some serious animation and behavior artifacts that even spoiled gameplay, and the performance gains are negligible.

Even with upcoming 3D hardware, DOOM-style rendering in a more general sense will have its place. What should be learned from DOOM is that restriction to a 2D representation of the environment is perfectly acceptable for first person perspective games, for a lot of reasons including limited user interface interactions, kinesthetics, and the fact that within a supposedly man-made virtual world with gravity, there is not much that is essentially or necessarily 3D.

Improved DOOM-style rendering might use Potential Visible Sets for 2D BSP, which are easier to approximate in 2D, and 2D based approximations of realtime incremental radiosity. Realtime update of a BSP will be feasible for 2D a lot earlier than in the 3D case. Polygon objects are possible using a simplified span buffer for clipping. Using a z-fill during the front to back rendering of the static environment does allow for particle animation. Many techniques currently developed for 3D rendering e.g. with id Software's Quake could be used in DOOM-style rendering with a lot less overhead. It might not always be worth the effort with a product in mind, but it will surely prove educational.

Acknowledgments and Copyright

The authors acknowledge the detailed description written by Matt Fell without which this document would not have been possible, and the PCGPE collected by Mark Feldman, therein the tutorial on texture mapping by Sean Barrett.

References

[1] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, *Computer Graphics - Principles and Practice* Addison Wesley, 1990, 1996.

[2] Matt Fell, *Unofficial DOOM Specs*, 1993, 1994.

[3] Philip Stephens, *Writing a fast 3D graphics engine*, 1995.

[4] Sean Barrett, *Texture Mapping*, In: Mark Feldman, PCGPE, 1994.

[5] William Doughty, *One Approach to Real Time Texture Mapping*, 1994.

[6] Bretton Wade, *BSP Frequently Asked Questions*, 1995, 1996.

[7] Mark Feldman, *PC Games Programmer's Encyclopedia*, 1994.