Home * Engines * CPW-Engine * **Move(0x88)**

Both move_make and move_unmake call the functions fillSq(color, piece, square) and clearSq(square). They are meant to encapsulate all the incremental updates (material and pcsq calues, hash keys etc.) and can be viewed in CPW-Engine_board(0x88). Ideally this will make both make and unmake functions independent from the board representation. Beside that, incremental update code will be rather long, if not necessarily ugly.

Code

```
#include "stdafx.h"
#include "transposition.h"

int move_makeNull() {
    b.stm = !b.stm;
    b.hash ^= zobrist.color;
    b.ply ++;
    if (b.ep != -1) {
        b.hash ^= zobrist.ep[b.ep];
        b.ep = -1;
    }
    return 0;
}

int move_unmakeNull(char ep) {
    b.stm = !b.stm;
    b.hash ^= zobrist.color;
    b.ply --;
    if (ep != -1) {
        b.hash ^= zobrist.ep[ep];
        b.ep = ep;
    }
    return 0;
}

int move_make(smove move) {

    /* switch the side to move */
    b.stm = !b.stm;
    b.hash ^= zobrist.color;

    /* a capture or a pawn move clears b.ply */
    b.ply ++;
```

```
    if ( (move.piece_from == PAWN) || move_iscapt(move) )
        b.ply = 0;

    /* a piece vacates its initial square */
    clearSq(move.from);

    /* in case of a capture, the "to" square must be cleared,
       else incrementally updated stuff gets blown up
    */
    if ( b.pieces[move.to] != PIECE_EMPTY )
       clearSq(move.to);

    /* a piece arrives to its destination square */
    fillSq( !b.stm, move.piece_to, move.to );

    /* castle flags
        if either a king or a rook leaves its initial square, the side
 looses its castling-right.
        The same happens if another piece moves to this square (eg.: c
aptures a rook on its initial square)
    */
    switch (move.from) {
    case H1: b.castle &= ~CASTLE_WK; break;
    case E1: b.castle &= ~(CASTLE_WK|CASTLE_WQ); break;
    case A1: b.castle &= ~CASTLE_WQ; break;
    case H8: b.castle &= ~CASTLE_BK; break;
    case E8: b.castle &= ~(CASTLE_BK|CASTLE_BQ); break;
    case A8: b.castle &= ~CASTLE_BQ; break;
    }
    switch (move.to) {
    case H1: b.castle &= ~CASTLE_WK; break;
    case E1: b.castle &= ~(CASTLE_WK|CASTLE_WQ); break;
    case A1: b.castle &= ~CASTLE_WQ; break;
    case H8: b.castle &= ~CASTLE_BK; break;
    case E8: b.castle &= ~(CASTLE_BK|CASTLE_BQ); break;
    case A8: b.castle &= ~CASTLE_BQ; break;
    }
    b.hash ^= zobrist.castling[move.castle];
    b.hash ^= zobrist.castling[b.castle];

    /* castle-move
        in communication with the gui a castling move is represented t
hrough
        the king move. (eg.: e1g1 = White castles short) This king mov
e already
        got executed in the code above with the fillSq() and clearSq()
```

```
 command.
        Whats missing now is the relating rook-move.
    */
    if (move.flags & MFLAG_CASTLE) {
        if (move.to == G1) {
            clearSq(H1);
            fillSq(WHITE,ROOK,F1);
        }
        else if (move.to == C1) {
            clearSq(A1);
            fillSq(WHITE,ROOK,D1);
        }
        else if (move.to == G8) {
            clearSq(H8);
            fillSq(BLACK,ROOK,F8);
        }
        else if (move.to == C8) {
            clearSq(A8);
            fillSq(BLACK,ROOK,D8);
        }
    }

    /* en-passant flag
        First erase the current state of the ep-
flag, then set it again
        in  case there has been a two square pawn move that allows suc
h
        capture. For example, 1.e4 in the initial position will not se
t
        the en passant flag, because there are no black pawns on d4 an
d f4.
        This soluion helps with opening book and increases the number
of
        transposition table hits.
    */
    if (b.ep != -1) {
        b.hash ^= zobrist.ep[b.ep];
        b.ep = -1;
    }
    if ( (move.piece_from == PAWN) && ( abs(move.from - move.to) ==
32 ) &&
          ( pawnRecapture( !b.stm, (move.from + move.to) / 2 ) )
       ) {
        b.ep = (move.from + move.to) / 2;
        b.hash ^= zobrist.ep[b.ep];
    }
```

```
    /* en-passant capture
        if the move is an en-
passant capture, the captured pawn has to be removed manually
    */
    if (move.flags & MFLAG_EPCAPTURE) {
        if (!b.stm == WHITE) {
            clearSq(move.to - 16);
        } else {
            clearSq(move.to + 16);
        }
    }

    ++b.rep_index;
    b.rep_stack[b.rep_index] = b.hash;

    return 0;
}

int move_unmake(smove move) {

    b.stm = !b.stm;
    b.hash ^= zobrist.color;

    b.ply = move.ply;

    /* set en passant square */
    if (b.ep != -1)
        b.hash ^= zobrist.ep[b.ep];
    if (move.ep != -1)
        b.hash ^= zobrist.ep[move.ep];
    b.ep = move.ep;

    clearSq(move.to);

    fillSq(b.stm, move.piece_from, move.from);

    /* un-capture
        in case of a capture, put the captured piece back
    */
    if ( move_iscapt(move) )
       fillSq(!b.stm, move.piece_cap, move.to );

    /* un-castle
        the king has already been moved, now move the rook
    */
```

```
    if (move.flags & MFLAG_CASTLE) {
        if (move.to == G1) {
            clearSq(F1);
            fillSq(WHITE,ROOK,H1);
        }
        else if (move.to == C1) {
            clearSq(D1);
            fillSq(WHITE,ROOK,A1);
        }
        else if (move.to == G8) {
            clearSq(F8);
            fillSq(BLACK,ROOK,H8);
        }
        else if (move.to == C8) {
            clearSq(D8);
            fillSq(BLACK,ROOK,A8);
        }
    }

    /* adjust castling flags */
    b.hash ^= zobrist.castling[move.castle];
    b.hash ^= zobrist.castling[b.castle];
    b.castle = move.castle;

    /* en-passant-uncapture
        put the captured pawn back to its initial square
    */
    if (move.flags & MFLAG_EPCAPTURE) {
        if (b.stm == WHITE) {
            fillSq(BLACK,PAWN,move.to - 16);
        } else {
            fillSq(WHITE,PAWN,move.to + 16);
        }
    }

    --b.rep_index;

    return 0;
}

int move_iscapt(smove m) {
    return (m.piece_cap != PIECE_EMPTY);
}

int move_isprom(smove m) {
    return (m.piece_from != m.piece_to);
```

```
}

int move_canSimplify(smove m) {
    if ( m.piece_cap == PAWN ||
         b.PieceMaterial[!b.stm] - e.PIECE_VALUE[m.piece_cap] > e.
ENDGAME_MAT )
        return 0;
    else
        return 1;
}

// this function returns number of legal moves in the current position

int move_countLegal() {
    smove mlist[256];
    int mcount = movegen(mlist, 0xFF);
    int result = 0;

    for (int i = 0; i < mcount; i++) {

        // try a move...
        move_make( mlist[i] );


// ...then increase the counter if it did not leave us in check
        if ( !isAttacked( b.stm, b.KingLoc[!b.stm] ) ) ++result;

        move_unmake(mlist[i]);
    }

    return result;
}

int move_isLegal(smove m) {
    smove movelist[256];
    int movecount = movegen(movelist, 0xFF);

    for (int i = 0; i < movecount; i++) {
        if ( movelist[i].from == m.from &&
             movelist[i].to   == m.to ) {

                int result = 1;

                // test if the move in question leaves us in check

                move_make( movelist[i] );
```

```
                if ( isAttacked( b.stm, b.KingLoc[!b.stm] ) )
 result = 0;
                move_unmake( movelist[i] );

                return result;
            }
    }

    return 0;
}
```

What links here?

Page	Date Edited
0x88	Nov 28, 2016
CPW-Engine	Dec 31, 2014
CPW-Engine_move(0x88)	Dec 30, 2014
CPW-Engine_root	Sep 27, 2008

Up one Level