SPRAWOZDANIE
z realizacji projektu Grafika Komputerowa

Skład zespołu: Maciej Bandura, Marcin Ślusarczyk, Szymon Śmiglarski; 3ID12B 2022/2023
Temat projektu: Gra horror - silnik typu raycaster.
Środowisko programistyczne (IDE): Visual Studio 2019/ Visual Studio 2022
Wspierane platformy: Windows (x86, x64), GNU Linux
Dodatkowe biblioteki graficzne: Allegro v5.2.7
Koordynator projektu: mgr inż. Daniel Kaczmarski

1. Projekt

Gra została wykonana jako projekt z przedmiotu: Grafika komputerowa. Kod źródłowy napisany w języku C posługuje się biblioteką Allegro 5. Środowiskiem programistycznym użytym podczas prac nad projektem było MS Visual Studio Code (Linux) oraz Visual Studio 2019/2022 (Windows), do linkowania biblioteki Allegro 5 w systemie Windows wykorzystano manager pakietów NuGet wbudowany w Visual Studio. Finalnie gra działała poprawnie na obu systemach - tj, Linux oraz Windows. Do kodu źródłowego został dołączony dedykowany plik makefile.

2. Rozgrywka

Człowiek Korek to nieliniowy horror od którego włos jeży się na głowie. Bohater, młody chłopak „Flop” w poszukiwaniu fortuny ryzykuje swoim życiem. Nocą zapuszcza się w opuszczone „kazamaty” poszukując Legendarnego Korka – średniowiecznego artefaktu, wartego kupę szmalu. Przeklęty szpunt owiany jest mroczną klątwą. Biada temu kto wyciągnie po niego swoje łapska. Dzyndzla strzeże Korkowate Bydle. Człowiek z korkiem zamiast głowy, a w jego ręce ostry jak brzytwa topór. Nikt nie śmie mu się stawić. Jednak chciwość „Flopa” przysłania mu strach przed klątwą. Niestety… Teraz, zagubiony, bez korka w kieszeni i z Korkiem za plecami walczy o swoje życie…

Gra polega na odnalezieniu wspomnianego artefaktu(korka). Ponieważ gracz przemieszcza się po labiryncie, w celu ułatwienia rozgrywki, dodane zostały głosowe podpowiedzi. Schemat dziecięcej gry „w ciuciubabkę”. W odnalezieniu korka, przeszkadza „Człowiek Korek” – potwór z toporem, który próbuje zabić gracza. Potwór szybko biega, lecz wolno przebija się przez drzwi. Przez co gracz musi podejmować decyzję, czy pozostać na swoim kursie, czy też zboczyć z obranej ścieżki do sąsiedniego pokoju i całkowicie się zgubić – za to przeżyć.

3. Zastosowane techniki programistyczne

- Mechanizm opóźnienia wywołania funkcji (scheduler)
- Funkcja do przybliżania dystansu (err ~5%) o stałej złożoności czasowej
- Metoda renderowania obrazu pseudo 3D (raycasting)
- Własny algorytm szybkiej trawersacji przestrzeni blockmap’owej
- Rysowanie sprite’ów w przestrzeni 3D
- Prop Clipping - przycinanie propów za ścianami
- Programowanie modułowo-proceduralne
- Własny manager procesów (pseudo demon)
- Czasomierz słabej rozdzielczości
- Blockmapy

4. Budowa programu

Jednowątkowy silnik implementujący własny menedżer procesów oraz nijako korutyny tj. wątki niemogące wykonywać się współbieżnie. Główny proces tj. ***proc1*** – silnik. Komunikuje się, ze światem zewnętrznym: systemem operacyjnym i biblioteką allegro5. Ponadto obsługuje on czasomierz słabej rozdzielczości. Najwyższym obsługiwanym modułem silnika jest menedżer procesów. (***pm.c***) Działa on na zasadzie nieskończonej pętli, w której wywoływane są procedury *update* wszystkich dołączonych pseudo-procesów. Ich czasy wywołania nie są, w żaden ściśle określony sposób określane ani ograniczane. Wywoływane są w takiej kolejności w jakiej zostały dołączone do *menedżera procesów*. Z nakreślonego designu silnika zawsze pierwszym procesem znajdującym się na liście procesów obsługiwanych przez wspomniany menedżer procesów jest proces reprezentujący silnik (***p_engine.c***). Pojedynczy obrót pętli *PM'a* będzie nazywany ramką czasową silnika. Poszczególne procesy mogą implementować mechanizm planisty (schedulera – ***c_sched.c***). Pozwala on w miarę dokładnie rozkładać poszczególne zadania w czasie. Co jest bardzo pożądane w silnikach gier komputerowych. Ponadto niektóre procesy (***p_render.c, p_game.c***) implementują dodatkowy mechanizm planowania / dzielenia się czasem procesorowym na „sztywno" wykorzystując przy tym deltę czasu silnikowego. Jest to ilość ramek czasowych jaką silnik jest w stanie wykonać w ciągu sekundy. Ze względu na poziom skomplikowania silnika, wartość ta rzadko przekracza 1000 tps. W związku z czym reprezentowana jest ona przy pomocy zmiennej typu **float**.

```
typedef struct engine_s
{
  ALLEGRO_DISPLAY * display;
  ALLEGRO_BITMAP * buffer;
  pm_t * pm;

  unsigned int width;
  unsigned int height;
  unsigned int save_no;

  unsigned int buffer_width;
  unsigned int buffer_height;

  // HRTimer
  unsigned long long ticks;
  unsigned long long last_tick;
  unsigned long long last_clock;
  float delta;
  float refresh;

  bool is_fullscreen;

  struct ipc_s
  {
    int target;
    int code;
  } ipc;

  bool use_captions;
  long gamma;

} engine_t;
```

Delta czasu silnikowego obliczana jest co sekundę w procedurze **update** procesu silnika. Dzielona jest na 1000 w celu uzyskania dokładności w rozdzielczości milisekundowej. Jednak jak to zostało wspomniane powyżej, jest to niemożliwe. Wartość ta nie jest liczbą całkowitą.

```
eng->ticks++;
if (time(NULL) != eng->last_clock)
{
  eng->delta = (float) (eng->ticks - eng->last_tick) / 1000;
  eng->last_clock = time(NULL);
  eng->last_tick = eng->ticks;
}
```

Silnik ekstensywnie wykorzystuje deltę czasu silnikowego do synchronizacji wielu swoich modułów. Począwszy od implementacji planisty, aż po regulowanie prędkości gracza. Poniższy fragment kodu przedstawia wykorzystanie delty czasu silnikowego przez planistę (schedulera) do określania czasu w pseudo-wątku.

```
task_t * ptr = &self->tasks[i];

ptr->delay -= 1 / engine->delta;

if (ptr->delay <= 0)
{
  ptr->funct(self, ptr->ptr, ptr->val);
```

Ponownie poniższy fragment kodu wykorzystuje deltę czasu silnikowego w celu synchronizacji prędkości gracza w czasie. Gdyby delta nie była uwzględniona to gracz poruszałby się ze zmienną prędkością w zależności od platformy i sprzętu – błąd który przypadkowo odkryliśmy testując grę na procesorze AMD Ryzen 5. Uprzednio gra była wyłącznie testowana na procesorach z rodziny Intel.

```
// apply velocity vectors (vx, vy)
const float speed = 0.25;
self->x += self->vx * (speed / game->engine->delta);
self->y += self->vy * (speed / game->engine->delta);
```

Również jedna z najbardziej newralgicznych sekcji programu wykorzystuje deltę czasu silnikowego, jest to proces renderer'a (***p_render.c***). Dzięki niej renderer utrzymuje mniej-więcej stały *frame-rate* na poziomie powyżej 60 FPS. Docelowo silnik usiłuje dostarczyć 120 klatek na sekundę, jest to jednak niemożliwe ze względu na niską rozdzielczość czasomierza. A więc, liczba klatek spada o około 30% i utrzymuje się na takim poziomie, że spadki FPS są niezauważalne dla gracza. Gdyby silnik próbował utrzymywać standardową ilość 60 klatek na sekundę, to amplituda klatek byłaby na wysokim poziomie, przez co spadki byłyby o wiele bardziej zauważalne dla ludzkiego oka.

p_render.c

```
game_t * game = (game_t *) (pm_get(self->pm, "game")->mem);
engine_t * engine = (engine_t *) self->pm->proc_list->mem;

if (engine->refresh <= 0)
  return;

al_set_target_bitmap(engine->buffer);
```

p_engine.c

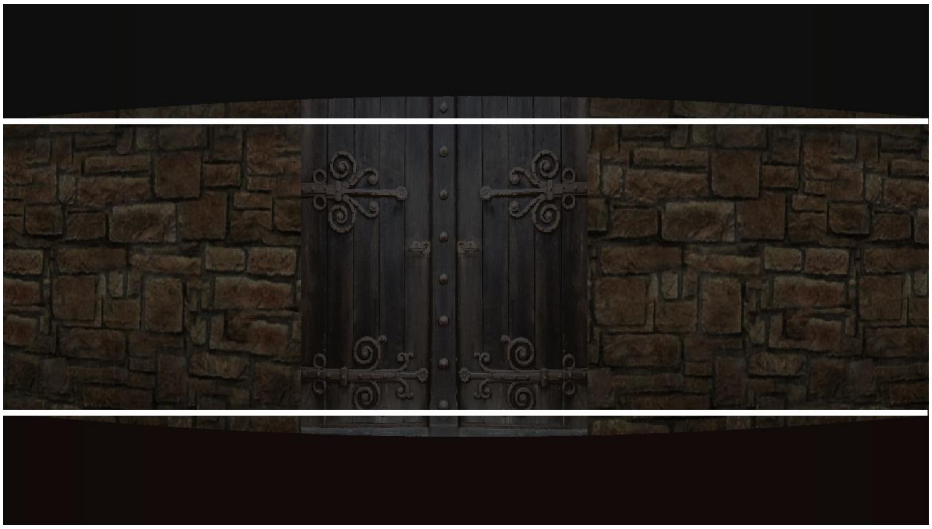
```
if (eng->refresh-- <= 0)
{
  al_flip_display();
  eng->refresh = (1000 / 120) * eng->delta;
}
```

Oczywiście można by sztucznie dostrajać deltę tak aby uwzględniała powyżej wspomnianą różnicę, nie mniej jednak w przypadku procedur odłożonych w czasie na okresy dłuższe niż setne sekundy błąd ten rośnie wykładniczo co czyni czasomierze bezużytecznymi. Co prawda biblioteka **Allegro5** posiada wbudowany czasomierz. Niestety jednak nie działa on wieloplatformowo – porównując jego działanie na systemach Windows i GNU Linux można zauważyć rozbieżności. Które uniemożliwiały uniwersalne synchronizowanie działań silnika na obu systemach operacyjnych.
Najważniejszą funkcjonalnością naszego silnika jest sam rendering. W oparciu o już archaiczną technikę *raycastingu* dostarczamy graczowi, obraz pseudo 3D.

TEORIA

Raycasting polega na *wystrzeliwaniu* z oczu gracza promieni pod odpowiednim kątem, w odpowiedniej ilości. Kąt każdego wystrzelonego promienia będzie kątem obrotu gracza +- kąt odchylenia promienia. Ilość promieni jaką wystrzeliwujemy zależy od szerokości ekranu, natomiast kąt jaki określa interwał między tymi promieniami możemy policzyć, dzieląc pole widzenia przez szerokość ekranu. W naszym silniku zastosowaliśmy system sztucznych buforów. O mniejszej rozdzielczości niż finalna rozdzielczość ekranu. Dzięki czemu jesteśmy w stanie renderować obraz w trybie pełnoekranowym wykorzystując o wiele mniejszą moc obliczeniową. Rozmiar bufora jest konfigurowalny. Można go zmienić w ustawieniach. Gra obsługuje 4 rożne rozmiary bufora: 320x160, 640x320, 1280x720, 1920x1080. Ilość klatek na sekundę nie jest konfigurowalna. Promienie wystrzeliwujemy aż do

momentu napotkania ściany. Zapamiętujemy odległość jaką promień przebył. Odległość ta posłuży nam do zdefiniowania jak wysoko tą ścianę wyrenderujemy na ekranie. Zasada jest prosta, im dłuższy dystans promień przebył tym mniejsza ściana powinna się wydawać. Dokładną wysokość ściany określamy dzieląc pewną stałą (w naszym przypadku wysokość ekranu * 512) przez *przebytą odległość*. Niestety takie podejście wywoła efekt tzw. rybiego oka. Można zauważyć, że gdy stoimy przodem do prostej ściany, to promienie wystrzelone na boki są dłuższe niże te na wystrzelone na wprost.



Widoczny efekt „rybiego oka”

Skalowanie pod kątem prostym do planu patrzenia
– wyeliminowany efekt rybiego oka

Idealnie byłoby wystrzeliwać promienie prostopadle do kąta patrzenia, niestety jednak w momencie kiedy gracz znajduje się blisko ściany, promienie mogą się przez nią przebić, powodując dziwne artefakty graficzne na ekranie. Co więcej wystrzeliwanie promieni pod kątem prostym dałoby w rezultacie projekcję ortograficzną, która nie charakteryzuje się dobrymi aspektami wizualnymi. Wniosek, promienie muszą być wystrzeliwane dokładnie z punktu gracza. Najprostszym sposobem na naprawienie efektu rybiego oka, jest wymożenie uzyskanej odległości jaką promień przebył, przez cosinus kąta patrzenia gracza. W ten sposób efekt rybiego oka zniknie. Natomiast perspektywa zostanie zachowana.

Czysto techniczną ciekawostką jest to, że znając już dokładną odległość i koordynat w jakich promień przecina się ze ścianą, możemy obliczyć w bardzo efektywny sposób (o ile mądrze użyliśmy proporcji do skalowania mapy) równoważną pozycję linii ściany w jej teksturze. Robimy to za pomocą poniższego kodu:

```
long mxs = ((long) rx) & 255;
long tx = (mxs == 0 || mxs == 255)
  ? (((long) ry) & 255)
  : mxs;
```

Powyższy kod korzysta z faktu że logiczna funkcja and jakiejkolwiek liczby całkowitej z liczbą 255 da taki sam wynik jak funkcja modulo tej samej liczby całkowitej z liczbą 256. Jest to jednostka skalowania mapy/ścian które notabene są umieszczone w kwadratowej siatce. Dzięki znajomości koordynatów miejsca kolizji promienia ze ścianą, możemy relatywnie określić offset tego miejsca w teksturze tejże ściany, co też powyższa funkcja robi. Poniższy fragment kodu, rysuje kolejne kolumny ścian.

```
// begin drawing
long r = sqrt(pow(game->player->x - rx, 2) + pow(game->player->y - ry, 2));
long h = 512 * game->engine->buffer_height / (long) (cos(a - game->player->ang) * r);
// long h = 512 * game->engine->buffer_height / (long) r;

// calculate lightness
long s = ((((r << 6) / engine->gamma) << 4) >> 7);

if (s < 0)
  s = 0;
else if (s > 255)
  s = 255;

// draw it and buffer rest
int y_off = ((long) game->engine->buffer_height - h) / 2 + game->player->yoff;
al_draw_scaled_bitmap(game->map->texture_buffer[tile], tx, 0, 1, 256, x, y_off, 1, h, 0);
ZBUFFER[x] = h;
LIGHTBUFFER[x] = (int) s;
break;
```

Trawersowanie przestrzeni mapy przez poszczególne promienie (domyślnie 1280 promieni) 120 razy na sekundę piksel po pikselu (tudzież jednostka po jednostce) byłoby bardzo nieefektywne. Dobrze było by znać punkt intersekcji promienia z siatką mapy w każdym pojedynczym kroku jego castowania. Zaoszczędziłoby to czas potrzebny promieniowi do przebycia pojedynczego tile'a (bloku 256x256). Istnieje wiele metod optymalizacji tej części raycastingu, począwszy od algorytmu DDA aż po nasz autorski algorytm *Ślusarczyk – Bandura***.** Bazuje on na znajdowaniu punktu przecięcia wykorzystując równanie dwóch linii na płaszczyźnie 2D. W pierwszym kroku określa on równanie linii aktualnie wyprowadzanego promienia.

```
for (x = 0; x < game->engine->buffer_width; x += 1)
{
  float rx = game->player->x;
  float ry = game->player->y;

  // find any far point in raycast path

  const float frx = game->player->x + cos(a) * 0xFFFF;
  const float fry = game->player->y + sin(a) * 0xFFFF;

  // basic line equation

  const float ra = game->player->y - fry;
  const float rb = frx - game->player->x;
  const float rc = game->player->x * fry - game->player->y * frx;
```

Algorytm będzie szukał punktów przecięcia z maksymalnie 32 krawędziami. Kolejnym krokiem jest wyprowadzenie równań dwóch linii: krawędzi siatki mapy horyzontalnej i wertykalnej. Krawędzie te są dobierane na podstawie kąta gracza. Do wyprowadzonych równań są od razu obliczane punkty przecięcia przy pomocy funkcji ***intersect()***.

```
// begin traversing plane

for (i = 0; i < 32; i++)
{
  // current ray to destination deltas
  const float dx = frx - rx;
  const float dy = fry - ry;

  // find tiles with possible intersections
  const int bmxoff = ((long) rx) & 255 == 0
    ? (a > ANGN90 || a < ANG90 ? 1 : -1) : dx < 0 ? 0 : 1;

  const int bmyoff = ((long) ry) & 255 == 0
    ? (a > ANG0 && a < ANG180 ? 1 : -1) : dy < 0 ? 0 : 1;

  float vix, viy, hix, hiy;
  intersect(&hix, &hiy, ra, rb, rc, -1 * (float) game->engine->buffer_width, 0, (float) game->engine->buffer_width
            * (float) (((int) rx >> 8) + bmxoff) * 256);
  intersect(&vix, &viy, ra, rb, rc, 0, (float) game->engine->buffer_width, -1 * (float) (((int) ry >> 8) + bmyoff)
            * 256 * (float) game->engine->buffer_width);
```

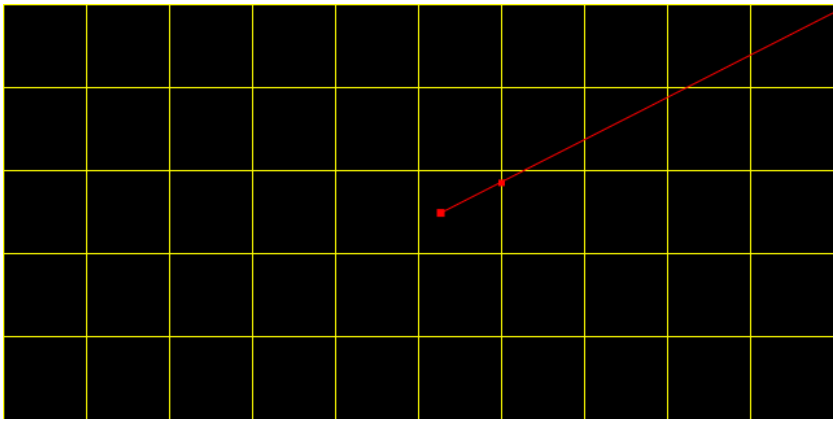
Wybierany jest punkt przecięcia najbliższy do punktu startowego. Użycie rzeczywistej odległości Euklideańskiej byłoby nierozsądne, i znacznie spowolniło proces renderowania. Dlatego też do wymnożonych przez siebie delt nie jest wyciągany pierwiastek kwadratowy, co zdecydowanie skraca czas oraz złożoność obliczeń.

```
// nearest one is the correct one

const float dvx = (vix - rx);
const float dvy = (viy - ry);
const float dhx = (hix - rx);
const float dhy = (hiy - ry);

long omx = (long) rx >> 8;
long omy = (long) ry >> 8;

if ((dvx * dvx) + (dvy * dvy) < (dhx * dhx) + (dhy * dhy))
{
  rx = vix;
  ry = viy;
}
else
{
  rx = hix;
  ry = hiy;
}
```

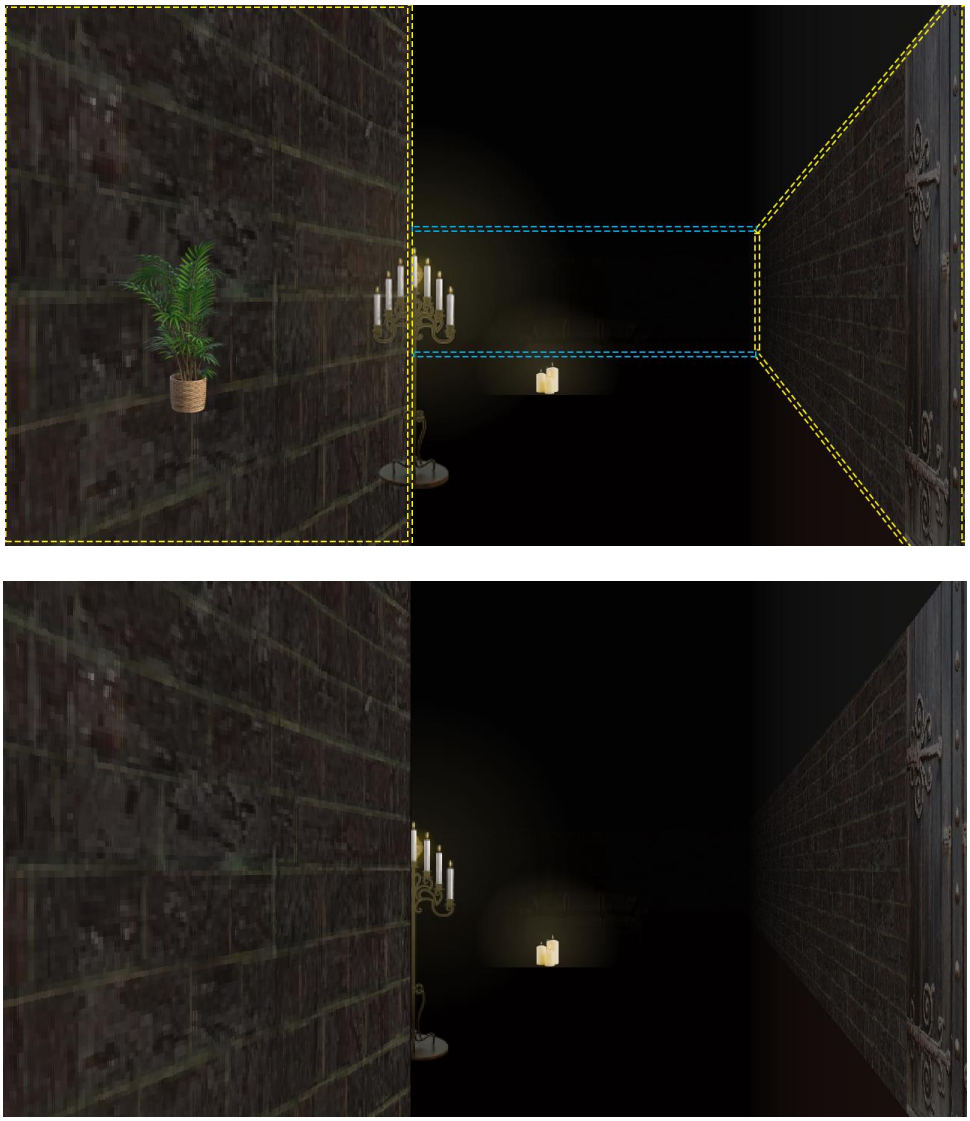
Ze względu na występujące niedokładności obliczeniowe spowodowanie wykonywaniem obliczeń na zmiennych typu **float** otrzymany punkt niekoniecznie leży jeszcze na krawędzi ale tuż przed nią. Należy zatem sprawdzić, czy punkt ten znajduje się w tym samym *tile'u* co wejściowy. Jeżeli tak, to należy pchnąć go o jedną jednostkę do przodu. Po wykonaniu szeregu testów, możemy stwierdzić, iż takie przesunięcie jest wystarczające.

```
long mx = (long) rx >> 8;
long my = (long) ry >> 8;

// unstuck if stuck

if (mx == omx && my == omy)
{
  // one more cast
  rx += cos(a);
  ry += sin(a);
  mx = (long) rx >> 8;
  my = (long) ry >> 8;
}
```

Ciekawostka: Na systemach z rodziny Windows istnieje szansa, że sporadycznie któryś z promieni wystrzeli w nieskończoność poza mapę, co spowoduje systemowe zatrzymanie programu/silnika. Dlatego też, zostało dodane awaryjne zabezpieczenie sprawdzające pozycje promienia na wypadek wystąpienia tegoż błędu. Początkowo wyżej wspomniany algorytm projektowany i testowany był w języku JS. Poniżej znajduje się screen symulujący działanie algorytmu.



Oprócz rysowania ścian, silnik nasz potrafi dodatkowo rysować tzw. propy, czyli proste sprite'y relatywnie do ich pozycji na mapie. Podczas rysowania tych prów, ważne jest aby nie rysować ich fragmentów schowanych za ścianą. Jak w rysunku poniżej:



Niewidoczne odcięte części propów.

W tym celu należy zastosować tzw. Prop Clipping. Algorytm sprawdza jak duża część obiektu(propa) jest zakryta od jego lewej i prawej strony, następnie rysuje tylko tą część która nie jest zakryta. To czy część jest zakryta sprawdzamy, porównując wysokość propa z wysokością wcześniej wyrenderowanej ściany (wysokości te zapamiętujemy w globalnej statycznej tablicy *static long ZBUFFER[1920]*. Same propy natomiast przetrzymywane są w kolejce *SPRITES* reprezentowane jako struktury ***sprite_t*** zawierającej ich pozycję oraz numer tekstury.

```
typedef struct sprite_s
{
  long mx, my;
  float x, y, z;
  uint8_t txt;
} sprite_t;


/*
  Clip from left and right
*/

if (near > far)
  goto __escape_prop_drawing;

if (ZBUFFER[near] > size && ZBUFFER[far - 1] > size)
  goto __escape_prop_drawing;

int clip_left = near;
int clip_right = far;

for (i = near; i < far; i++)
  if (ZBUFFER[i] <= size)
  {
    clip_left = i;
    break;
  }

for (i = far - 1; i >= near; i--)
  if (ZBUFFER[i] <= size)
  {
    clip_right = i;
    break;
  }

const int width = clip_right - clip_left;
```

Ponadto struktura ta zawiera pozycję propa w *tilemap'ie,* na czas rysowania tile z propami są podmieniane na powietrze, zaraz po ich zakolejkowaniu w celu uniknięcia zjawiska wielokrotnego kolejkowania tych samych propów.

```
// HACK: Remove momentarly sprite evidence from map heap
// and restore as soon as finished drawing
game->map->tiles[mx][my] = 0;
sprite_count++;
```

Propy te są przywracane w *tilemap'ie* zaraz po ich narysowaniu. Sama kolejka zaimplementowana jest jako statyczny bufor z ograniczonym rozmiarem do 1024. Takie rozwiązanie jest najszybsze, natomiast implementacja dynamicznej struktury danych opartej o funkcje ***malloc()*** albo nawet i własny alokator pamięci byłaby nieefektywna i nie sprostałby postawionemu zadaniu.

Poza propami z *tilemap'y* silnik jest w stanie rysować tak zwane **vizplane'y,** czyli sprite'y wolnostojące. One same są już oparte o dynamiczną strukturę danych jaką jest lista. Poza statycznym indeksem tekstury zawierają dwa wskaźniki na pozycję obiektu które je wykorzystują, np. Człowiek Korek albo sam item korka. Jako, że ich pozycje mogą się w każdej chwili dynamicznie zmienić, wskaźniki zagwarantują integralność danych.

```
typedef struct viz_s
{
  struct viz_s * next;
  float * x;
  float * y;
  uint8_t txt;
} viz_t;
```

Finalnie wszystkie elementy listy **vizplane'ów** kolejkowane są w buforze *sprite'ów.* Kiedy w ów buforze, znajdują się wszystkie założone sprite'y, to przed ich wyrenderowaniem należy je posortować od najdalej odsuniętego od gracza do najbliższego i w tej kolejności je wyrenderować. Tak aby przedmiot będący najbliżej gracza był renderowany na samym końcu, nad innymi przedmiotami. Oczywiście sortowanie całymi obiektami potrzebowałoby dużej złożoności obliczeniowej przez co jest nieoptymalne. Dlatego też sortujemy samymi indeksami tej tablicy. A dopiero potem korzystając z tablicy indeksów uzyskujemy dostęp do posortowanych elementów kolejki.

```
void draw_sprites (game_t * game) // using globals for optimalization
{
  // sort sprites regarding z
  int i, j;

  for (i = 0; i < sprite_count; i++)
    SPRITE_ADDR[i] = sprite_count - i - 1;

  for (i = 0; i < sprite_count; i++)
    for (j = i; j < sprite_count; j++)
      if (i != j)
        if (SPRITES[SPRITE_ADDR[i]].z < SPRITES[SPRITE_ADDR[j]].z)
        {
          int adr = SPRITE_ADDR[i];
          SPRITE_ADDR[i] = SPRITE_ADDR[j];
          SPRITE_ADDR[j] = adr;
        }
```

Silnik posiada prymitywny system oświetlenia, nie mniej jednak w grze z gatunku horror daje on satysfakcjonujące rezultaty. Polega on na prostym aplikowaniu cienia, z intensywnością bazującą na odległości od gracza, poszczególnym *scanline'om.* Co ciekawe zazwyczaj cieniuje się tylko fragment ekranu na którym rysuje się ścianę, my postanowiliśmy zacieniować cały *scan* od góry na dół, co daje pożądany przez nas efekt głębi mroku wpasowujący się w ramy gry typu horror.

Samo światło, może być parametryzowane w menu za pomocą pozycji: *jasność*. Wyróżniamy 6 stopni jasności.

```
void draw_light (game_t * game)
{
  int x;
  for (x = 0; x < game->engine->buffer_width; x++)
  {
    int light = LIGHTBUFFER[x] * game->map->apprx_lightness / 512;

    if (light < 0)
      light = 0;
    else if (light > 255)
      light = 255;

    al_draw_line(x, 0, x, game->engine->buffer_height, al_premul_rgba(0, 0, 0, light), 1);
  }
}
```

Dynamiczność światła uzależniona jest od propów stojących na mapie oznaczonych jako „lampy”. Patrzenie się w ich kierunku zwiększa ogólne naświetlenie ekranu, co powoduje rozjaśnienie się planszy. Efekt cienia nakładany jest również na propy, lecz propy emitujące światło nie posiadają cienia. Są pomijane podczas procesu nakładania cienia. Również postać „Człowieka Korka” nie jest objęta cieniem, zostało to zastosowane w celu zwiększenia imersji gracza.

```
void calc_lightness (game_t * game)
{
  // find lights in spritebuffer
  game->map->lightness = 512 + 256;

  int i;
  for (i = 0; i < sprite_count; i++)
    if (SPRITES[i].txt > MAP_LAMPS_START && SPRITES[i].txt < MAP_LAMPS_END)
      game->map->lightness = 512 - 32;

  if (game->map->apprx_lightness < game->map->lightness)
    game->map->apprx_lightness += 2;
  else
    game->map->apprx_lightness -= 3;
}
```

Kolejnym ludzkim zmysłem na jaki nasz silnik oddziałuje jest słuch. Gry typu horror charakteryzują się bogatym i różnorodnym udźwiękowieniem działającym na ludzką wyobraźnię. Dlatego też zaimplementowany przez nas moduł audio jest dość rozbudowany.

```
typedef struct audio_s
{
  char name[32];
  unsigned int normal_count;
  unsigned int muffled_count;
  ALLEGRO_SAMPLE * normal[4];
  ALLEGRO_SAMPLE * muffled[4];
  struct audio_s * next;
} audio_t;
```

Poza skalowaniem głośności dźwięku względem odległości punktu jego grania od gracza, sterujemy również panningiem dźwięku, tj. jego stereofonią. W przypadku, jeżeli jakiś dźwięk odgrywany jest z naszej lewej strony to słyszymy go głośniej na lewym kanale. Co więcej każdy dźwięk możemy przechowywać w wielu wersjach. Gdzie przy każdym odegraniu dźwięku losowana jest jego wersja. Dodatkowo silnik umożliwia przechowywanie stłumionej wersji dźwięku (***muffeld***) w przypadku próby odegrania go za ścianą właśnie taką wersję usłyszymy. Jest to wykorzystywane między innymi podczas odgrywania kroków „Korka”.

```
if (self->atacking <= 0)
  try_moving(game);
else
  self->atacking -= 1;

if ((val & 15) == 0)
  audio_play(game->audio, game, "step", self->x, self->y);

if ((val % 40) == 0)
  enemy_chase_update(game);
```

Silnik umożliwia również granie dźwięku nie punktowego tj. dźwięku „na masterze” co jest przydatne np. w odgrywaniu dźwięku ataku na gracza, czy też kroków gracza.

```
// handle player bobbing effect
if (self->vx != 0 || self->vy != 0)
{
  player_bobbing(self, game);
  if (self->walking_audio_timer)
  {
    self->walking_audio_timer = false;
    audio_play_master(game->audio, "step");
    yield(game->sched, player_reset_walking_aut, self, 0, PLAYER_WALKING_AUDIO_DELAY);
  }
}
```

Zarówno rysowane propy jak i ściany mogą być animowane. Rozróżniamy dwa typy ścian i propów: posiadające animacje jak i statyczne. Silnik nasz obsługuje 8 różnorodnych ścian statycznych (o niezmiennej teksturze) oraz 4 ściany animowane posiadające po 4 klatki animacji. Same zaś propy dzielą się na kolejne podgrupy: wyróżniamy propy które emitują światło, propy animowane oraz propy statyczne. Pierwsze dwie grupy posiadają po 4 klatki animacji, ogólnie silnik obsługuje 24 różne propy.

```
void map_load_texures (map_t * self)
{
  char tokens [][0xF][0xF] =
  {
    {"air", {1}}, // 0
    {"sky", {4}}, // 1
    {"wd",  {4}}, // 2
    {"ws",  {8}}, // 3
    {"wa",  {4}}, // 4
    {"psc", {8}}, // 5
    {"pac", {4}}, // 6
    {"plc", {4}}, // 7
    {"pln", {4}}, // 8
    {"psn", {4}}  // 9
  };

   ...

  // -- Load Cork Textrues
  for (i = 0; i < MAP_CORK_SIZE; i++)
    map_load_specific_texture(self, 0, "cork", i >> 2, i & 3, MAP_CORK_BEG + i, true);

  // -- Load Item And Explosion Textrues
  map_load_specific_texture(self, 0, "item", 0, 0, MAP_ITEM_BEG, false);

  for (i = 0; i < 7; i++)
    map_load_specific_texture(self, 0, "expl", 0, i, MAP_EXPL_BEG + i, true);
```

Same zasoby przechowywane w postaci plików wpadają w powyższą konwencję nazewniczą, gdzie kolejno litery oznaczają:
[w – ściana p – prop]
[s – statyczny a – animowany l – emitujący światło d-drzwi]
[c – posiadające kolizję n – bez kolizji]
Zarówno *air* jak i *sky* nie są ładowane z zasobów. Liczby obok oznaczają ilość ładowanych plików. Animacje realizowane są na zasadzie podmieniania wskaźnika w indeksie *tilemap'y* odpowiadającego konkretnej teksturze. Same tekstury przechowywane są w dalszej części *tilemap'y* do której generator map nie ma dostępu. Zarówno przy tworzeniu jak i usuwaniu obiektu mapy przechowującego wszystkie „surowe" tekstury dalsze ewidencje są wskaźnikowo linkowane z tymi umówionymi w dostępnej części *tilemap'y.*

```
void map_unlink_animated_texures (map_t * self)
{
  int offsets [16];
  int i;

  // walls
  for (i = 0; i < 4; i++)
    offsets[i] = MAP_ANIM_WALL + i;

  // props
  for (i = 0; i < 4; i++)
    offsets[i + 4] = MAP_ANIM_PROP + i;

  // lamps
  for (i = 0; i < 8; i++)
    offsets[i + 8] = MAP_ANIM_LAMP + i;

  // now just update 'em
  for (i = 0; i < 16; i++)
    self->texture_buffer[offsets[i]] = NULL;

  // unlink cork
  self->texture_buffer[0] = NULL;

  // unlink explosion
  self->texture_buffer[MAP_EXPL] = NULL;
}
```

Sam proces aktualizacji wskaźnika odbywa się przy pomocy coroutine wywoływanej około 4 razy na sekundę. Podczas niej obliczana jest odpowiedni numer klatki tekstury, czyli jej *offset+indeks bazowy* w tilemapi'e.

```
void map_update_animated_textures (sched_t * sched, void * ptr, int ticks)
{
  map_t * self = (map_t *) ptr;
  int frame = ticks & 3;
  // update texure buffer frames and pointers

...

  self->texture_buffer[MAP_EXPL] = self->texture_buffer[MAP_EXPL_BEG + explosion_frame];

  yield(sched, map_update_animated_textures, ptr, ticks + 1, 250);
}
```

Jak już wspomnieliśmy zarówno ściany jak i propy mogą posiadać kolizję. W przypadku ścian jest ona sprawdzana w bardzo prosty sposób, pozycję gracza konwertuje się do pozycji *blockmap'y* nakładającej się na *tilemap'ę*. Jeżeli w indeksie *tilemap'y* odpowiadającej jej pozycji z *tilemap'y* występuje kolidowalny obiekt (wartość *tilemap'y* większa niż ***MAP_COLLIDABLE_START*** i mniejsza niż ***MAP_COLLIDABLE_END***), to znaczy, że wystąpiła kolizja. Warto wspomnieć, że dla gracza kolizja podczas chodzenia sprawdzana jest dla 4 punktów rozmieszczonych na krawędziach kwadratu dookoła gracza – jego „collidera".

```
for (y = -1; y <= 1; y += 2)
  for (x = -1; x <= 1; x += 2)
  {
    mx = (long) (((int) self->x) + x * 9) >> 8;
    my = (long) (((int) self->y) + y * 9) >> 8;

    tile = game->map->tiles[mx][my];
```

Natomiast, jeżeli mówimy o propach, to w tym przypadku przeprowadzamy dodatkowe sprawdzenie odległości propa od gracza – odległość ta może być o wiele mniejsza niż w przypadku ściany – jeżeli gracz mieści się w zadanej odległości, to dopiero wtedy kolizja zostaje aktywowana.

```
// if prop -> recheck()
if (tile > MAP_COLLIDABLE_PROP_START)
{
  // have to recheck
  if (q_dist((float) mx * 256 + 128, (float) my * 256 + 128, game->player->x, game->player->y) > 128)
     return;
}
```

W silniku istnieje rodzaj specjalnej kolizji, mianowicie – kolizja z drzwiami. W przypadku wychwycenia takiego zdarzenia silnik robi bardzo prostą rzecz, teleportuje gracza po linii kolizji w kierunku drzwi o ściśle określoną stałą. System ten jest nie tylko jest prosty do zaimplementowania, ale też bardzo optymalny w swoim działaniu. Dodaje on dużo urozmaicenia do rozgrywki.

```
if (tile > MAP_DOOR_START && tile < MAP_DOOR_END)
{
  is_door = true;
  is_inside = true;
  goto _skip_check_fast;
}

...

if (is_door)
{
  long ox = mx - ((long) self->x >> 8);
  long oy = my - ((long) self->y >> 8);
  self->x += ox * WALL_JUMP;
  self->y += oy * WALL_JUMP;
  game->map->darkness = 255;
  char door_sample_name[0xF];
  sprintf(door_sample_name, "door%d", tile - MAP_DOOR_START - 1);
  audio_play_master(game->audio, door_sample_name);
}
```

Kolejnym aspektem ważnym do omówienia jest „sztuczna inteligencja” przeciwnika, bazuje ona na prostym algorytmie *trailling'u* – w celu optymalizacji tegoż algorytmu, zaimplementowany został system *mappoint'ów,* są one buforowane w listach jednokierunkowych, przypisanych do poszczególnych elementów mapy blokowej o rozmiarze 32x32 rozciągającej się na całą planszę gry o rozmiarach 511x511. Przeciwnik działa w dwóch trybach, pogoni oraz ucieczki. Jak sama nazwa wskazuje, w pierwszym trybie przeciwnik podąża za graczem, w drugim się od niego oddala. Dodatkowo, jeżeli przeciwnik jest zbyt daleko to jest on teleportowany bliżej gracza.

```
point_t * near = find_nearest_point(game, px, py);

if (near != NULL)
{
  if (get_dist(near->x, near->y, game->enemy->x, game->enemy->y) < 32)
    near->enabled = false;
  ang = atan2(near->y - game->enemy->y, near->x - game->enemy->x);
}

...

if (game->enemy->facing == 1 && dist > CROK_SAFE_DIST)
{
  game->enemy->x = near->x;
  game->enemy->y = near->y;
  near->enabled = false;
  return;
}
```

Dodatkowo przeciwnik może być w jeszcze jednym trybie – trybie ataku. W tym przypadku nie porusza się on wcale, stoi w miejscu i macha siekierą. Jeżeli gracz jest odpowiednio blisko to traci życie. Ponieważ algorytm *trailling'u* jest bardzo prymitywny, istnieje duże prawdopodobieństwo, że przeciwnik przejdzie przez ścianę, postanowiliśmy wykorzystać jego słabość i podczas próby przejścia przez ścianę (lub drzwi, propy z kolizją) przeciwnik chwilowo przełączany jest w tryb atakowania, co wygląda jakby próbował się „przebić” przez jakiś mebel lub drzwi.

```
/*
  Update enemy sprite animation
*/
void enemy_sprite_update (sched_t * sched, void * ptr, int val)
{
  game_t * game = (game_t *) ptr;

 ...

  if (game->enemy->atacking > 0)
    sprite = MAP_CORK_ATTACK + frame;

. . .

else if (game->enemy->atacking > 0)
{
  if (game->enemy->door_check == false)
  {
    long mx = ((long) game->enemy->x) >> 8;
    long my = ((long) game->enemy->y) >> 8;
    uint8_t tile = game->map->tiles[mx][my];
    if (val % 2 == 0)
    {
      audio_play_master(game->audio, (tile < MAP_DOOR_END) ? "cork_door" : "cork_expl");
    }
  }
```

Menu jest nieodłącznym elementem każdej gry. Ręczne hardkodowanie w nim przycisków, biorąc pod uwagę, że jest to głównie projekt silnika jest działaniem niedopuszczalnym. Menu w takich aplikacjach musi być generowane według ściśle określonego schematu – proceduralnie. Postanowiliśmy napisać własne, bardzo proste lecz efektywne api, działające w oparciu o wybraną bibliotekę graficzną oraz strukturę naszego silnika, umożliwiające generacje interfejsu dla użytkownika. Przyciski – najbardziej elementarne części menu – zebrane są one w listy przycisków, które rysowane są na ekranie. Symetrycznie z dośrodkowaniem wertykalnym po lewej stronie ekranu. Każdy przycisk może posiadać dwa stany: *aktywny i nieaktywny.* Ponadto każdy przycisk posiada etykietę (wyświetlaną w trakcie renderowania). Dodatkowo każdy przycisk zawiera odwołanie do funkcji – która zostanie wywołania po

naciśnięciu konkretnego przycisku – o ile dany przycisk nie jest w stanie *nieaktywnym*. Ostatnim elementem wchodzącym w skład przycisku jest jego wartość – liczba całkowita, służy do proceduralnej obsługi renderingu/generowania przycisku dynamicznie zmieniającego swoje stany(*toggle button)*.

```
typedef struct button_s
{
  struct button_s * next;
  void (*event)( struct button_s * self, struct menu_s * menu );
  char text[32];
  int value;
  bool enabled;
} button_t;
```

Jeżeli mowa o podstawowej zasadzie działania procesu menu, oparty jest o strukturę **menu_t,** która zawiera w sobie: po pierwsze obsługę powyżej wspomnianego api, dwa wskaźniki ***buttons, *swap** gdzie pierwszy to lista obecnie wyświetlanych przycisków, natomiast drugi służy do dynamicznego przepinania wyświetlanych przycisków. Ponadto struktura zawiera implementację *schedulera,* umożliwiającego przeciwdziałanie efektu ghostingu przycisków. Poza tym, zawarte są inne niezbędne wskaźniki oraz parametry potrzebne do dostarczenia jak najlepszego doświadczenia użytkownikowi (wyświetlany w tle film, zapamiętywanie pozycji myszki przy przełączaniu menu itp.)

```
typedef struct menu_s
{
  button_t * buttons;
  button_t * swap;
  engine_t * engine;
  sched_t * sched;
  ALLEGRO_FONT * font;
  ALLEGRO_VIDEO * video;
  pm_t * pm;
  bool during_game;
  bool ghosting;

  int lmx, lmy; // remember mouse
                // so player doesn't get roteted
} menu_t;
```

Jeżeli menu jest wyświetlane w trakcie gry, to nie jest wyświetlany filmik w tle, tak jak się to dzieje w przypadku menu uruchomionego bez rozgrywki. Ponadto zmiana ustawień graficznych jest dynamicznie aktualizowana przez działający w tle proces renderera (trwającej rozgrywki bez logiki – tj. z zamrożonym procesem **p_game**).

Ponieważ silnik implementuje mechanizm sztucznych procesów, konieczne było dostarczenie środka komunikacji między nimi. Początkowo wykorzystywaliśmy do tego funkcję **pm_get()** pozwalającą każdemu procesowi na bezpośredni dostęp do zasobów pamięciowych innego procesu. Jednak w przypadku gdy nie wszystkie procesy są aktywne w danej chwili czasowej takowe zasoby mogą nie istnieć albo mogą być dopiero tworzone. Powstaje ryzyko niespójności pamięci, które może doprowadzić do destabilizacji pracy silnika. Zaimplementowaliśmy więc prosty mechanizm *IPC* w zasobach zawsze aktywnego procesu silnika **p_engine** zrealizowany przy pomocy struktury **struct ipc** zawierającej dwa pola typu int: *target* oraz *code*. Dzięki temu każdy proces może zarezerwować sobie dowolny kod *wywołania* na który będzie nasłuchiwać – jeżeli w polu *ipc.target* pojawi się ten kod to wtedy obsłuży żądany kod operacyjny tj. wartość z pola *ipc.code*, np. proces odpowiedzialny za odgrywanie cutscenek (**p_cutscene**) identyfikuje się jako *ipc.target = 1;* Natomiast kody operacyjne kolejno od 0 do 3 odpowiadają wybranym filmikom, które zostaną odegrane przez ten właśnie proces w chwili odpowiedniego ustawienia struktury ipc.

```
typedef struct engine_s
{
  ...

  struct ipc_s
  {
    int target;
    int code;
  } ipc;

  ...

} engine_t;
```

W oparciu o powyższą mechanikę, proces **p_game** identyfikujący się jako *ipc.target=666* będzie nasłuchiwał na dwa podstawowe kody: <50; 60) – ładowanie stanu gry oraz <60; 70) – zapis stanu gry. Zastosowaliśmy tutaj ciekawy zabieg, gdzie najmniej znacząca cyfra oznacza numer slotu zapisu (*save_no*). Oczywiście oznacza to, że silnik implementuje takową mechanikę. Od strony programistycznej, wypracowaliśmy następującą ideologię:

- Każdy obiekt który może być serializowany, zawiera metodę (proceduralną) **nazwaObiektu_save(***save_no***)** która *w jakiś sposób z*serializuje i zapisze do **swojego** pliku swój stan – stan tego obiektu
- Każdy obiekt który może być serializowany, w swoim konstruktorze, jeżeli silnik wymaga od niego odtworzenia swojego stanu (wczytania z wcześniej utworzonego zapisu) zawiera funkcjonalność pozwalającą to zrobić.

Przykładowy zapis stanu obiektu:

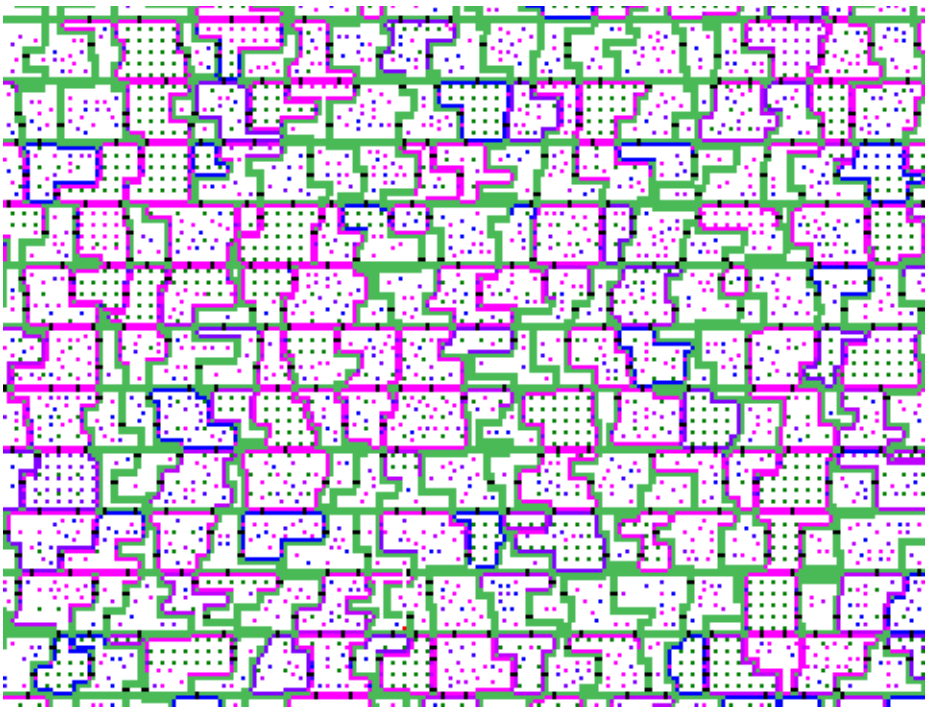
```
void player_save (game_t * game, int save_no)
{
  /*
    FILE FORMAT:
    0 float   player->x
    1 float   player->y
    2 float   player->health
  */
  char save_path [0xFF];
  sprintf(save_path, "./DATA/sav/player%d.sav", save_no);

  #ifdef _WIN32
    int fd = open(save_path, O_CREAT | O_WRONLY | O_BINARY, 0600);
  #else
    int fd = open(save_path, O_CREAT | O_WRONLY, 0600);
  #endif

  assert(write(fd, &game->player->x, sizeof(float)) == sizeof(float));
  assert(write(fd, &game->player->y, sizeof(float)) == sizeof(float));
  assert(write(fd, &game->player->health, sizeof(float)) == sizeof(float));

  close(fd);
}
```

Przykładowy odczyt stanu obiektu:

```
player_t * player_ctor (engine_t * engine)
{
  player_t * self = (player_t *) malloc(sizeof(struct player_s));

  ...

  if (engine->ipc.target == 666 && engine->ipc.code > 50  && engine->ipc.code < 60)
    player_load (self, engine->ipc.code % 10);

  return self;
}

void player_load (player_t * self, int save_no)
{
  char save_path [0xFF];
  sprintf(save_path, "./DATA/sav/player%d.sav", save_no);

  #ifdef _WIN32
    int fd = open(save_path, O_CREAT | O_RDONLY | O_BINARY, 0600);
  #else
    int fd = open(save_path, O_CREAT | O_RDONLY, 0600);
  #endif

  assert(read(fd, &self->x, sizeof(float)) == sizeof(float));
  assert(read(fd, &self->y, sizeof(float)) == sizeof(float));
  assert(read(fd, &self->health, sizeof(float)) == sizeof(float));

  close (fd);
}
```

Warto zauważyć, iż na systemach z rodziny MS Windows implementacja POSIX'owej funkcji **read()** jest raczej dziwna – tz. w przypadku nieużycia flagi **O_BINARY** (dostępnej tylko na tym systemie) psuje ona zapis danych binarnych do pliku. Stąd też zastosowaliśmy mechanizm *ifdef* tak aby zachować wieloplatformowość projektu.

Ostatnim aspektem wartym omówienia jest generator map (***c_gen.c***). Zastosowaliśmy go w celu urozmaicenia rozgrywki dla gracza. Ponieważ prototypowanie tak dynamicznych rzeczy w „czystym" języku C jest raczej „karkołomne", postanowiliśmy najpierw sam algorytm generowania map napisać w języku JS by móc w przeglądarce w sposób graficzny wizualizować generowane przez niego mapy.



Przykładowy fragment mapy wygenerowanej w JS

Proces generacji mapy rozpoczyna się od wypuszczenia szeregu spadających z góry na dół kwadratowych „fal” następnie fale te są przecinane poziomymi liniami rysowanymi w równych odstępach. Tak wygenerowany szkielet pokoi urozmaicany jest drzwiami generowanymi na razie na „falkach”. W kolejnym kroku generator usiłuje jednocześnie dodać drzwi do poziomych linii, pokolorować ściany oraz symetrycznie rozmieścić w kolorowanych pokojach lampy. W końcowej fazie generacji mapy dodawane są losowe propy rozmieszczone na całej mapie, obwódka ścian na krawędziach mapy oraz wystrzeliwany jest z pozycji startowej gracza promień wokół którego usuwane są ściany tak aby gracz miał „nie był uwięziony” na początku rozgrywki. Ostatecznie kod został przekonwertowany na C, finalnym funkcjom został dodany prefix ***js***_.

```
void js_map_generator (uint8_t map[511][511])
{
  js_horizontall_wall_randomizer(map);

  int doors[4096][2]; /* fails sometimes! */
  int doors_ptr = 0;
  int x, y, i, j;

  for (y = 0; y < 504; y += 16)
  {
    ... // Generowanie drzwi
  }

  for (i = 0; i < doors_ptr; i = i + 1)
  {
    ... // Kolorowanie ścian
  }


  for (i = 0; i < doors_ptr; i++)  // randomizacja tekstur drzwi
  {
    map[doors[i][0]][doors[i][1]] = get_random_int(5, 8, 1);
    map[doors[i][0]][doors[i][1] + 1] = get_random_int(5, 8, 1);
  }

  js_make_doors(map);
  js_gen_props(map);

  // let player out
  for (y = 255; y > 255 - 16; y--)
  {
   ... // Uwolnienie gracza w pokoju startowym
  }

  // secure far lands (generowanie obwoluty wokół mapy)
  for (i = 12; i < 511 - 12; i++)
  {
    map[i][12] = 9;
    map[i][511 - 12] = 9;
    map[12][i] = 9;
    map[511 - 12][i] = 9;
  }
}
```

5. Działania podjęte przez członków zespołu

Praca nad projektem wymagała od nas poświęcenia wielu godzin spędzonych przed ekranami komputerów. W tym czasie prowadziliśmy ożywione dyskusje dotyczące poszczególnych zadań, którym musieliśmy sprostać i strategii działania. Ten etap pracy możemy określić jako efektywną pracę zespołową. Chcąc wyodrębnić konkretne obowiązki członków zespołu, należy wymienić:
Marcin: renderer, menedżer procesów, scheduler, przeciwnik(Człowiek Korek), menu, mappoint’y
Maciek: renderer, menedżer procesów, scheduler, player, cutscenki, item
Szymek: menedżer procesów, IPC, scheduler, animacje tekstur

Mniej istotne aspekty silnika, które nie zostały wyszczególnione powyżej zostały zrealizowane wspólnie przez cały zespół projektowy.

Podczas prac nad projektem nie ucierpiał żaden student.