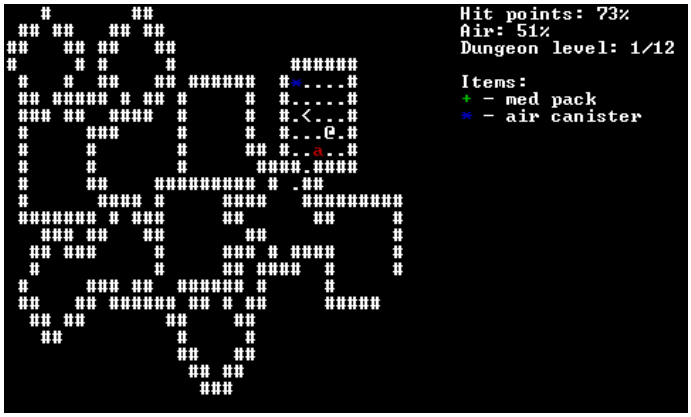
Crypt Rover

You are an archaeologist trapped in an underground crypt with a limited supply of air. You must survive long enough to reach the crypt's exit. Be too rash and the arachnids will nibble you to death. Be too wary and your air supply will run out. Luckily, the ancients have left behind med packs and air cans in the crypt. Use them wisely!

Roguelikes are a type of game that incorporate a couple of different aspects that were popularized by a game game written in 1980 called “rogue.” Fans of roguelike games will differ on what is necessary for a game to be a roguelike, and games that call themselves roguelike are likewise varied. Crypt Rover started as a 7 day roguelike challenge (7drl). The version presented here was written by its author in only 7 days and features random dungeon generation, turn based combat, field of view, as well as the traditional ‘@’ for a protagonist. Since the version presented here the author has made many modifications to the game. While this version remains useful for instruction purposes you can find the latest version of Crypt Rover at:



http://code.google.com/p/cryptrover/

Make sure you have numlock on when you play. You move by pressing the number keys and when you see the exit from the level (<) press the ‘<‘ or comma key to go to the next level. Escape all 12 levels of the dungeon to win!

Crypt Rover is written by Ido Yehieli.

CRYPTROVER.C	You will need the Curses Library installed.

```
#ifdef WIN32
  #include <pdcurses.h>
#else
  #include <ncurses.h>
#endif
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define ESC 27 //ASCII for escape

//entities: the player and his enemies
typedef struct {
  int id,y,x,hp,air;
  chtype type;
  bool awake;
}ent;

typedef struct {
  int y,x;
  chtype type;
  bool used;
}item;

typedef struct {
  int y,x;
  chtype type;
}tile;
```

Listing continued on next page...

```
typedef enum {
  UNSEEN,
  SEEN,
  IN_SIGHT
}view;

//map attributes
#define Y_ 24
#define X_ 48
#define WALL '#'
#define FLOOR '.'
#define NEXT_LEVEL '<'
tile map[Y_][X_];
view view_m[Y_][X_];
//map generation parameters
#define ROOM_RADIUS 2
#define PATHS 5
#define LAST_LEVEL 12
//entities
#define ENTS_ 12
ent ent_l[ENTS_];
ent *ent_m[Y_][X_];
//items
#define ITEMS_ 6
#define MED_PACK ('+'|COLOR_PAIR(COLOR_GREEN))
#define AIR_CAN ('*'|COLOR_PAIR(COLOR_BLUE))
#define MED_CHARGE 6
#define AIR_CHARGE 27
item item_l[ITEMS_];
item *item_m[Y_][X_];
//player attributes
#define PLAYER_HP 30
#define PLAYER_AIR 135
#define FOV_RADIUS 5

int errs;
void init_curses() {
  errs=0;
  if (ERR==keypad(stdscr=initscr(),true))
    mvaddstr(Y_+errs++,0,"Cannot enable keypad");
  if (ERR==noecho())
    mvaddstr(Y_+errs++,0,"Cannot set noecho mode");
  if (ERR==curs_set(0))
    mvaddstr(errs++,X_+1,"Cannot set invisible cursor");
  if (ERR==start_color())
    mvaddstr(errs++,X_+1,"Cannot enable colors");
  else {
    for (int c=0; c<8; c++)
      init_pair(c,c,0);
  }
}

void init_ents(int level) {
  memset(ent_m,(int)NULL,sizeof(ent *)*Y_*X_);
  for (int e=0; e<ENTS_; e++) {
    ent *ce=&ent_l[e];
    ce->id=e;
    ce->awake=false;
    do {
      ce->y=rand()%Y_;
```

Listing continued on next page...

```
      ce->x=rand()%X_;
    } while (WALL==map[ce->y][ce->x].type || NULL!=ent_m[ce->y][ce->x]);
    if (e>0) {
      ce->hp=2;
      ce->type='a'|COLOR_PAIR(COLOR_RED);
    }

    ent_m[ce->y][ce->x]=ce;
  }
  //initial player attributes
  if (1==level) {
    ent_l[0].hp=PLAYER_HP;
    ent_l[0].air=PLAYER_AIR;
    ent_l[0].type='@';
  }
}

void init_items() {
  memset(item_m,(int)NULL,sizeof(item *)*Y_*X_);

  for (int i=0; i<ITEMS_; i++) {
    item *ci=&item_l[i];
    do {
      ci->y=rand()%Y_;
      ci->x=rand()%X_;
    } while (WALL==map[ci->y][ci->x].type ||
             NULL!=ent_m[ci->y][ci->x] ||
             NULL!=item_m[ci->y][ci->x]);
    ci->type=(i<ITEMS_/2?MED_PACK:AIR_CAN);
    ci->used=false;

    item_m[ci->y][ci->x]=ci;
  }
}

void swap(int *i, int *j) {
  int t=*i;
  *i=*j;
  *j=t;
}

int min(int i,int j) {
  return i<j?i:j;
}

int max(int i,int j) {
  return i>j?i:j;
}

int dist(int y0,int x0,int y1,int x1) {
  return pow(y0-y1,2)+pow(x0-x1,2);
}

bool in_range(int y0,int x0,int y1,int x1,int r) {
  return dist(y0,x0,y1,x1)<=pow(r,2);
}

//compare 2 tiles by their distance to the player
int compare_tiles(const void* t1, const void* t2) {
  int py=ent_l[0].y;
```

Listing continued on next page...

```
  int px=ent_l[0].x;
  tile* tile1 = (tile*)t1;
  tile* tile2 = (tile*)t2;
  if (dist(tile1->y,tile1->x,py,px)<dist(tile2->y,tile2->x,py,px))
    return -1;
  else if (dist(tile1->y,tile1->x,py,px)==dist(tile2->y,tile2->x,py,px))
    return 0;
  else
    return 1;
}

int density() {
  int size= Y_*X_;
  int walls=0;
  for (int y=0; y<Y_; y++)
    for (int x=0; x<X_; x++)
      if (WALL==map[y][x].type)
        walls++;

  return 100*walls/size;
}

//line of sight
bool los(int y0,int x0,int y1,int x1,chtype opaque,void(*apply)(int,int)) {
  //Bresenham's line algorithm
  //taken from: http://en.wikipedia.org/wiki/Bresenham's_line_algorithm
  bool steep=fabs(y1-y0)>fabs(x1-x0);
  if (steep) {
    swap(&x0,&y0);
    swap(&x1,&y1);
  }
  if (x0>x1) {
    swap(&x0,&x1);
    swap(&y0,&y1);
  }
  float err_num=0.0;
  int y=y0;
  for (int x=x0; x<=x1; x++) {
    if (x>x0 && x<x1) {
      if (steep) {
        if (opaque==map[x][y].type)
          return false;
        else if (apply)
          apply(x,y);
      } else {
        if (opaque==map[y][x].type)
          return false;
        else if (apply)
          apply(y,x);
      }
    }

    err_num+=(float)(fabs(y1-y0))/(float)(x1-x0);
    if (0.5<fabs(err_num)) {
      y+=y1>y0?1:-1;
      err_num--;
    }
  }
  return true;
```

Listing continued on next page...

```
}

//check if there is enough free space for a room
bool has_space(int y, int x, int radius) {
  if (y-radius<1 || x-radius<1 || y+radius>=Y_-1 || x+radius>=X_-1)
    return false;
  for (int yy=y-radius-1; yy<=y+radius+1; yy++)
    for (int xx=x-radius-1; xx<=x+radius+1; xx++)
      if (FLOOR==map[yy][xx].type)
        return false;
  return true;
}

void dig_tile(int y, int x) {
  map[y][x].type=FLOOR;
}

void dig_path(int y0, int x0, int y1, int x1) {
  los(y0,x0,y1,x1,(char)NULL,&dig_tile);
}
bool dig_room(int y, int x, int radius, bool radial) {
  if (!has_space(y,x,radius))
    return false;

  for (int yy=y-radius; yy<=y+radius; yy++)
    for (int xx=x-radius; xx<=x+radius; xx++)
      if ((radial&&in_range(y,x,yy,xx,radius)) || !radial)
        map[yy][xx].type=FLOOR;

  return true;
}

void dig_level() {
  int new_ry=0;
  int new_rx=0;
  int radius=1+rand()%ROOM_RADIUS;
  //radial or square room
  bool radial=(bool)rand()%2;
  while (true) {
    //continue digging from the last new room or
    //dig the first room in the middle of the level
    int ry=(0==new_ry?Y_/2:new_ry);
    int rx=(0==new_rx?X_/2:new_rx);
    if ((0!=new_rx&&0!=new_ry) || dig_room(ry,rx,radius,radial)) {
      int paths=1+rand()%PATHS;
      for (int p=0;p<paths;p++) {
        int tries=0;
        //try to find an empty space and dig a room there
        while (tries++<10000 &&
                !dig_room(new_ry=ry+rand()%(8*radius)-4*radius,
                          new_rx=rx+rand()%(8*radius)-4*radius,
                          radius=1+rand()%ROOM_RADIUS,
                          radial=rand()%2)
              );
        if (tries>10000)
          return;
        //connect the old room to the new room
        dig_path(ry,rx,new_ry,new_rx);
      }
```

Listing continued on next page...

```
    }
  }
}

void init_map() {
  for (int y=0; y<Y_; y++) {
    for (int x=0; x<X_; x++) {
      map[y][x].type=WALL;
      map[y][x].y=y;
      map[y][x].x=x;
      view_m[y][x]=UNSEEN;
    }
  }
  dig_level();

  //entry to next level
  int ny,nx;
  while (WALL== map[ny=rand()%Y_][nx=rand()%X_].type);
  map[ny][nx].type=NEXT_LEVEL;
}

//radial field of view
void fov(int y, int x, int radius) {
  for (int yy=max(y-radius,0); yy<=min(y+radius,Y_-1); yy++)
    for (int xx=max(x-radius,0); xx<=min(x+radius,X_-1); xx++)
      if (in_range(y,x,yy,xx,radius) && los(y,x,yy,xx,WALL,NULL))
        view_m[yy][xx]=IN_SIGHT;
}

//move entity if there is no living entity on the way
bool move_to(int *y,int *x,int dy,int dx) {
  //don't move into walls
  if (WALL==map[*y+dy][*x+dx].type)
    return false;

  int id=ent_m[*y][*x]->id;
  //if the destination tile has an entity in it
  if (NULL!=ent_m[*y+dy][*x+dx]) {
    ent *de=ent_m[*y+dy][*x+dx];
    //to prevent enemies from attacking one another
    if (0==id||0==de->id) {
      de->hp--;
    } else {
      return false;
    }
    //if it's still alive don't move into its place
    if (de->hp>0) {
      //the move was still successful because of the attack
      return true;
    }
  }
  //remove reference to the entity's old position
  ent_m[*y][*x]=NULL;
  //update entity's position
  *y+=dy;
  *x+=dx;
  //add reference to the entity's new position
  ent_m[*y][*x]=&ent_l[id];
  return true;
```

Listing continued on next page...

```
}

void move_enemy(ent *enemy, ent *player) {
  int *ey=&enemy->y;
  int *ex=&enemy->x;
  if (enemy->awake ||
          (in_range(*ey,*ex,player->y,player->x,FOV_RADIUS) &&
           los(*ey,*ex,player->y,player->x,WALL,NULL))) {
    enemy->awake=true;

    //sort the adjunct tiles by their distance to the player
    tile adj_tile[9];
    int t=0;
    for (int y=*ey-1;y<=*ey+1;y++)
      for (int x=*ex-1;x<=*ex+1;x++)
        adj_tile[t++]=map[y][x];
    qsort(adj_tile,9,sizeof(tile),compare_tiles);

    //move to the closest possible tile
    t=0;
    while (t<9 && !move_to(ey,ex,adj_tile[t].y-*ey,adj_tile[t].x-*ex)) {
      t++;
    }
  } else {
    //sleeping enemies move randomly
    move_to(ey,ex,-1+rand()%3,-1+rand()%3);
  }
}

void print_info(int level) {
  int msgs=errs;
  char msg[50];
  sprintf(msg, "Hit points: %d%%     ",100*ent_l[0].hp/PLAYER_HP);
  mvaddstr(msgs++,X_+1,msg);
  sprintf(msg, "Air: %d%%            ",100*ent_l[0].air/PLAYER_AIR);
  mvaddstr(msgs++,X_+1,msg);
  sprintf(msg, "Dungeon level: %d/%d ",level,LAST_LEVEL);
  mvaddstr(msgs++,X_+1,msg);
  mvaddstr(++msgs,X_+1,"Items:" );
  mvaddch(++msgs,X_+1,MED_PACK);
  addstr(" - med pack ");
  mvaddch(++msgs,X_+1,AIR_CAN);
  addstr(" - air canister ");
}

int end_game() {
  getch();
  exit(endwin());
}

void you_won() {
  mvaddstr(Y_/2,X_/2," YOU HAVE WON! :) ");
  end_game();
}

void you_lost() {
  mvaddstr(Y_/2,X_/2," YOU HAVE LOST! :( ");
  end_game();
}
```

Listing continued on next page...

```
int main() {
  //current dungeon level
  int level=1;

  srand((unsigned)time(NULL));
  init_curses();
  init_map();
  init_ents(level);
  init_items();

  //the player's coordinates
  int *y=&ent_l[0].y;
  int *x=&ent_l[0].x;

  //last key pressed
  chtype key=0;
  do {
    //move player
    if ('8'==key)//up
      move_to(y,x,-1,0);
    if ('2'==key)//down
      move_to(y,x,1,0);
    if ('4'==key)//left
      move_to(y,x,0,-1);
    if ('6'==key)//right
      move_to(y,x,0,1);
    if ('7'==key)//upper left
      move_to(y,x,-1,-1);
    if ('9'==key)//upper right
      move_to(y,x,-1,1);
    if ('1'==key)//lower left
      move_to(y,x,1,-1);
    if ('3'==key)//lower right
      move_to(y,x,1,1);
    if (('<'==key || ','==key) && NEXT_LEVEL==map[*y][*x].type) {
      if (++level>LAST_LEVEL)
        you_won();
      init_map();
      init_ents(level);
      init_items();
    }
    //move living enemies in the player's direction
    for (int e=1;e<ENTS_;e++) {
      if (ent_l[e].hp>0)
        move_enemy(&ent_l[e],&ent_l[0]);
    }

    //use unused item if the player is standing on one
    item* ci=item_m[*y][*x];
    if (NULL!=ci && !ci->used) {
      //heal hp
      if (MED_PACK==ci->type && ent_l[0].hp<PLAYER_HP) {
        ent_l[0].hp=min(ent_l[0].hp+MED_CHARGE,PLAYER_HP);
        ci->used=true;
      }
      //replenish air
      if (AIR_CAN==ci->type && ent_l[0].air<PLAYER_AIR) {
        ent_l[0].air=min(ent_l[0].air+AIR_CHARGE,PLAYER_AIR);
```

Listing continued on next page...

CRYPTROVER.C	Listing continued from previous page...

```
        ci->used=true;
      }
    }

    //mark last turn's field of view as SEEN
    for (int yy=0;yy<Y_;yy++)
      for (int xx=0;xx<X_;xx++)
        if (IN_SIGHT==view_m[yy][xx])
          view_m[yy][xx]=SEEN;
    //mark current field of view as IN_SIGHT
    fov(*y,*x, FOV_RADIUS);

    //draw map
    for (int yy=0; yy<Y_; yy++) {
      for (int xx=0; xx<X_ ;xx++) {
        chtype tile=map[yy][xx].type;
        if (IN_SIGHT==view_m[yy][xx]) {
          mvaddch(yy,xx,tile);
        } else if (SEEN==view_m[yy][xx]) {
          if (WALL==tile)
            mvaddch(yy,xx,tile);
          else
            mvaddch(yy,xx,' ');
        } else {
          mvaddch(yy,xx,' ');
        }
      }
    }
    //draw items
    for (int i=0; i<ITEMS_; i++) {
      if (!item_l[i].used && view_m[item_l[i].y][item_l[i].x]==IN_SIGHT)
        mvaddch(item_l[i].y,item_l[i].x,item_l[i].type);
    }
    //draw entities
    for (int e=0; e<ENTS_; e++) {
      if (ent_l[e].hp>0 && view_m[ent_l[e].y][ent_l[e].x]==IN_SIGHT)
        mvaddch(ent_l[e].y,ent_l[e].x,ent_l[e].type);
    }

    print_info(level);
    ent_l[0].air--;
    key=getch();
  } //exit when the player is dead or when ESC or q are pressed
  while (ent_l[0].hp>0 && ent_l[0].air>0 && ESC!=key && 'q'!=key);
  you_lost();
}
```