AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji

Projekt i realizacja gry w przestrzeni nieeuklidesowej

Autor: Michał Flak (294309)

Opiekun: dr. inż. Witold Alda

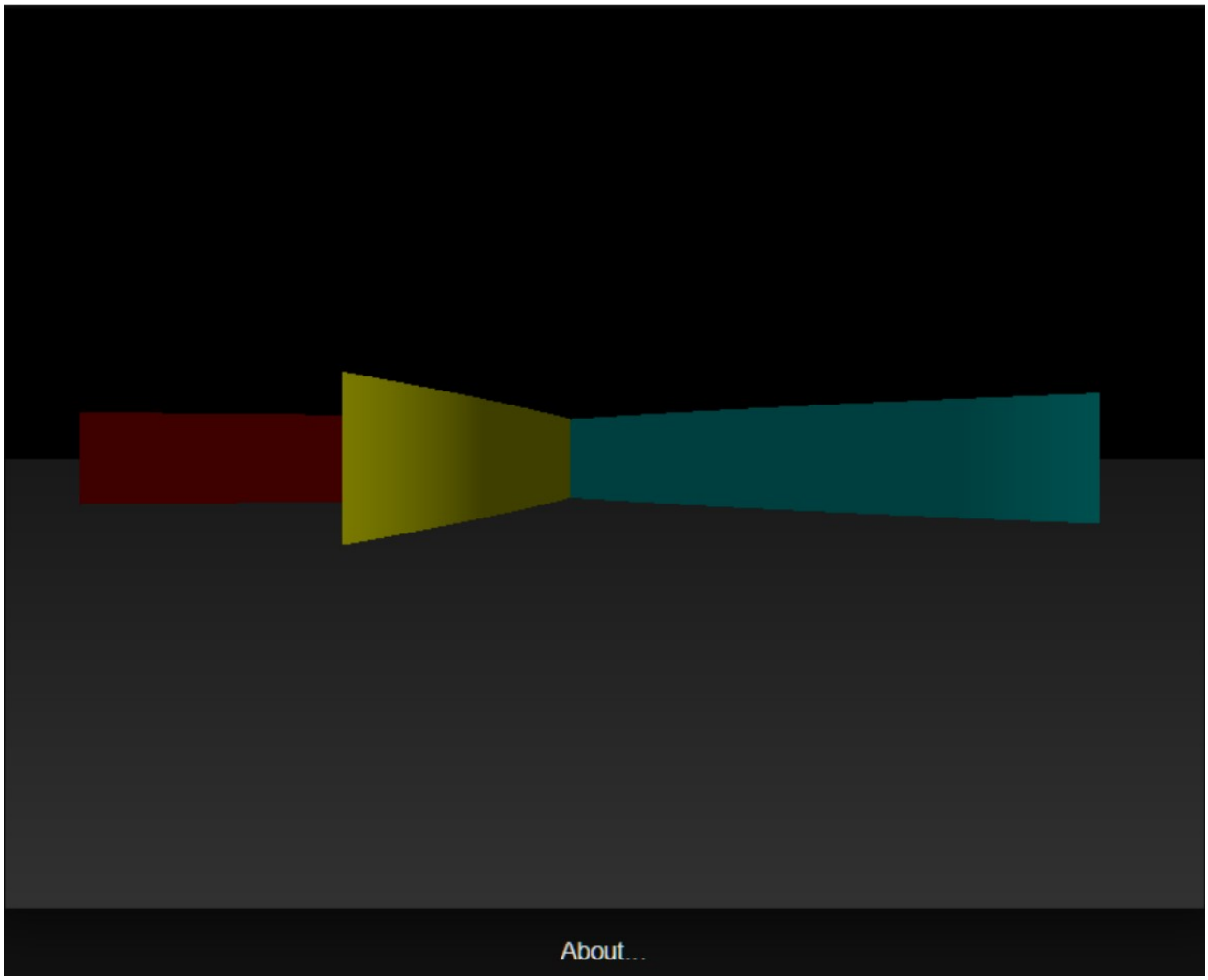
Dokumentacja deweloperska

Link do kodu źródłowego

https://github.com/elo-siema/HyperMaze

Studium wykonalności

W celu sprawdzenia wykonalności projektu wykonany został prototyp. Oparłem go na prostym raycasterze: https://github.com/hydrixos/raycaster-rust



Dostępny on jest do wypróbowania w przeglądarce: https://elo-siema.github.io/hyperbolic-raycaster-rust/index.html

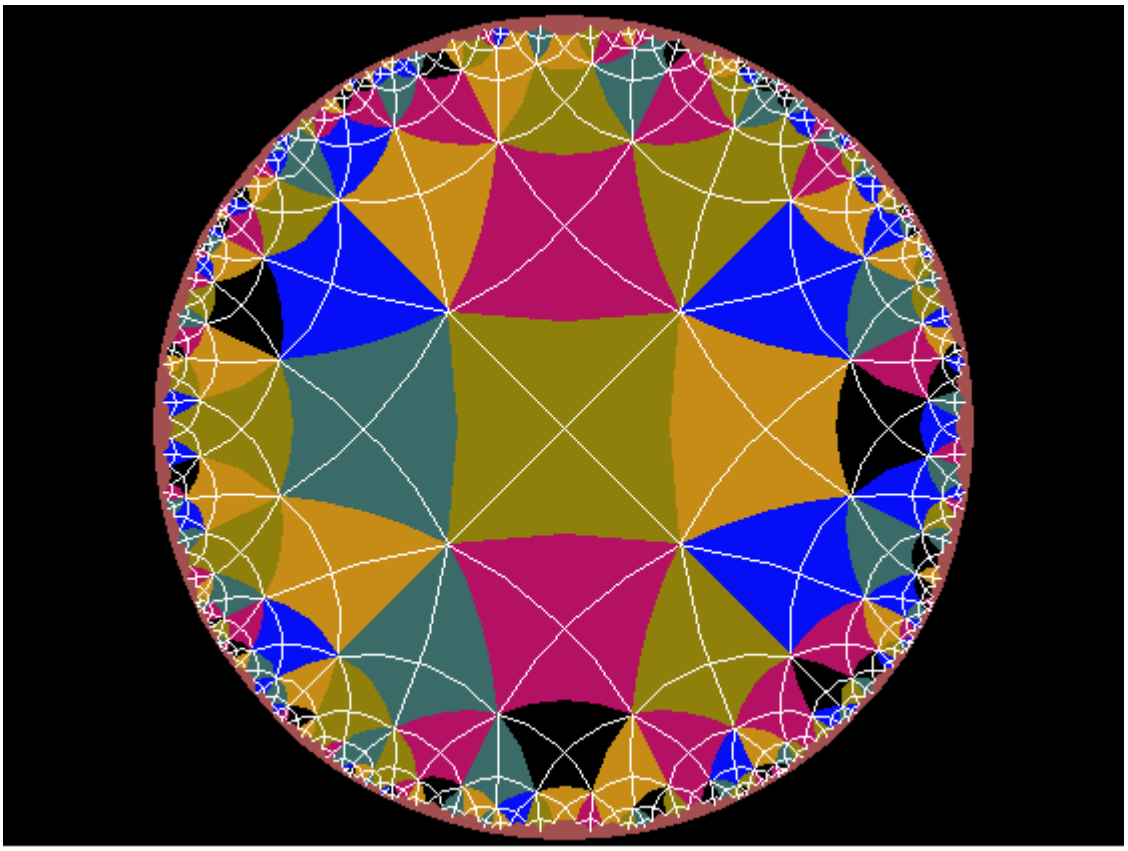
Kod źródłowy: https://github.com/elo-siema/hyperbolic-raycaster-rust

Napisany został w języku Rust, z wykorzystaniem bibloteki SDL2. Jest kompilowany do przeglądarki (WebAssembly / Emscripten) oraz do desktopowej aplikacji okienkowej.

Podczas tworzenia prototypu ujawniło się kilka problemów, do których znalazłem rozwiązania:

1. Wybór modelu przestrzeni / układu współrzędnych do łatwej edycji mapy

Początkowo rozważałem system oparty na siatce powstałej w wyniku tesselacji dysku Poincaré – numerując poszczególne kwadraty, ale szybko stało się jasne, że to rozwiązanie nie skaluje się oraz jest niezwykle niewygodne do projektowania poziomów:



Następnym pomysłem było stworzenie mapy jako zbioru ścian, z czego każda ma 2 punkty – początek I koniec. Punkty reprezentowane są we współrzędnych dysku Poincaré. Przykładowy poziom z dwiema ścianami:

```
[
	{
		"beginning": [0.0, 0.032],
		"end": [0.0, 0.31],
		"color": {
			"red": 255,
```

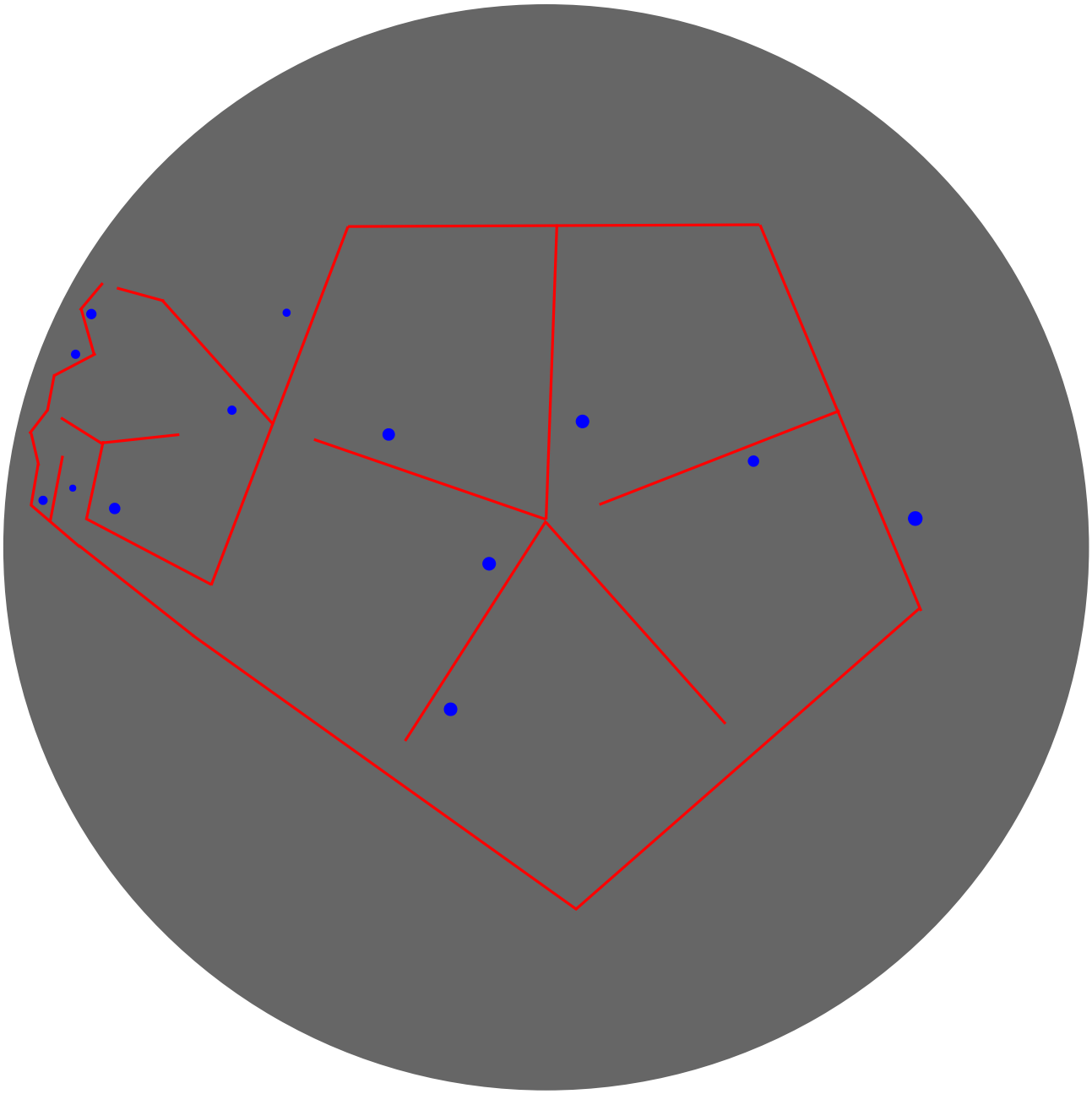
```
                "green": 0,
                "blue": 0
            }
        },
        {
            "beginning": [0.0, 0.032],
            "end": [0.270, 0.129],
            "color": {
                "red": 255,
                "green": 255,
                "blue": 0
            }
        }
]
```

Szybko okazało się jednak, że rozwiązanie to nie nada się do stworzenia większego poziomu niż trywialny – niemożliwe jest wpisanie dużej ilości tych punktów ręcznie.

Zdecydowałem się zmienić model przestrzeni na dysk Beltramiego-Kleina, ponieważ linie proste są w nim zachowane jako linie proste.

Zmieniłem tez sposób zapisu mapy, z pliku JSON na plik SVG - pozwoliło to na użycie graficznego edytora do edycji pozomu.

Rysunek: Plik SVG z poziomem



Gra oczekuje pliku SVG o wymiarach 2000x2000mm.

Gra iteruje po tagach <line>:

```
<line class="wall6" fill="none" id="svg_6" stroke="#ff0000" stroke-width="5" x1="635" x2="382.89449" y1="409.2105" y2="1069.73681"/>
```

tworząc na ich podstawie ściany. Tekstura ściany wybierana jest na podstawie atrybutu „class”.

Znajdźki (obiekty) tworzone są następująco:

Gra iteruje po tagach <ellipse>:

```
<ellipse cx="1066.99976" cy="768.25" fill="#0000ff" id="svg_13" rx="10" ry="10" stroke="#0000ff" stroke-width="5"/>
```

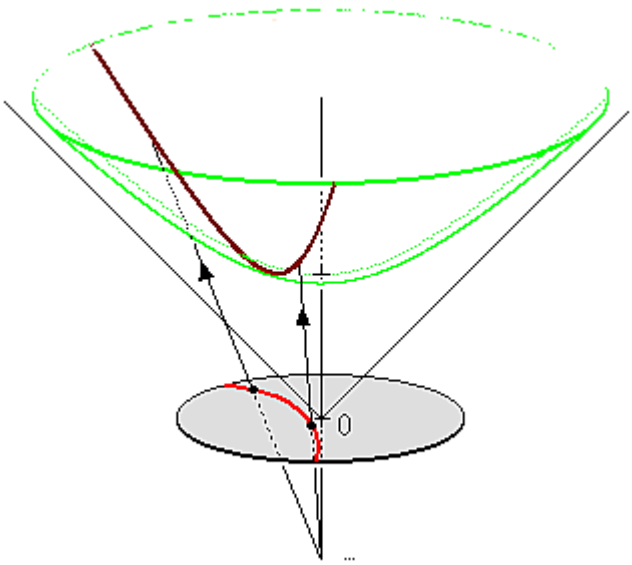
cx, cy oznaczają współrzędne znajdźki.

2. Wybór modelu przestrzeni do reprezentacji wewnętrznej świata gry

Tym razem priorytetem była łatwość dokonywania transformacji. Początkowo posiłkowałem się materiałami developerskimi z tworzenia gry Hyperbolica, gdzie reprezentacja świata trzymana jest w modelu dysku Poincare, tworzy to jednak spory problem z transformacjami – wymagają użycia żyrowektorów, które nie są jednak wspierane przez biblioteki oraz są zupełnie nieintuicyjne w użyciu.

Zdecydowałem się na reprezentację w postaci modelu hiperboloidu Minkowskiego, ze względu na analogiczne transformacje do przestrzeni euklidesowej – zmieniają się tylko macierze przekształcenia oraz funkcje trygonometryczne na ich hiperboliczne odmiany.

3. Rzutowanie świata na ekran



Tutaj zaszła największa zmiana względem prototypu. Pierwotnie użyta została technika raycastingu. Polega ona na mapowaniu wypuszczaniu promieni, od lewej do prawej, w ilości odpowiadającej ilości kolumn pikseli ekranu.
Promień wykrywałby pierwszą napotkaną ścianę, i rysowałby kolumnę pikseli o wysokości odwrotnie proporcjonalnej do odległości.
Zdecydowałem się zrezygnować z tej techniki z kilku powodów:

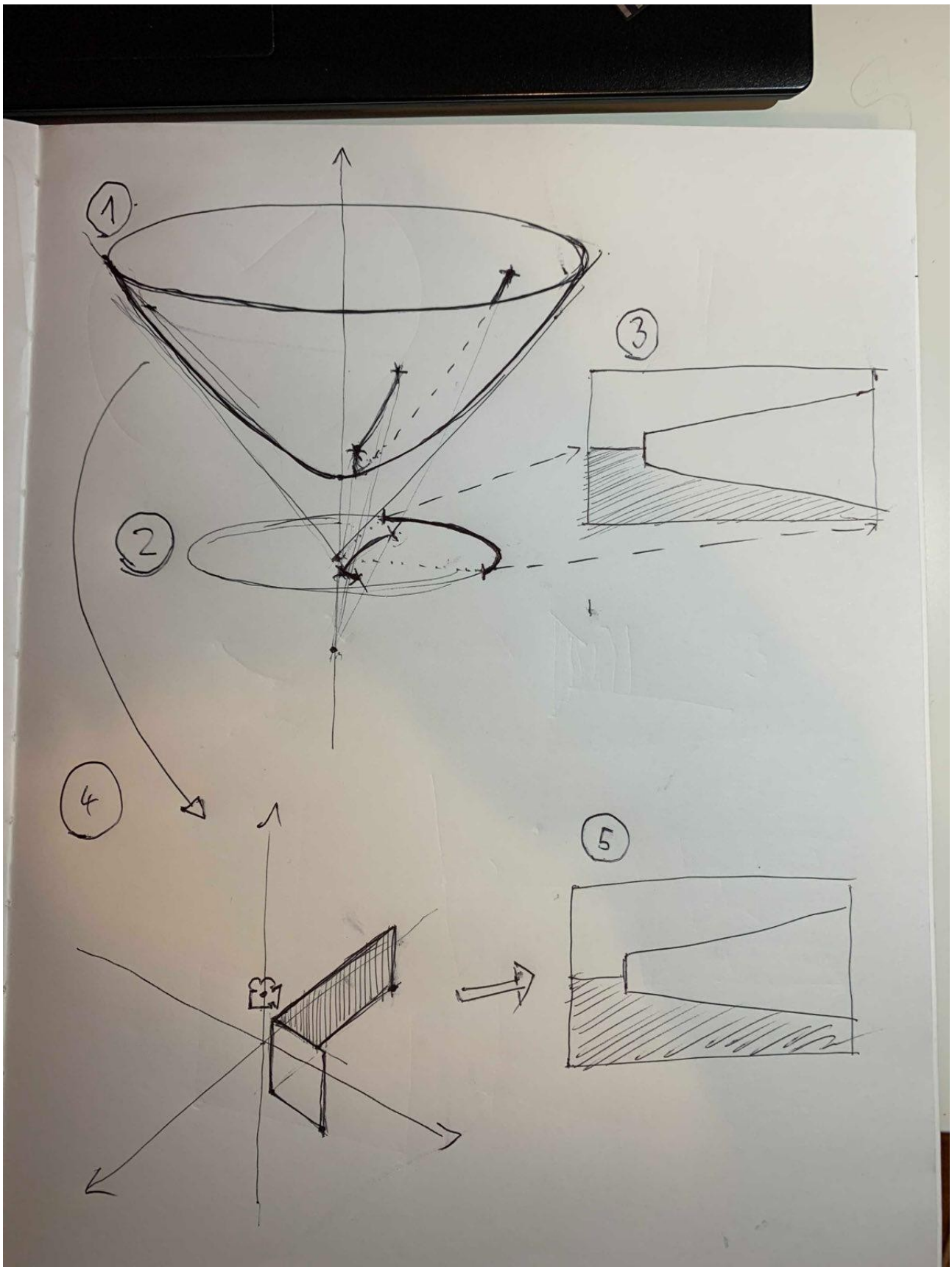
- Teksturowanie byłoby bardzo trudne – musiałbym pisać procedury do rysowania kolumna po kolumnie

- Byłby problem z rysowaniem jednocześnie ściany i znajdźki, gdzie znajdźka byłaby częściowo przezroczysta (nie zajmowałaby wysokości całej kolumny)
- Byłby problem z rysowaniem częściowo przysłoniętych znajdziek

Zamiast tego, zdecydowałem się na następujące rozwiązanie:

1. Projekcja wewnętrznej reprezentacji na dysk Kleina
2. Stworzenie sceny 3d w zwykłej przestrzeni euklidesowej, gdzie współrzędne punktów brałbym na podstawie współrzędnych polarnych na dysku Kleina, z zastrzeżeniem że jako odległości używałbym metryki Kleina
3. Renderowanie tej sceny zwykłym silnikiem 3d.

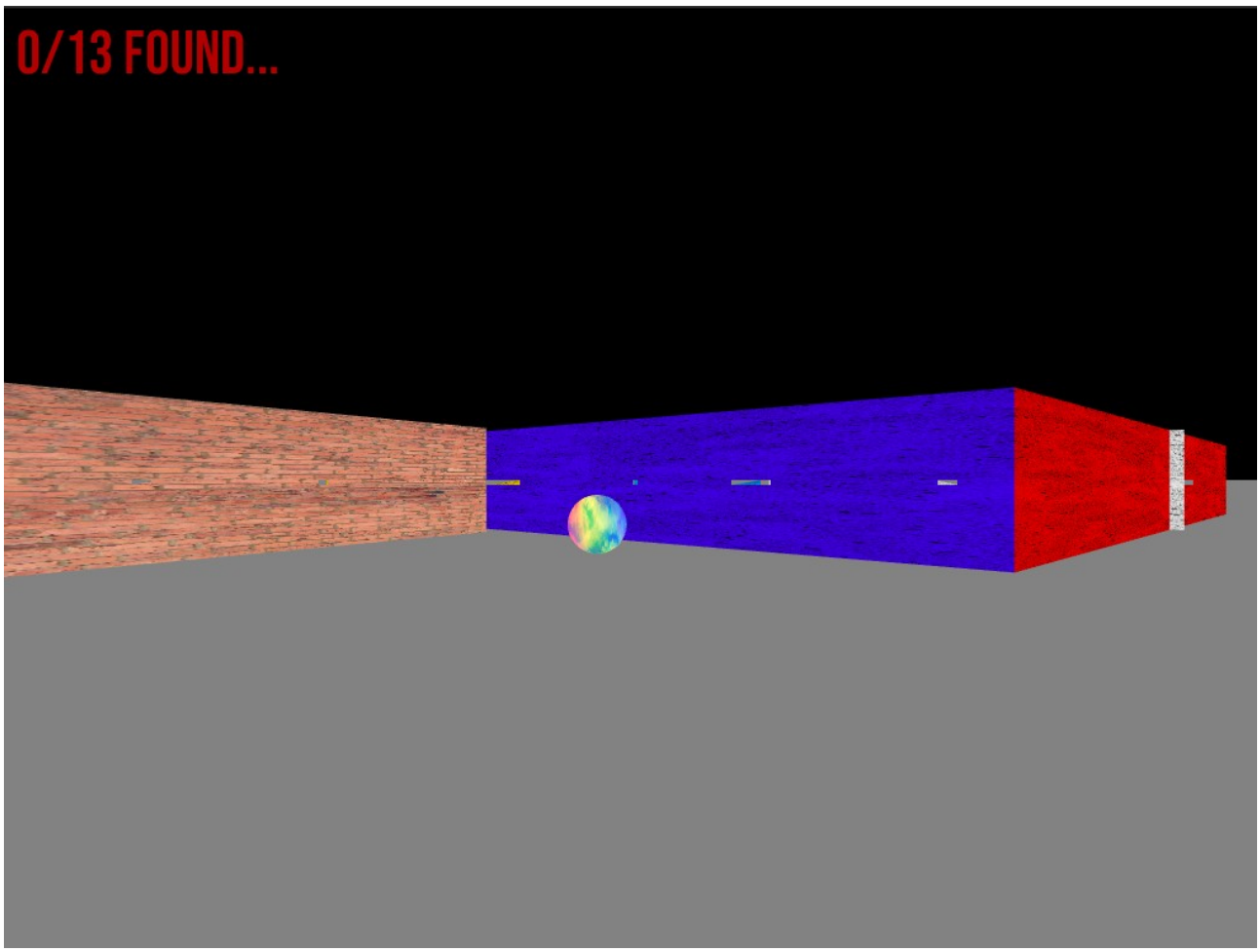
Wykonałem rysunek żeby zilustrować stare i nowe podejście:



Rysunek:

Stare podejście: Hiperboloid (1) → Dysk Poincarego (2) → Raycasting (3)

Nowe podejście: Hiperboloid (1) → Scena 3D euklidesowa (4) → Rendering 3D (5)



Screenshot: Nowa metoda renderingu

4. Wybór technologii

Zdecydowałem się w prototypie na język Rust oraz bibliotekę graficzną SDL2, ze względu na osobiste doświadczenie, szybkość, doskonały system typów, wsparcie społeczności oraz wsparcie platform na których chciałbym, żeby gra działała.

Wykonanie prototypu uświadomiło mi jednak, że powinienem zmienić silnik graficzny.

Zdecydowałem się na Macroquad, ze względu na prostotę użycia oraz wsparcie dla renderowania 3D w takim zakresie jakiego potrzebowałem.

Algorytmy i rozwiązane problemy

1. Detekcja kolizji

Trudnym problemem do rozwiązania była obsługa kolizji gracza ze ścianą.

Gracz znajduje się zawsze w środku układu współrzędnych. Wykonanie ruchu transformuje świat, przesuwając go w jego kierunku. Musimy jednak wykryć, kiedy transformacja zakończyła się przejściem przez ścianę, i odpowiednio ją skorygować.

Algorytm który skonstruowałem działa następująco:

1. Bierzemy ścianę
2. Obliczamy przybliżony punkt styczny prostej, na której leży ściana, z okręgiem ze środkiem w punkcie 0,0
3. Sprawdzamy, czy punkt styczny mieści się między początkiem i końcem ściany na tej prostej
4. Jeśli tak, sprawdzamy czy promień okręgu jest mniejszy niż stała COLLISION_RADIUS
5. Jeśli tak, konstruujemy wektor normalny do ściany, zwrócony w stronę gracza (punktu 0,0)
6. Nadajemy mu długość równą odległości – COLLISION_RADIUS
7. Dodajemy go do kolekcji wektorów „corrections”
8. Powtarzamy powyższe dla reszty ścian

Musimy również zadbać o przypadki podejścia do ściany od boku. W tym celu postępujemy podobnie jak powyżej, z tym że badamy punkt styczny okręgu ze środkiem 0,0, z okręgiem ze środkiem w wierzchołku ściany.

Następnie dodajemy uzyskane wektory i przesuwamy świat o uzyskany wektor.

Opis użytych technologii

1. Język programowania: Rust

Język stworzony przez Mozillę w 2010 roku jako odpowiedź na największe bolączki C i C++, szczególnie w zakresie bezpieczeństwa pamięci oraz zarządzania zależnościami.

Gwarantuje bezpieczeństwo pamięci bez użycia garbage collectora przez tzw. borrow checker – mechanizm, który śledzi to, który fragment kodu jest w posiadaniu którego zakresu pamięci. Nie pozwala na przykład, żeby istniały dwie referencje umożliwiające zapis do tego samego miejsca w pamięci naraz.

Użyłem go w tym projekcie ze względu na szybkość, nowoczesny menadżer pakietów i narzędzie do budowania (Cargo), oraz osobiste zainteresowanie.

2. Silnik graficzny: Macroquad

https://github.com/not-fl3/macroquad

Pozwala na konstruowanie prostych scen 3d z wielokątów, obsługuje teksturowanie.

Pozwala wyświetlać tekst na ekranie.

Obsługuje również input z klawiatury.

3. Biblioteka do algebry liniowej – Nalgebra

https://nalgebra.org/

Używam jej do wszystkich operacji matematycznych na wektorach i macierzach.

Na podstawie Nalgebrowego Vector2 / Vector3 zbudowałem wrappery, odpowiadające konkretnym modelom przestrzeni hiperbolicznej (Kleinpoint, Hyperpoint, Poincarepoint). Zrobiłem to w celu rozróżnienia semantycznego, zachowując jednak możliwość korzystania z funkcji Nalgebry na tych typach.

4. Parser svg – svg

https://crates.io/crates/svg

Używam go do wczytania mapy z pliku svg.

5. Manager pakietów – Cargo

Rustowy manager pakietów.

Definiujemy zależności pod „dependencies” w pliku cargo.toml:

```
[package]
name = "hypermaze"
version = "0.1.0"
edition = "2018"

# See more keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
macroquad = "*"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
nalgebra = {version = "*", features = ["serde-serialize"]}
futures = {version = "0.3", features = ["thread-pool"]}
svg = "0.10.0"

[profile.dev]
opt-level = 0
```