SPRAWOZDANIE
z realizacji projektu Programowanie w Języku C2

Skład zespołu: Maciej Bandura, Marcin Ślusarczyk, Szymon Śmiglarski; 2ID14B 2021/2022
Temat projektu: Gra horror - silnik typu raycaster.
Środowisko programistyczne (IDE): Visual Studio 2017/ Visual Studio 2019
Dodatkowe biblioteki graficzne: Allegro v5.2.7

1. Projekt

Gra została wykonana jako projekt z przedmiotu: Programowanie w Języku C2. Kod źródłowy napisany w języku C posługuje się biblioteką Allegro 5. Środowiskiem programistycznym użytym podczas prac nad projektem było MS Visual Studio 2017, do linkowania biblioteki Allegro 5 wykorzystano manager pakietów NuGet.

Dodatkowo kod źródłowy gry został skompilowany na platformie linux pod dystrybucjami zarówno Debian jak i Arch. Finalnie gra działała poprawnie na obu systemach - tj, Linux oraz Windows. Do kodu źródłowego został dołączony plik makefile, nazwa pliku wynikowego to ./HWL

2. Rozgrywka

Gra składa się z ośmiu unikalnych i przerażających map, mrożących krew w żyłach. Na każdej mapie grasuje straszliwy demon, zwany Jęczadłem (ang. Howler). Zadaniem gracza jest oczyszczenie wspomnianych lokacji z grasującego zła jakim jest wspomniany Howler. Do wypędzenia piekielnego bytu pomaga okultystyczne kadzidło. Po okadzeniu odpowiednio dużej przestrzeni duch sam zniknie a mapa zostaje uznana za zaliczoną. *Gwiazdki w prawym górnym rogu ekranu informują o stopniu oczyszczenia mapy.* Jęczadło za wszelką cenę będzie usiłowało przeszkodzić Ci w Twoim zadaniu. Rycząc od ucha do ucha, patrolując korytarze będzie nawiedzać gracza aż do jego śmierci. Kiedy tylko usłyszysz demoniczny skowyt - schowaj się! Módl się żeby Cię nie znalazł… A kiedy już Cię znajdzie.. Biegnij! Biegnij ile sił w nogach! Aby rozpocząć rozgrywkę należy uruchomić aplikację i za pomocą strzałek wybrać mapę na której gracz chce spróbować swoich sił.

3. Zastosowane techniki programistyczne

- Własny manager pamięci (dataset)
- Mechanizm opóźnienia wywołania funkcji (scheduler)
- Funkcja do przybliżania dystansu (err ~5%) o stałej złożoności czasowej
- Generator liczb pseudo-losowych
- Metoda renderowania obrazu pseudo 3D (raycasting)
- Matematyka z prawidłowego silnika 3D do określania pozycji propów na ekranie
- Prop Clipping - przycinanie propów za ścianami
- Programowanie obiektowe (do pewnego stopnia)
- Własny manager procesów (pseudo demon)
- Listy jednokierunkowe i dwukierunkowe
- Kolejki i słowniki

4. Budowa programu

Cały program zbudowany jest na podstawie autorskiego demona – tj. prostego managera procesów – omylnie nazwanego demonem (z ang. daemon) gdyż nie dostarcza on żadnych usług sieciowych. Przez procesy rozumiemy pewne zestawy funkcji, spakowane w struktury. Za pomocą tego typu danych reprezentujemy właśnie pojedyncze procesy. Po za zestawami funkcji, struktury zawierają w sobie pewne parametry kontrolne, takie jak: flaga statusu procesu, nazwę procesu jako ciąg znaków w stylu C, wskaźnik na lokacje pamięci dzieloną między funkcjami procesu oraz wskaźnik na następny proces – należy też wspomnieć że procesy przechowywane są za pomocą listy jednokierunkowej.

Nasze zadania działają w bardzo prosty sposób, przy wywoływaniu procesu – to jest, jego aktywacji, uruchamiana jest tzw. funkcja początkowa procesu. Funkcje te zazwyczaj służą do alokacji pamięci i podpinania pod jej wskaźnik pamięci wspólnej procesu. Na zadeklarowanym obszarze proces będzie później pracował. Analogicznie działają funkcje końcowe procesu. Wywoływane są one w chwili dezaktywacji tudzież wyłączania procesu. Służą one (zazwyczaj) do zwalniania wcześniej zaalokowanej pamięci w przestrzeni programu. Nieco inaczej działają funkcje aktualizujące. Są one wywoływane z każdym taktem głównej pętli programu, o ile dany proces jest aktywny. Pełnią one role tzw. myślicieli (ang. thinkers), ich zadania trudno ująć jednym zdaniem gdyż w zasadzie każdy proces pełni jaką swoją oddzielną rolę, a co za tym idzie, funkcje te też będą pełniły oddzielne role. Dobrym przykładem funkcji aktualizacyjnej będzie funkcja znajdująca się w porcesie renderera. W każdym takcie pętli rysuje ona na nowo wszystko na ekranie, (wszystko związane z grą – nie wolno mylić z menu czy też podsumowaniem, tam rysowanie na ekranie, jest zrealizowane inaczej). Główna pętla programowa, jest sztucznie opóźniana do 60 taktów na sekundę – funkcja aktualizacyjna renderera też jest wywoływana 60 razy na sekundę dając nam tzw. stały frame-rate (pl. ilość klatek na sekundę) równy 60 [fps]. Żeby dobrze zrozumieć funkcje procesowe, należałoby się im przyjrzeć z bliska, przykład takich funkcji podano poniżej:

```
void proc_init (daemon_t * daemon object, daemonprocess_t * this_process)
{
	// – funkcja początkowa, przyjmuje ona wskaźnik na obiekt demona,
	//   oraz wskaźnik na obiekt swojego procesu
}

void proc_update (daemon_t * daemon object, daemonprocess_t * this_process, void * self)
{
	// – funkcja aktualizacyjna, przyjmuje ona wskaźnik na obiekt demona,
	//   wskaźnik na obiekt swojego procesu oraz wskaźnik na swoją pamięc (dzieloną)
}

void proc_destroy (daemon_t * daemon object, daemonprocess_t * this_process, void * self)
{
	// – funkcja końcowa, przyjmuje takie same argumenty jak aktualizacyjna
}
```

Takowe funkcje będą odpowiednio wywoływane zgodnie z tym co zostało wcześniej opisane, oczywiście żeby demon w ogóle o nich wiedział trzeba je najpierw zarejestrować. Robi się to w bardzo prosty sposób, a mianowicie używając procedury daemon_register(), jeżeli nie potrzebujemy którejś z funkcji procesowych, to możemy ją po prostu pominąć, podając w miejscu jej oddelegowania wartość NULL. Poniżej znajduje się przykład rezerwacji powyższych funkcji procesowych dla procesu o nazwie Proc1Example:

```
daemon_register(*obiekt_demona, „Proc1Example”, proc_init, proc_update, proc_destroy);
```

Teraz aby aktywować funkcję wystarczy wywołać polecenie: daemon_proces_begin(*demon, „Proc1Example”).

Przechodząc do budowy silnika. W kodzie źródłowym silnika zarezerwowaliśmy sobie następujące procesy:

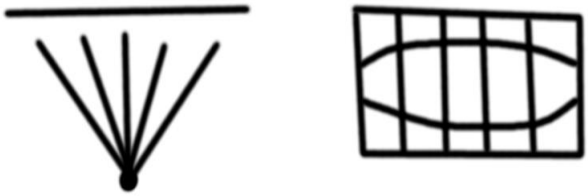
- *loader* – Ładowanie plików gry.
- *worker* – Rozgrywka.
- *render* – Rysowanie rozgrywki na ekranie.
- *summary* – Podsumowanie po zakończonej grze.
- *menu* – Menu gry.

Pierwszym aktywowanym procesem jest oczywiście proces Menu Gry („menu”), mechanika obsługi tego procesu znajduje się w pliku menu.c – plik ten jest bardzo prosty i przejrzysty. Z poziomu menu możemy uruchomić samą rozgrywkę. Tutaj zastosowanie autorskich procesów jest nieco bardziej skomplikowane. Na początek uruchamiany jest proces **loadera**, odpowiedzialny za ustawienie „klawiszologi” gry oraz załadowanie kontentu mapy poziomu gry (tekstury, schematu mapy, plików konfiguracyjnych itp..). Po załadowaniu plików, proces **loadera** aktywuje dwa kolejne procesy wykonujące się *współbieżnie** ***render*** oraz ***worker*.** Podczas gry proces *workera* dba o aktualizowanie stanu gry (ruchy gracza, progresja, animacje, dźwięki – tak naprawdę wszystko o czym gra **myśli**), proces rendera zajmuje się tylko i wyłącznie rysowaniem grafiki na ekranie – grafiki na podstawie tego co worker „wymyśli”. Dzięki temu, przykładowo jesteśmy w stanie zamrozić proces workera nie przerywając przy tym działania procesu rendera – da nam to możliwość stałego rysowania tego samego obrazu na ekranie w stałej ilości klatek na sekundę, bez aktualizowania stanu gry. Warto jeszcze wspomnieć, że proces workera jest tak zaprojektowany, że w chwili jego dezaktywacji, dodatków deaktywuje on proces render, dodatkowo niszcząc / zwalniając znajdujące się w pamięci pliki gry.

Ostatnim i bardzo istotnym elementem naszego projektu jest silnik typu raycaster znajdujący się w funkcji aktualizacyjnej procesu rendera. Jego zasada działania jest dość prosta lecz jej wytłumaczenie wymaga kilku zdań.

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. Promienie wystrzeliwujemy w jednej i tej samej osi, a mianowicie w osi Y. Ilość promieni jaką wystrzeliwujemy zależy od szerokości ekranu, natomiast kąt jaki określa interwał między tymi promieniami możemy policzyć, dzieląc pole widzenia przez szerokość ekranu. 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 * 256) 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.



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. 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 stanie się znikomy a kod będzie w dalszym ciągu dość szybki.

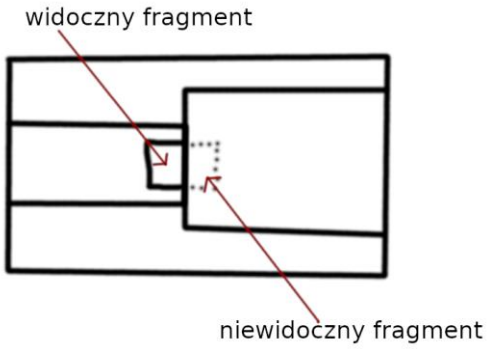
Czysto techniczną ciekawostką jest to, że znając już dokładną odległość i koordynat w jakich promień przecina się ze ścianą, możemy obliczyć w bardzo efektywny sposób (o ile mądrze użyliśmy proporcji do skalowania mapy) równoważną pozycję linii ściany w jej teksturze. Robimy to za pomocą poniższego kodu:

```
const int cx_off = (int)cx & 127;
const int cy_off = (int)cy & 127;
int tx_off = (cx_off == 0 || cx_off == 127)
      ? cy_off : cx_off;
```

Powyższy kod korzysta z faktu że logiczna funkcja and jakiejkolwiek liczby całkowitej z liczbą 127 da taki sam wynik jak funkcja modulu tej samej liczby całkowitej z liczbą 128. 128 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 tej że ściany, co też powyższa funkcja robi.

Kolejnym istotnym elementem naszego silnika grafiki jest cieniowanie. Odbywa się ono w bardzo prosty sposób. Znając odległość gracza od fragmentu ściany obliczamy współczynnik natężenia cienia w danym miejscu za pomocą funkcji *r_get_opacity().* Cień rysowany jest jako pionowa czarna linia z odpowiednim parametrem alfa, dzięki czemu im dalszy fragment ściany, tym ciemniejszy się on wydaje. Funkcja ta uwzględnia tzw. mapy świetlne. Cała mapa podzielona jest na fragmenty o różnym natężeniu światła (jest to istotne z punktu widzenia samej rozgrywki). W zależności od ilości natężenia światła w miejscu w którym gracz się znajduje (nie ściana) parametr/współczynnik cienia jest odpowiednio modyfikowany.

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:



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 tablicy *static float scan_lines[0xFFF]*. Ze względu na naturę tej tablicy, silnik nasz jest w stanie obsługiwać ekrany do rozdzielności nie szerszej niż 4K.

Pozostała funkcjonalność (scheduler, listener, sounds, overlay i inne obiekty) została opisana w dokumentacji wygenerowanej przy pomocy programu doxgen.

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: szkielet silnika, demon, renderer, scheduler, prop clipping, edytor map.
Maciek: nakładki, dźwięki, listener, zarządzanie pamięcią, podsumowanie gry.
Szymek: mapy, poziomy, mapy światła, obsługa drzwi.
Kolektywnie: cała reszta(testy, debugowanie, ogólna wizja projektu)

Podczas prac nad projektem nie ucierpiał żaden student.