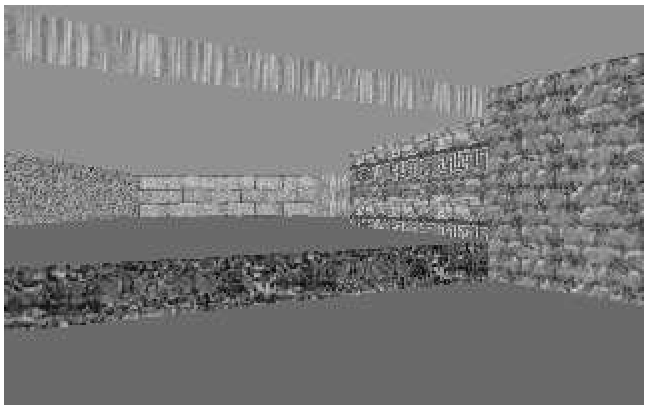
OCamlDoom: ML for 3D action games

François Pessaux
INRIA Rocquencourt
Francois.Pessaux@inria.fr

Abstract

This paper describes a 3D graphics engine with texture mapping for Doom-style computer games entirely written in Objective Caml. This work demonstrates the applicability of ML for interactive computer graphics.



1 Introduction

The traditional area of application for ML is symbolic processing: theorem proving, compilers, ... It has been claimed that ML is ill-suited to other areas by lack of speed and access to low-level machine features. Recent work on network protocols implemented in ML [1, 4] has refuted this claim. In this paper, we attack another stronghold of C (and even assembly language) programming: realtime 3D graphics as found in action computer games such as ID Software's infamous "Doom".

We describe a 3D graphics engine with texture mapping entirely written in Objective Caml. The purpose is twofold: first, give a tutorial introduction to the main algorithms used in this kind of graphics engines; second, study the adequacy of ML for such applications. We show that ML's datatypes and recursive functions are a natural match for those algorithms. We also study the performances obtained with the Objective Caml compiler and compare them against those of a C implementation of the same algorithms. The Caml implementation delivers approximatively 75% of the performance of the C implementation, and achieves a highly respectable frame rate of 100 frames per second on a 333 Mhz Pentium II[1].

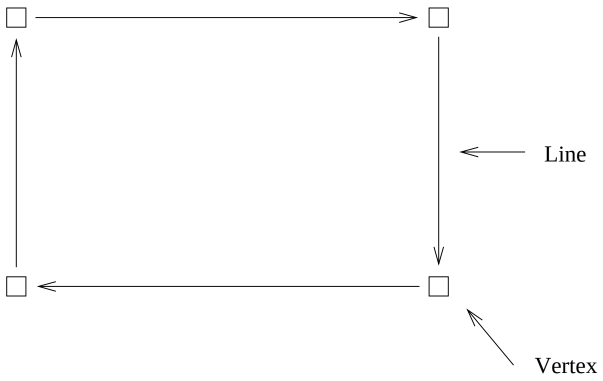
The implementation described in this paper is not state-of-the-art: it is clearly not optimized as it should be in order to make a real game. More powerful algorithms exists to get a more realistic rendering, but are too complex for a tutorial introduction. Finally, we only deal with the rendering of the scenery, but not with sprites, ballistic aspects, nor sound. Still, the model we talk about is a good starting point before going deeper in 3D graphics programming...

The remainder of this paper is organized as follows. We first introduce the basic idea behind Doom-like engines. Then we will inspect the core data structure of the renderer, i.e the BSP tree, and see how it can be implemented in ML. In section 3 we will go deeper in the rendering mechanism to understand how the view is built before being displayed. Section 4 shows performance figures and compare them with the C version of the same program. Section 5 discusses possible enhancements to make the program more efficient.

2 Basics of pseudo-3D rendering

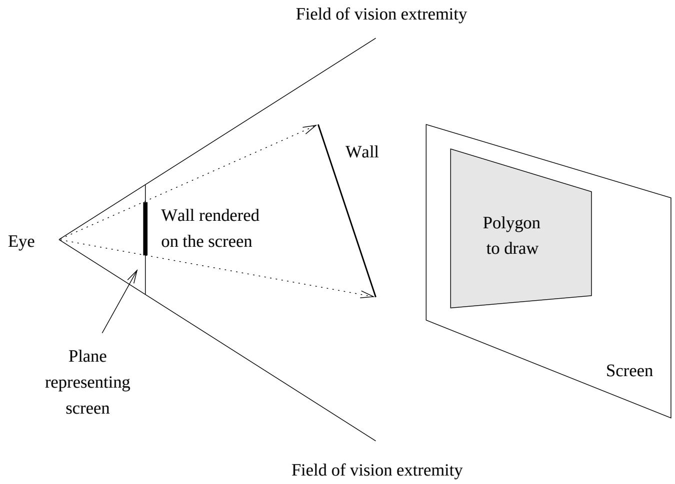
For performance reasons, games such as Doom do not perform full 3D rendering, but restrict themselves to a particular class of models that we call pseudo-3D. In these models, the virtual world where players move is represented as a 2-dimensional map viewed from above, with the third dimension (the height) added afterwards. For instance, here is how a simple rectangular room is represented:

[1] 25 to 30 frames per second is generally considered acceptable, as it corresponds to TV-quality animation.



Such a room, also called a *sector*, is composed of several vertices (4 in our example) connected by oriented lines, also called *linedefs*. These lines define the walls of the room where the player moves. The height is given as a separate attribute of the sector.

To display this room on the screen, each linedef is projected onto the player's field of vision, according to its position in space. Because a linedef represents in fact a surface of constant height, once projected on the plane representing the visible part of the space, it leads to a simple polygon. (This is the reason why these graphics engines are called *polygon-based renderers.*)



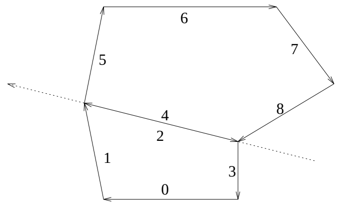
Before going deeper in the projection-rendering process, let us address the problem of knowing which walls are visible from a specific position. This problem is important for two reasons. First, it conditions the correct rendering of overlapping walls: we draw walls starting by those that are closest to the player, and never drawing over a wall that has already been drawn. This way, we can stop rendering as soon as the screen is filled; this would not be possible with a simple painter algorithm, which forces to draw all walls. Second, we avoid computing and displaying walls that lie outside of the player's field of vision.

To address this issue, we represent the scenery by a *binary space partitioning* (BSP) tree. This is a static data structure computed at the time where the virtual world is designed, and saved in the file describing that world.

2.1 The BSP tree

A BSP tree represents a recursive, hierarchical partitioning, of a n-dimensional space. In our case, the space we want to partition is the map of the virtual world. Hence, our space's dimension is 2, and we will partition it with hyperplanes that are 1-dimensional objects, that is, lines.

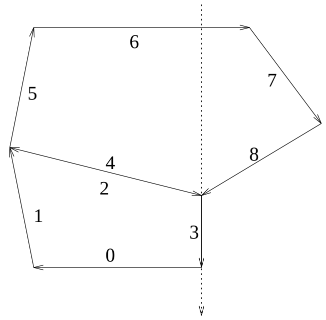
The process for building such a tree is rather simple: choose a partition line l_p and determine the set $\mathcal{L}$ of linedefs in the map which lie on the the left of l_p, as well as the set $\mathcal{R}$ of linedefs in the map which lie on the the right of l_p. Recursively represent $\mathcal{L}$ and $\mathcal{R}$ by BSPs. Finally, create a node, labeled by l_p, with left and right children the two BSPs representing $\mathcal{L}$ and $\mathcal{R}$.



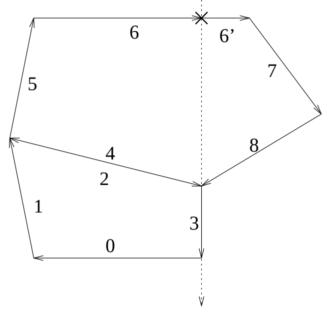
Consider the world depicted above. It is composed of two sectors. Note that on the junction of these two sectors, there are two linedefs of opposite direction, not only one. This is because a sector must be a closed polygon.

To decompose this world into a BSP, we choose a partition line along the linedef 4. Hence, the top node of the BSP tree is labeled by 4, the left child represents the set containing linedefs $\{0; 1; 2; 3\}$, and the right child represents $\{5; 6; 7; 8\}$.

In some cases, the chosen partition line can cross some linedefs. Hence, it is impossible to determine whether these linedefs are on the left or on the right of the partitioning line. For instance, assume we partition the world above along linedef 3. Then, segment 6 is neither on the left nor on the right of the partition line:



In this case, we split the crossed linedefs in two by adding a new vertex at the intersection point. Hence, the left child of the top node (labeled 3) is $\{6'; 7; 8\}$ and the right child is $\{0; 1; 2; 3; 4; 5; 6\}$.



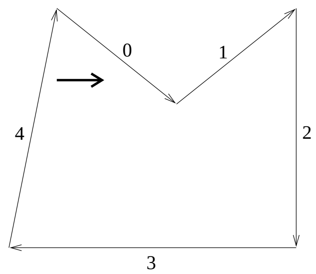
Note that the choice of the partition line strongly influences the shape of the tree. In some applications, the criterion for choosing partition lines can be very important. For the sake of simplicity, in our case, we choose partition lines at random among the linedefs of the world.

To complete the description of the algorithm, we need a criterion to know when to stop partitioning. This criterion depends on the intended use of the BSP. For example, one can decide to stop when a limit is reached on the number of linedefs contained in a leaf, or the tree depth, or the number of splits, or the number of leaves, ... In our application, we stop partitioning when:

- all linedefs in a leaf belong to the same sector,
- and all linedefs in a leaf form a convex polygon[2].

The first point makes sure that all linedefs in a leaf have the same height (because they belong to the same sector, and the height is an attribute of the sector).

The second point makes sure that when drawing walls in a leaf, we can draw them in any order without risk of overlapping. To see that this is not the case when the polygon is not convex, consider the following example, where the viewing direction is the bold arrow.



If we draw linedef 1 before linedef 0, the view looks correct, but in the inverse order the closer linedef 0 is masked by linedef 1 and the result is visually wrong.

Now that we know how to build a BSP tree, let us discuss its relevance to our graphics engine. The first problem was to perform hidden surface removal, i.e to display closest walls first. To achieve this, all we need to do is walk the tree in the following way:

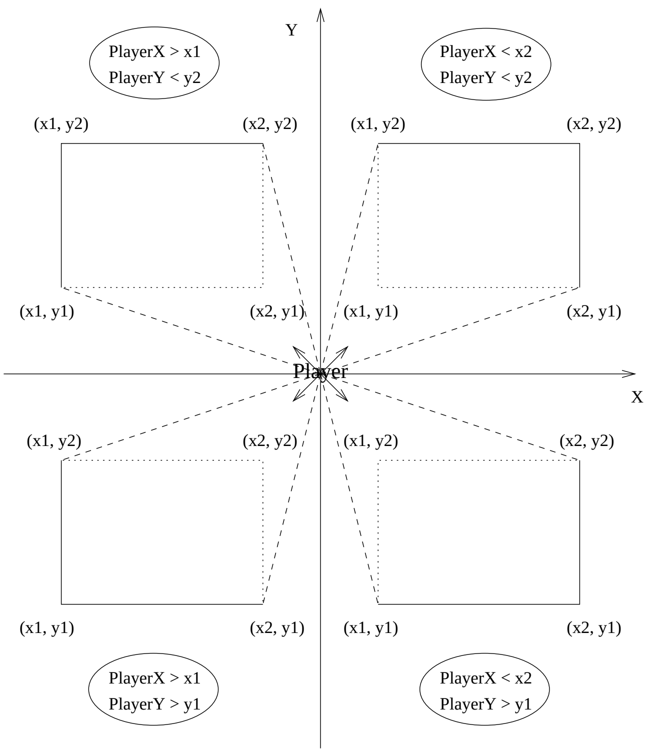
1. Determine player's position
2. If the tree is a leaf, then display each lines of this leaf
3. If the tree is a node, classify the player's position according to the partition line (i.e determine whether he is on its left or on its right).
    - If on the left, then recurse on left subtree, display partition line, recurse on the right subtree.
    - If on the right, then recurse on right subtree, display partition line, recurse on the left subtree.
    - If on the partition line, choose to proceed like one the previous cases.

[2]In some cases, it is possible that there is not enough linedefs at a leaf to form a closed polygon. We also accept non-closed polygons as long as they are "convex" in the following sense: the angle between consecutive linedefs l_n and l_{n+1} must be $\leq 180^o$. This can easily be determined by the sign of a dot product.

It is easy to convince oneself that if two walls overlap viewed from the player's standpoint, then the closer one will be visited first.

Now we still have to address the problem of determining which walls are in the player's field of vision. To handle this in a simple but effective way, we add two bounding boxes (axis-aligned) at each node of the tree, one for each subtree. These boxes contain coordinates of the minimal rectangle bounding linedefs of the left child (respectively right child). So, before recursing in a subtree, we simply check the player's field of vision against the bounding box. This can be achieved quickly and with a sufficient precision using quadrants [7]. To achive this issue, we split the space (let's assume it is centered on the player's position) in four parts along the x and y axis. Then we consider the direction of the leftmost ray in the player's field of vision according to the direction (by block of 90^o) the player is currently looking at.

If we only considerer that the user's cone of vision is the central ray pointing toward the direction the player is looking at, then the following figure shows conditions on relations between player's position and bounding box coordinates to consider this bounding box can be visible (i.e the ray can intersect the box).



In fact, the player's cone of vision is wider that a simple ray. We assume that the angle of this cone is lower than 180^o. By considering the leftmost extremity of this cone, we make sure that all rays in this cone won't cross a quadrant on the left of the one player is looking at. But, depending on the real angle where the player is looking, some rays of the cone can travel into the quadrant just on the right (not more because we assumed the cone to be lower than 180^o). Then to be correct, the test of visibility must not perform, for each quadrant, the two tests given in each part of the figure, but rather the only test common to the quadrant and the quadrant immediately on its right.

So this test of visibility is an approximation which eliminate, for one quadrant, all bounding boxes lying on the other semi-plane than the one defined by the quadrant and the one immediately on its right.

Hence, test visibility is performed by:

```
let is_right_visible playerX playerY
                     playerLeftAngle
                     (x1, y1, x2, y2) =
  if (playerLeftAngle < 90.0)
  then (playerX <= x2)
  else
    if (playerLeftAngle < 180.0)
    then (playerY < y2)
    else
      if (playerLeftAngle < 270.0)
      then (playerX >= x1)
      else (playerY >= y1)
```

2.2 BSP tree structure

The BSP tree structure is elegantly described by the following ML datatype:

```
type tree =
 | Leaf of int list
 | Node of node

and node = {
 (* Label of partition line for the current node *)
 partline : int ;
 (* Bounding box for the left subtree *)
 leftbbox : (float * float * float * float) ;
 (* Left subtree : contains linedefs on *)
 (* the right of the partition line     *)
 left : tree ;
 (* Bounding box for the right subtree *)
 rightbbox : (float * float * float * float) ;
 (* Right subtree : contains linedefs on *)
 (* the right of the partition line      *)
 right : tree
}
```

Note that linedefs are represented as integers, indices into a global array; this makes it easier to save the structure to disk.

The recursive function to walk down the tree in the order described in section 2.1 is also easily written in ML:

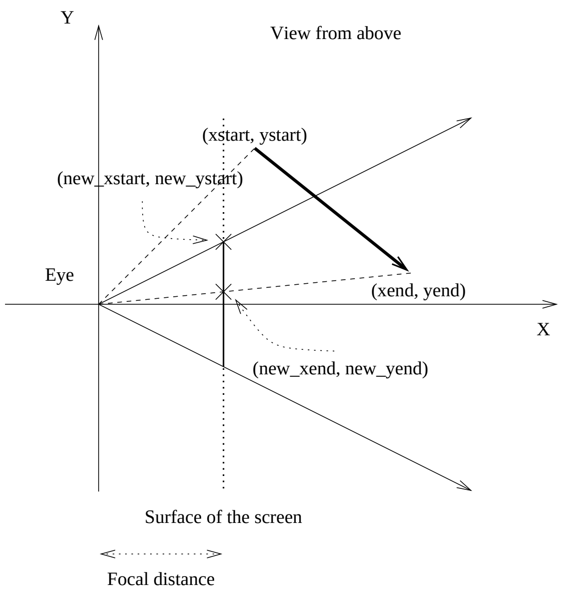
```
let rec front_to_back_tree_parsing tree =
 if not (screen_full ()) then
  match tree with
   | Leaf lines -> List.iter draw_lidenef lines
   | Node nd ->
      (* Get partition line start point *)
      let v1 = lines.(nd.partline).start
      (* Get partition line end point *)
      and v2 = lines.(nd.partline).stop in
      match get_point_position !playerX !playerY
                             v1.x v1.y v2.x v2.y with
        | P_Left ->
           if is_left_visible nd.leftbbox
             then front_to_back_tree_parsing nd.left ;
           draw_linedef nd.partline ;
           if is_right_visible nd.rightbbox
             then front_to_back_tree_parsing nd.right
        | P_Right | P_On ->
           if is_right_visible nd.rightbbox
             then front_to_back_tree_parsing nd.right) ;
           draw_linedef nd.partline ;
           if is_left_visible nd.leftbbox
             then front_to_back_tree_parsing nd.left)
```

3 More on rendering

We now know how linedefs are passed to the renderer. It's now necessary to render individually each linedef as a wall on the screen.

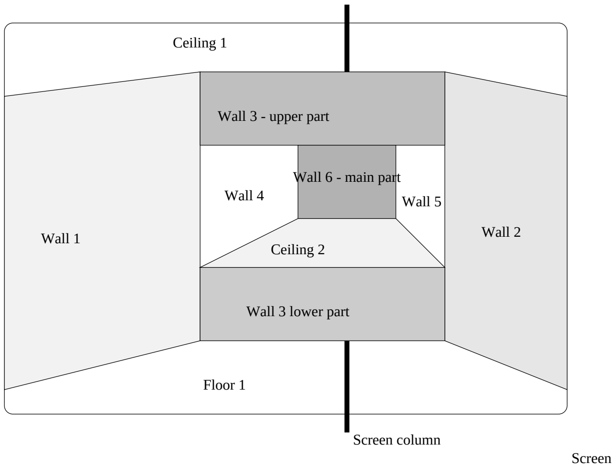
The first step is to transform linedef coordinates into the player's coordinate system. This is simply achieved by a translation plus a rotation on the linedef's vertices coordinates. Then the linedef obtained has to be clipped in case it would be behind the player.

At this point, we know that a part of the wall the linedef represents is potentially visible. We need to project it on the virtual plane representing the screen using a simple mechanism of perspective, in order to get horizontal coordinates of this wall on the screen surface. We now know where this wall will extend horizontally on the surface of the screen, but we still need to clip against the player's field (cone) of vision to remove non visible parts (see following figure).



Once this is done, we know the horizontal extent of the wall on the screen. Now the problem is to get the height of this wall. This height can change along the wall because of perspective effect. In fact, computing the height of the starting and ending wall points is sufficient, a linear interpolation will give us the height in any point between.

The description above is sufficient for drawing plain walls like those of a closed room. In the worlds we try to model, it is possible to have two adjacent rooms, each sharing one of its sides with the other. This corresponds to "doors" and "windows" between rooms. In this case, the limit linedef must not be drawn, otherwise it would look like the player cannot walk from one room to the other. A solution could be to check whether the linedef we want to draw is a two sectors junction, and not to render it in this case. In fact, this solution is not correct because our worlds can have two adjacent sectors with different ceiling and/or floor altitude. So in some cases the junction can look like a step on the floor or on the ceiling (see the following figure).



For this reason each linedef contains 3 different textures, the *upper* texture which is used if the wall shows a upper visible part, the *lower* texture which is used if the wall shows a lower visible part, and the *main* texture which is used to paint the wall if it is plain (that is if it is not a junction between two sectors). We can notice that if a lower or upper texture is needed, the main one is not used. This corresponds to the case where the wall is a sector junction.

When lower and upper textures are used, instead of drawing one plain wall, we need to draw two walls: the upper part of the wall, and it lower part (see figure above). Of course, one of this part can be null (even the two parts, in this case, the wall is simply a junction between two sectors of same ceiling and floor altitude).

Hence, from this description, we see that we need to compute, for each wall, the vertical start and stop position of each extremity of this wall (intermediate values are computed by linear interpolation) before being able to really draw it.

So, the linedef drawing routine in ML looks like:

```
let draw_linedef linedef =
  (* Transform line extremities coords *)
  (* into the viewer space system.     *)
  let (xstart, ystart) =
   rotate_translate lines.(linedef).start in
  let (xend, yend) =
   rotate_translate lines.(linedef).stop in

  (* Check if the wall is completely invisible *)
  if (is_behind_us xstart) && (is_behind_us xend)
  then ()
  else
   let (new_xstart, new_xend, new_ystart, new_yend) =
     clip xstart xend ystart yend in
   (* Project coords on the computer screen space *)
   let scr_xstart = project new_xstart new_ystart in
   and scr_xend = project new_xend new_yend in
   (* Check if the wall is completely out of our   *)
   (* screen. In this case, do not compute anymore *)
   if (scr_xstart = scr_sxend) || (scr_xend < 0)
      || (scr_xstart > screen_width) then ()
   else
    if linedef.maintx <> 255 then
      add_wall new_xstart new_xend
               scr_xstart scr_xend ;
    else
     begin
     if linedef.uppertx <> 255 then
       add_wall new_xstart new_xend
                scr_xstart scr_xend ;
     if linedef.lowertx <> 255 then
       add_wall new_xstart new_xend
                scr_xstart scr_xend
     end
```

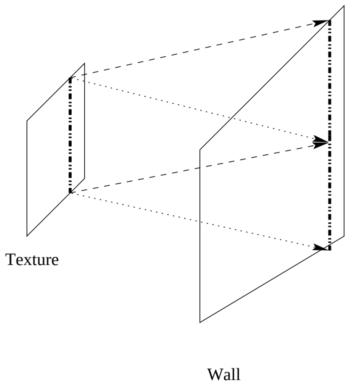
This pseudo code is a simplified version of the real code implanted in OcamlDoom. We can see we took the convention of 255 meaning "no texture used". In the real implantation, before calling `add_wall`, we need to compute a few coefficients used for texture mapping, and vertical wall extremities.

Displaying a wall (done by `add_wall` in the above pseudo-code) consists in drawing it vertical line per vertical line on all its visible length. Each time a vertical line is drawn in a screen column, if this line totally fills the column, we mark this column. If a column is already marked as filled, of course, we don't draw again on it (this is part of hidden face removing). If a column is partially filled, we don't mark it, but we record which part is still unfilled.

A strong invariant of Doom-like worlds is that the unfilled part of a wall is always a contiguous space comprised between the bottom and the top of the screen. Hence we only need to record two integers per column. Notice that adding a wall also draw the ceiling (resp. floor) visible in this column. Because of simplifications used in our engine, floors and ceilings are not textured, so a simple plain vertical line drawing routine is sufficient in this case.

3.1 Texture mapping

In order to get a more realistic rendering, we need to apply textures on these walls. Several texturing techniques exist, varying in quality and complexity. We simply choose to tile an arbitrary bitmap. First we need to know which column of the bitmap maps on each column of the wall. Then, according to the distance from the player to the wall we can determine factors for scaling vertically this column to make it fit the vertical dimension of the wall. Hence, the vertical line drawing consists, for each point of the wall on the screen, in fetching the color of the source point in the texture and to write it on the screen.




```
let vertical_textured_line_draw column top bottom
                                hindex vindex vincr
                                current_bitmap
                                current_bitmap_height
                                current_bitmap_width =
  let index_start = (top * 320 + column)
  and index_end = (bottom * 320 + column) in
```

```
let rec draw index vi =
  if index < index_end then begin
    let color =
      current_bitmap.(((truncate vi)
                        mod current_bitmap_height)
                       * current_bitmap_width
                       + hindex) in
    String.unsafe_set double_buffer index color ;
    draw (index + 320) (vi +. vincr)
  end in
draw index_start !vindex
```

Because of restrictions on the world we model, each time we draw a column for a wall, we can notice that for this column then z coordinate in space remains constant. This is the reason why such kind of engine is known to use *constant Z texture mapping*.

3.2 Screen drawing

To avoid flicker while drawing, we prefer to build the view image in an offline buffer and then blit this temporary image in the video memory of the graphic card. This buffer is a simple character string representing a 320x200 array of points, with 8 bits per point (i.e 256 colors). Note that on slow computers (i.e when building a frame view takes longer than the time the video card needs to refresh the screen), it can be useful to synchronize this blit with the end of video refresh.

This low level access to the video hardware is done using SVGALib under Linux. This library provides a set of C primitives to manipulate SVGA video cards. We only need three of those primitives: obtain the address of the video memory; access the color table of the video card; and wait for the end of screen refresh. The only other part of our engine that is written in C is a simple "blit" function that copies the ML string representing the off-screen buffer into the video memory.

Note that a version running under X-Window also exists and was used to extract the snapshots shown in this paper. The amount of C code is the exactly the same; it only uses X primitives instead of SVGALib primitives.

3.3 Editing worlds

Given the complexity of the data structures, it's obvious that world descriptions cannot be made by hand. For this reason, we also developed a basic editor in Objective Caml, using our CamlTK library [8]. This allows to enter world description using mouse, windows and buttons. Besides this editor, a BSP compiler also exists which takes as input the world description and build the associated BSP tree. Because of the recursive structure of the tree, and its complex data structure, a functional language with high level data types such as ML is very attractive to write such a tool.

4 Performances

In this section, we compare the performances of our ML engine with a C implementation of the same algorithms. The C implementation is functionally equivalent to the ML version, except that it handles only 64x64 bitmaps for textures, while the ML version handles arbitrary bitmaps (GIF images). This allows to replace, in the C version, some modulos and multiplications by masks and shifts. To make comparision more precise, we patched the C engine in order to simulate arbitrary sized bitmap (i.e we removed hardcoded size constants and replaced shifts and mask by their corresponding arithmetic operations). The ML implementation is compiled by the Objective Caml 1.07 native-code compiler, with default settings. Array bound checking is not globally turned off; we only used the "unsafe" version of array access primitives in the vertical textured lines drawing function. The C implementation is compiled with `gcc -O2`.

To compare sources size, even if it is difficult to compare a program written in two different language, we can say that once comments and useless blank lines are removed the C version is about 1000 lines of source, and the ML version is about 700 lines of code. (Note that identifiers have roughly the same names in both versions).

The main performance criterion is the *frame rate*, i.e the number of screen images rendered per second, assuming they are rendered continuously without intervening pauses. The higher the frame rate, the smoother the animation. The frame rate depends obviously on the complexity of the scenery; the measures below are for a simple, but not trivial scenery.

	Objective Caml	C
Pentium 166 Mhz	34 fps	47 fps
Pentium II 333 Mhz	64 fps	81 fps

These figures show that the Caml version is competitive with the C version, despite the fact that we use full ML functionalities such as datatypes, lists and their iterators, and recursion. On the Pentium II, Caml achieves 80% of the performances of C.

Both the C implementation and the Caml implementation give visually satisfying animations, without perceptible pauses or "hiccups". A frame rate of 25 to 30 frames per second is usually considered comfortable.

The first implementations (both in C and ML) used floating point operations everywhere, for the sake of simplicity. Profiling shows that about 80% of the running time is spent in the `vertical_textured_line_draw` function shown above. The inner loop of this function is executed approximately once for each screen point, i.e 64000 times per frame. Examination of the assembly code generated by Objective Caml and by GCC shows that by far the most expensive operation in the inner loop is the `truncate` float-to-integer conversion. This operation is extremely costly on the Intel x86 architecture, as it involves changing the rounding mode of the FPU to "truncate towards zero", then performing the conversion, then restoring the rounding mode. This takes a whopping 55 to 60 cycles on the Pentium II.

To address this bottleneck, we replaced floating-point arithmetic by fixed-point arithmetic in the vertical line texturing function (`vertical_textured_line_draw`). This leads to a significant speed improvement of the engine as shown by the following figures:

	Objective Caml	C
Pentium 166	45 fps	61 fps
Pentium II 333	100 fps	125 fps

Garbage collection accounts for a very small part of the execution time (less than 5%). This is because the renderer allocates all data structures once and for all at initialization-time. Most of the time is spent in the vertical textured line drawing routine. The next most time consuming routine is `add_wall`, i.e the one which computes the dimensions of each wall on the screen and calls the vertical line drawing when needed. The remainder of the renderer, and in particular the traversing of the BSP tree, takes negligible time.

5 Possible enhancements

Of course, our engine is far from performances obtained with the original Doom engine, and it is far from those needed by a real game. But as we said previously it was not intended to be a real game. It was just written to be understandable by beginners, and to be a demonstration. To get a more efficient engine, in both ML and C version, several enhancements can be and should be done:

1. use of fixed arithmetic everywhere instead of floats,
2. tabulate trigonometric functions like $\sin, \cos, \tan$, instead of calling them each time we need them,
3. use fixed power of 2 as dimensions for textures, which would lead to use masks and shifts instead of modulos, multiplications and divisions,
4. explicitly share common sub-expressions while computing data needed by the rendering.

To get a more impressive result, it could also be interesting to add textures onto floors and ceilings. This complicates the engine because floors and ceiling have to be drawn as horizontal lines (not as vertical in the current description of the algorithm). In this way, we keep the *Z constant* invariant we noticed above. Hence, an intermediate structure has to be used to record vertical lines making up the walls and horizontal lines making up floors and ceiling. The difficult part for building this structure lies in recording horizontal runs for floors and ceiling; vertical runs for walls are computed in the same way than we did before (we just record them instead of drawing them on the fly).

A new texture mapper, more complex but more powerful is also needed. We then have to use *correct perspective texture mapping* techniques. Schematically, for each screen projected polygon we want to texture, knowing a screen point location, we must be able to recover its corresponding position on the polygon in the space. Hence knowing how the texture is applied on the polygon in the space, we can determine which point of this texture is used, and so which color to use for this screen point. Because of *constant Z* invariant, depending on wether we are drawing a vertical run or an horizontal run, some computation can be extracted from the inner loop of the mapper, hence reducing the amount of time needed to texture one run.

A version of our renderer incorporating those enhancements is currently under development.

6 Conclusion

Our OCamlDoom renderer demonstrates that ML can also be used for interactive graphical applications, where response time is an important factor and heavy computations are performed in real time. For these applications, using sophisticated data structures and algorithms is as important as raw computing power in achieving good performances; what ML loses in raw execution speed on numerical computations is compensated by the ease with which it handles the complex data structures.

On OCamlDoom, the Objective Caml native-code compiler delivers about three fourths of the performances of an optimizing C compiler. This is consistent with the general claim that good, modern functional compilers such as Objective Caml or GHC stay within a factor of two of C compilers. The difference in execution speed is acceptable in practice, and the time saved in debugging and coding can give ML a great advantage. We have also implemented other graphical applications in Caml: a real-time mouse driven image "warper", and several image processing algorithms. All of them deliver entirely satisfactory performances.

References

[1] Edoardo Biagioni, Rober Harper, Peter Lee, and Brian G. Milnes. *Signatures for a Network Protocol Stack: A Systems Application of Standard ML*. Lisp and Functional Programming, ACM Press, 1994.

[2] Matthew S. Fell. *The unofficial DOOM specs*, April 1994. WEB: `http://doomgate.cs.buffalo.edu/docs/FAQ/DOOM.FAQ.Specs.html`

[3] J. Foley, A. van Dam, S. Feiner, J. Huges *Computer Graphics Principles and Practice*, second edition, 1990. Addison-Wesley Publishing Company

[4] Mark Hayden. *The Ensemble System*, Cornell University Technical Report, TR98-1662, January 1998.

[5] Xavier Leroy, Jérôme Vouillon, and Damien Doligez. *Objective Caml*, INRIA 1998. Software and documentation available at `http://caml.inria.fr`

[6] François Pessaux. *BSP Trees pour la 3D Mappée*, Nov 1996. Available at `http://pauillac.inria.fr/~pessaux/bsparticle.html`

[7] François Pessaux. *Réalisation d'un moteur graphique en pseudo-3D mappée*, January 1997. Software and documentation available at `http://pauillac.inria.fr/~pessaux/engine.html`

[8] François Pessaux and François Rouaix, Projet Cristal. *The CamlTk interface*, INRIA Rocquencourt. Software and documentation available at `http://caml.inria.fr/~rouaix/camltk-readme.html`

[9] Mel Slater. *A Comparison of Three Shadow Volume Algorithms*, The Visual Computer, (1992), Vol. 9(1), 25-38.

[10] *comp.graphics.algorithms newsgroup FAQ* Available at `http://wuarchive.wustl.edu/graphics/graphics/faq/comp.graphics.algorithms-faq`

[11] *Bsp Tree Frequently Asked Questions* `http://reality.sgi.com/bspfaq/index.shtml`

[12] The source code for *Doom* is now available on the WEB (December 1997) `ftp://ftp.idsoftware.com/idstuff/source/doomsrc.zip`

More snapshots