

WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI
POLITECHNIKA WROCŁAWSKA

GRAFIKA NIEEUKLIDESOWA NA PRZESTRZENI DWUWYMIAROWEJ

MACIEJ HAJDUK

NR INDEKSU: 236596

Praca inżynierska napisana
pod kierunkiem
Prof. dr hab. Jacka Cichonia



Politechnika
Wrocławska

WROCŁAW 2019

Wydział Podstawowych Problemów Techniki

Maciej Hajduk

Politechnika Wrocławska

Wrocław 2019

Spis treści

1	Wstęp	4
1.1	Kontekst historyczny	4
1.2	Wybrane zagadnienie	5
2	Analiza problemu	7
2.1	Podstawowy podział	7
2.1.1	Geometria Łobaczewskiego-Bolyaia (hiperboliczna)	7
2.1.2	Geometria Riemanna (eliptyczna)	7
2.1.3	Różnice pomiędzy geometriami	8
2.2	Popularne modele geometrii hiperbolicznej	8
2.2.1	Model Kleina	8
2.2.2	Model półpłaszczyzny Poincaré	9
2.2.3	Model dysku Poincaré	10
2.2.4	Model Hemisfery	11
2.3	Uzasadnienie wyboru modelu dysku Poincaré	11
3	Projekt systemu	14
3.1	Cykl pracy silnika	14
3.2	Typy obiektów renderowanych przez silnik	15
3.3	Obiekty geometrii Euklidesowej	17
3.3.1	Klasa Point	17
3.3.2	Klasa Line	17
3.3.3	Klasa Circle	18
3.3.4	Klasa Plane	18
3.4	Obiekty geometrii hiperbolicznej	18
3.4.1	Klasa HypLine	18
3.4.2	Klasa HypPoint	19
3.4.3	Klasa HypPolygon	19
3.4.4	Klasa HypTile	20
3.5	Funkcje dodatkowe	20
3.6	Transformacja Möbiusa	20
4	Implementacja systemu	22
4.1	Opis technologii	22
4.2	Poszczególne składowe systemu	22
4.3	Konfiguracja systemu	22
4.3.1	Biblioteki projektu	22
4.3.2	Bundlowanie aplikacji	22
4.3.3	Konfiguracja języka	22

4.4	Pliki źródłowe silnika	22
4.4.1	Polygon Demo	24
4.4.2	Interaction Demo	24
4.4.3	Tesselation Demo	25
4.5	Pliki źródłowe pracy	25
5	Instalacja i wdrożenie	27
5.1	Serwer deweloperski	27
5.2	Wdrożenie na serwerze WWW	28
6	Podsumowanie	30
7	Bibliografia	32
8	Zawartość płyty CD	34

1 Wstęp

1.1 Kontekst historyczny

Geometria jest nauką o mierze. Nazwa ta narzuca silne skojarzenia z nauką niemalże przyrodniczą. Nauczana we wszystkich szkołach od dwóch i pół tysiąca lat - wydawałoby się jest już czymś bardzo dobrze poznanym. Nowe teorie matematyczne doprowadziły jednak do podważenia tej pewności i powstania geometrii alternatywnych.

O życiu Euklidesa wiemy bardzo niewiele, a przecież to jemu zawdzięczamy nazwę *naszej* geometrii. Ani data urodzenia, ani pochodzenie nie są nam znane, a wszystkie informacje o nim czerpiemy z antycznych dzieł w których opisana jest matematyka. Około 300 roku przed naszą erą, Euklides - dyrektor Biblioteki Aleksandryjskiej, wydał swoje największe dzieło - *Elementy Geometrii*, na które składa się 13 ksiąg zawierających właściwie całą wiedzę matematyczną tamtych czasów. Początkowe definicje pierwszej księgi posiadają 5 stwierdzeń, które według Euklidesa są tak proste, że nie wymagają uzasadnienia. Euklides nazwał je aksjomatami:

1. Od dowolnego punktu do dowolnego innego można poprowadzić prostą.
2. Ograniczoną prostą można dowolnie przedłużyć.
3. Z dowolnego środka dowolnym promieniem można opisać okrąg.
4. Wszystkie kąty proste są równe (przystające).
5. Jeśli 2 proste na płaszczyźnie tworzą z trzecią kąty jednostronne wewnętrzne o sumie mniejszej od 2 kątów prostych, to proste te, po przedłużeniu, przetną się i to z tej właśnie strony. ¹

Piąty aksjomat mówi o tym, że z jednej strony przecinanej linii dwie proste będą się przybliżać. Zaczął on dość szybko wzbudzać podejrzenia. Jest znacznie bardziej skomplikowany od pozostałych, a już na pewno nie tak intuicyjny. Nawet Euklides unikał używania go w swoim dziele tak długo, jak to było możliwe i użył go dopiero w dowodzie własności 29.

Można śmiało powiedzieć, że piąty aksjomat w kolejnych wiekach spędzał uczonym sen z powiek. Przez kolejne 1500 lat matematycy próbowali udowodnić, że o wiele bardziej skomplikowany postulat musi wynikać z pozostałych czterech. Jednym z pierwszych zajmujących się tym problemem uczonych, był żyjący w V wieku naszej ery Proklos. Stwierdził on w swoim komentarzu do dzieł Euklidesa:

Nie jest możliwe, aby uczony tej miary co Euklides godził się na obecność tak długiego postulatów w aksjomatyce – obecność postulatów wzięła się z pośpiesznego kończenia przez niego Elementów, tak aby zdążyć przed nadejściem słusznie oczekiwanej rychłej śmierci; my zatem – czcząc jego pamięć – powinniśmy ten postulat usunąć lub co najmniej znacznie uprościć. ²

Wyzwanie usunięcia piątego aksjomatu podjęło wielu matematyków w kolejnych wiekach. Prowadziło to do postania wielu nowych twierdzeń, które w istocie były piątemu aksjomatowi równoważne. Prowadziło to do sprzeciwu innych uczonych. W szczególności Immanuel Kant w swoim dziele *Krytyka czystego rozumu* stwierdził, że intuicja geometryczna jest wrodzona, więc nie może istnieć wiele równoległych geometrii, a każdy kto chciałby zajmować się alternatywnymi geometriami nie nadaje się do nauki. Nie wszyscy zgodzili się z tym stwierdzeniem. Udano się do największego w tamtym czasie autorytetu - Carla Friedricha Gaussa, który jednak wycofał się, bojąc się - jak pisał - wrzasku Beotów. Do problemu należało się jednak odnieść. Odważyło się na to dwóch młodych ludzi, którzy uparli się nie tylko na uprawianie tej geometrii, ale wręcz głosili jej równoprawność. Rosjanin, Nikołaj Łobaczewski oraz Węgier - Janos Bolyai, niezależnie od siebie opublikowali prace w których - chociaż odmiennie - nowa geometria była konsekwentnie wyprowadzona. Obu odkrywców spotkała też za to kara, Łobaczewski został wręcz zmuszony do opuszczenia katedry.

¹Geometria euklidesowa. Encyklopedia PWN

²Najgłupiej postawiony problem matematyki. Marek Kordos - Delta, maj 2012

Sprawę nowej geometrii (nazywanej już geometrią Bolyaia-Łobaczewskiego) przejął Felix Klein. Postawił on tezę, że jeżeli za pomocą geometrii euklidesowej jesteśmy w stanie przedstawić tę nieeuklidesową - i odwrotnie, to oba modele są sobie w istocie równoważne. Opublikował też w 1870 roku dzieło, w którym dowiódł równoprawności obu modeli.

Dosadnie do nowego modelu odniósł się fizyk - Hermann Helmholtz, publikując pracę, w której określił matematykę jako skrzynkę z narzędziami dla nauk przyrodniczych, czym odebrał jej walor nauki przyrodniczej jako takiej.

1.2 Wybrane zagadnienie

Celem niniejszej pracy jest implementacja prostego silnika graficznego skupiający się na renderowaniu wizualizacji płaszczyzny dysku w modelu Poincarégo geometrii hiperbolicznej.

Praca swoim zakresem obejmuje obsługę rysowania linii, okręgów, wielokątów na tejże płaszczyźnie oraz implementacje przykładowych programów obejmujących wizualizację bardziej skomplikowanych struktur. Na tle innych implementacji, aplikacja wyróżnia się dostarczaniem możliwości i realizacją problemu z pomocą matematycznego opisu pewnego modelu. Przykładowe demonstracje możliwości aplikacji są dostarczone razem z kodem źródłowym, jest to, poza możliwością narysowania dowolnego wielokąta, rysowaniem figur foremnych czy prostych animacji, także interakcja z urządzeniami peryferyjnymi i tesselacja przestrzeni hiperbolicznej. Niewątpliwą zaletą dostarczonej aplikacji jest prostota implementacji własnych rozwiązań, na co składa się silne typowanie języka Typescript wraz z dokładnymi interfejsami dla klas oraz funkcje dostarczone przez silnik, pozwalające na łatwe manipulowanie wyświetlającymi się obiektami, nie wymagające przy tym zrozumienia modelu.

Praca składa się z czterech rozdziałów:

Rozdział pierwszy: Omówienie analizy wybranego problemu, przedstawienie motywacji podjęcia tego tematu oraz uzasadnienie wyboru modelu płaszczyzny Poincaré. Rozdział zawiera poza tym komentarz do różnych rodzajów geometrii nieeuklidesowych, oraz krótki opis i porównanie innych modeli geometrii hiperbolicznej.

Rozdział drugi: Szczegółowa charakterystyka systemu wraz z opisem poszczególnych plików oraz przeznaczeniem klas i funkcji składających się na program. Opisanie algorytmów przekształcających byty w geometrii Euklidesowej na odpowiadające im elementy geometrii hiperbolicznej, funkcji pomocniczych, reprezentacji punktów i linii w obu modelach.

Rozdział trzeci: Opis technologii użytych do implementacji projektu: wybranego języka programowania, środowiska składającego się na aplikację oraz bibliotek wykorzystanych w programie.

Rozdział czwarty: Instrukcje instalacji i wdrożenia systemu w środowisku docelowym. Końcowy rozdział stanowi podsumowanie uzyskanych wyników i ewentualne możliwości rozwoju projektu.

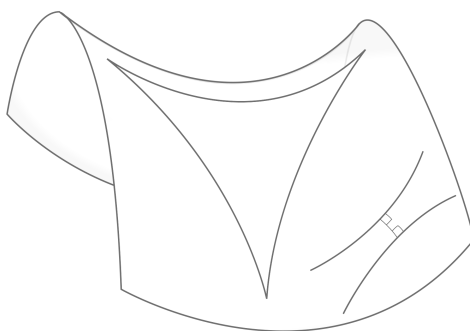
2 Analiza problemu

W niniejszym rozdziale przedstawiona będzie analiza problemu, opis matematyczny modelu płaszczyzny dysku Poincaré oraz przegląd kilku wybranych modeli geometrii nieeuklidesowej.

Odkrycie, że piątego aksjomatu nie można udowodnić na podstawie pozostałych czterech aksjomatów, było dla naukowców niespodzianką. Zrobiono to, demonstrując istnienie geometrii, w której pierwsze cztery aksjomaty utrzymywały się, ale piąty nie. Debata nad piątym postulatem Euklidesa stworzyła problem, jak powinna wyglądać alternatywna geometria. Umiano pokazać zaledwie poszczególne właściwości takich geometrii. Pierwszy model geometrii nieeuklidesowej został stworzony przez Kleina. W sprawę zaangażowało się wielu matematyków, w tym również Bernard Rieman. Stwierdził on, że można opisać nieskończenie wiele struktur matematycznych, które nie będą spełniały postulatów Euklidesa, będąc dalej geometriami.

2.1 Podstawowy podział

Geometria nieeuklidesowa to każda geometria, która nie spełnia przynajmniej jednego z postulatów Euklidesa. Geometrie nieeuklidesowe możemy podzielić na dwa rodzaje:



Rysunek 1: Trójkąt oraz dwie proste przedstawione na powierzchni o geometrii hiperbolicznej

2.1.1 Geometria Łobaczewskiego-Bolyaia (hiperboliczna)

Geometria hiperboliczna jest bliżej związana z geometrią euklidesową, niż się wydaje. Jedyną różnicą aksjomatyczną jest postulat równoległy. Po usunięciu postulatu równoległego z geometrii euklidesowej geometria wynikowa jest geometrią absolutną. Wszystkie twierdzenia o geometrii absolutnej, w tym pierwsze 28 twierdzeń zaprezentowanych przez Euklidesa, obowiązują w geometrii zarówno euklidesowej jak i hiperbolicznej.

W modelu hiperbolicznym, w płaszczyźnie dwuwymiarowej, dla dowolnej linii L i punktu X , który nie jest na L , istnieje nieskończenie wiele linii przechodzących przez X , które nie przecinają L .

2.1.2 Geometria Riemanna (eliptyczna)

Geometria eliptyczna jest geometrią nieeuklidesową o dodatniej krzywiznie, która zastępuje postulat równoległy stwierdzeniem “przez dowolny punkt na płaszczyźnie, nie ma linii równoległych do danej linii”. Geometria eliptyczna jest czasem nazywana również geometrią Riemannowską. Model można wizualizować jako powierzchnię kuli, na której linie przyjmowane są jako wielkie koła. W geometrii eliptycznej suma kątów trójkąta wynosi więcej niż 180 stopni.

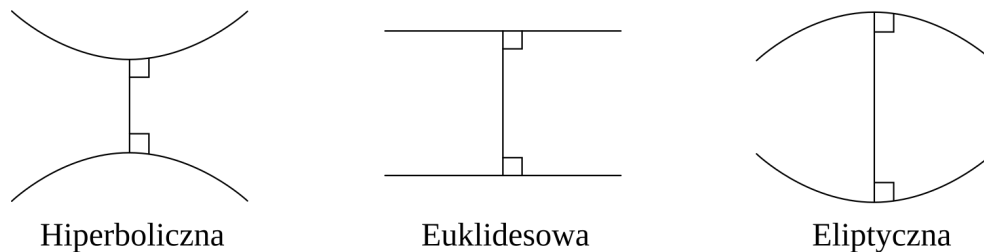
W modelu eliptycznym dla dowolnej linii L i punktu X , który nie jest na L , wszystkie linie przechodzące przez X przecinają L .

2.1.3 Różnice pomiędzy geometriami

Sposobem opisania różnic między tymi geometriami jest rozważenie dwóch linii prostych rozciągniętych w nieskończoność w płaszczyźnie dwuwymiarowej, które są prostopadłe do trzeciej linii:

- W geometrii euklidesowej linie pozostają w stałej odległości od siebie (co oznacza, że linia narysowana prostopadłe do jednej linii w dowolnym punkcie przecina drugą linię, a długość odcinka linii łączącego punkty przecięcia pozostaje stała) i są znane jako równoległe.
- W geometrii hiperbolicznej linie *zakrzywiają się* od siebie, zwiększając odległość w miarę przesuwania się dalej od punktów przecięcia ze wspólną prostopadłą; linie te są często nazywane ultraparallelami.
- W geometrii eliptycznej linie *zakrzywiają się* do siebie i w końcu przecinają.

Omawiane różnice pokazane są na poniższym rysunku.



Rysunek 2: Zachowanie linii ze wspólną prostopadłą w każdym z trzech rodzajów geometrii

Niniejsza praca skupia się na geometrii hiperbolicznej.

Istnieje kilka możliwych sposobów wykorzystania części przestrzeni euklidesowej jako modelu płaszczyzny hiperbolicznej. Wszystkie te modele spełniają ten sam zestaw aksjomatów i wyrażają tę samą abstrakcyjną płaszczyznę hiperboliczną. Dlatego wybór modelu nie ma znaczenia dla twierdzeń czysto hiperbolicznych, jednak różnica występuje podczas ich wizualizacji.

Następne podrozdziały są poświęcone krótkiemu omówieniu najpopularniejszych z nich.

2.2 Popularne modele geometrii hiperbolicznej

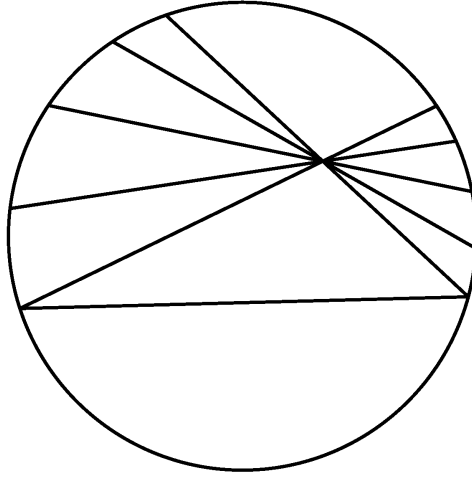
Geometria hiperboliczna została opisana za pomocą wielu modeli. Poniżej zaprezentowano cztery popularne modele.

2.2.1 Model Kleina

Model Kleina - a w zasadzie model dysku Beltrami-Kleina jest modelem geometrii hiperbolicznej, w którym punkty są reprezentowane przez punkty we wnętrzu dysku. Przyjmuje on następujące założenia:

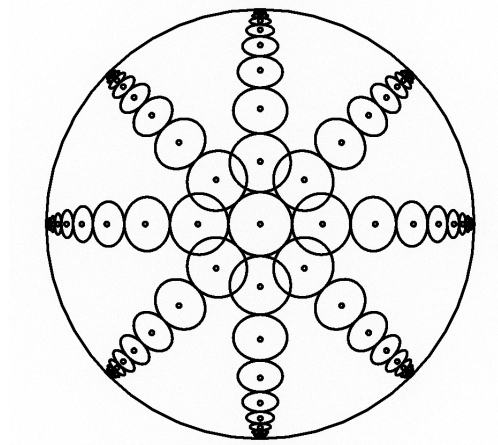
- **Płaszczyzną hiperboliczną** jest wnętrze koła bez krawędzi.
- **Prostymi hiperbolicznymi** są cięciwy tego koła (końce prostej).

- **Proste będą prostopadłe** wtedy, gdy przedłużenie jednej z nich przechodzi przez punkt przecięcia stycznych do obu linii.



Rysunek 3: Przykład kilku linii na modelu Kleina

Linie w modelu pozostają proste, a cały model można łatwo osadzić w ramach rzeczywistej geometrii rzutowej. Model ten nie jest jednak zgodny, co oznacza, że kąty są zniekształcone, a okręgi na płaszczyźnie hiperbolicznej na ogół nie są okrągłe w modelu.



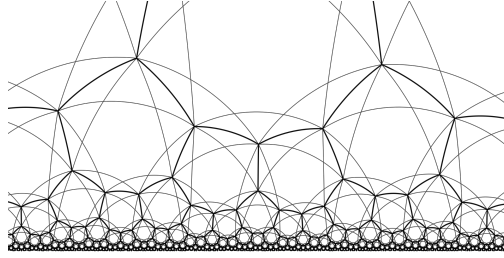
Rysunek 4: Graficzne zobrazowanie kół w modelu Kleina

2.2.2 Model półpłaszczyzny Poincaré

Model półpłaszczyzny Poincaré to model dwuwymiarowej geometrii hiperbolicznej, jest to płaszczyzna:

$$\{(x, y) \mid y > 0; x, y \in \mathbb{R}\}$$

Model nosi imię Henri Poincaré, ale został stworzony przez Eugenio Beltrami, który użył go wraz z modelem Kleina i modelem dysku Poincaré, aby pokazać, że geometria hiperboliczna jest równie spójna, jak spójna jest geometria euklidesowa. Omawiany model jest zgodny, co oznacza, że kąty zmierzone w punkcie modelu są równe kątom na płaszczyźnie hiperbolicznej.



Rysunek 5: Przykład tesselacji w modelu półpłaszczyzny Poincaré

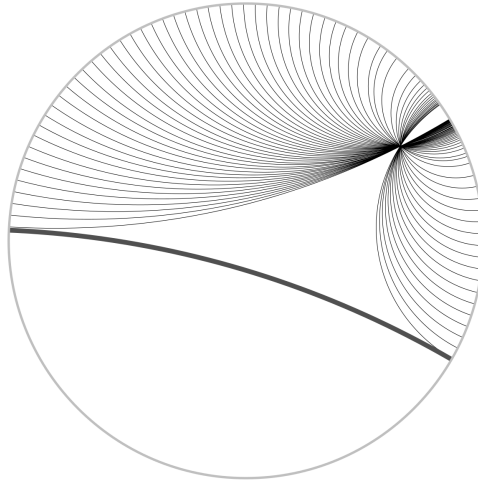
2.2.3 Model dysku Poincaré

Model dysku Poincaré wykorzystuje wnętrze dysku jako model płaszczyzny hiperbolicznej. W poniższych przykładach omawiany będzie dysk jednostkowy, który będzie również przedmiotem dalszych rozważań.

- **Punkty hiperboliczne** to punkty wewnątrz dysku jednostkowego.
- **Linie hiperboliczne** to łuki koła prostopadłe do dysku. Linie hiperboliczne przechodzące przez początek degenerują się do średnic.
- **Kąty** są mierzone jako kąt euklidesowy między stycznymi w punkcie przecięcia.
- **Odległości** między punktami hiperbolicznymi można mierzyć w oparciu o normę euklidesową:

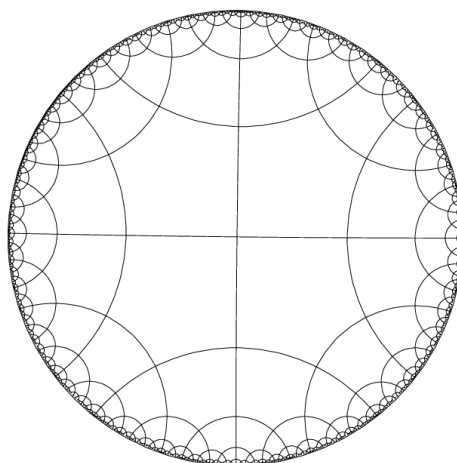
$$\delta(u, v) = 2 \frac{\|u - v\|^2}{(1 - \|u\|^2)(1 - \|v\|^2)}$$

Ponieważ rozpatrywany jest dysk jednostkowy, powyższy wzór nie zawiera w zmiennej dla promienia.



Rysunek 6: Przykład kilkunastu linii w modelu dysku Poincaré

Model jest zgodny, to znaczy, że zachowuje kąty. Oznacza to, że kąty hiperboliczne między krzywymi są równe kątom euklidesowym w punkcie przecięcia. Wadą jest, że linie hiperboliczne są modelowane poprzez łuki koła euklidesowego, przez co wydają się zakrzywione.

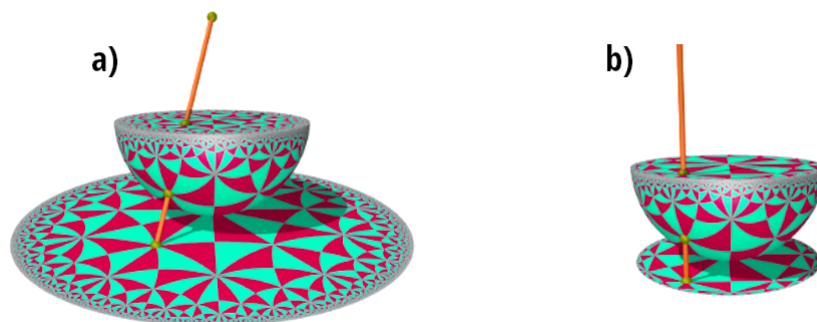


Rysunek 7: Przykład tesselacji w modelu dysku Poincaré

2.2.4 Model Hemisfery

Hemisfera nie jest często używana jako model płaszczyzny hiperbolicznej. Jest jednak bardzo przydatna w łączeniu różnych innych modeli za pomocą różnych rzutów, jak pokazano na poniższym rysunku.

- **Punkty hiperboliczne** to punkty na półkuli południowej.
- **Linie hiperboliczne** to półkola powstałe z przecięcia półkuli południowej z płaszczyznami prostopadłymi do równika.



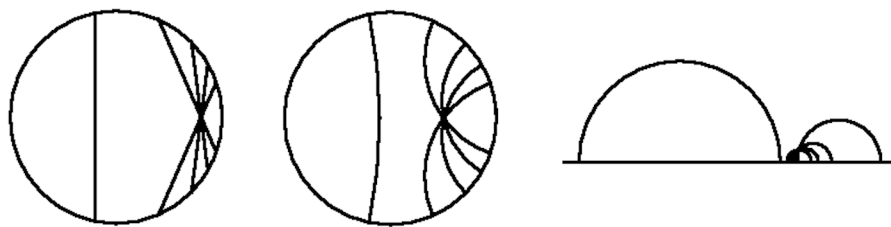
Rysunek 8: Rzut na dysk Poincaré (a) i projekcja do modelu Klein-Beltrami (b)

Wadą omawianego rozwiązania, jest dodatkowy wymiar, jaki należy rozpatrzyć podczas implementacji, co komplikuje pracę z tym modelem.

2.3 Uzasadnienie wyboru modelu dysku Poincaré

Jak zaznaczono na wstępie, kolejne rozdziały, a także opisane implementacje będą prawie wyłącznie korzystać z modelu dysku Poincaré. Wydaje się to być właściwym wyborem, z uwagi na wartości

estetyczne i wspomnianą zgodność tego modelu.



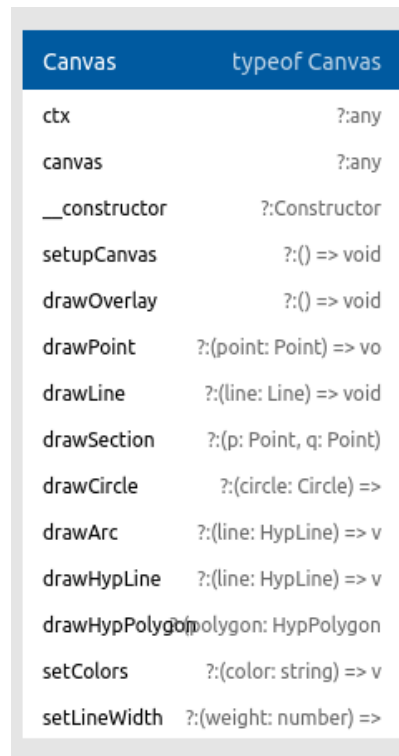
Rysunek 9: Porównanie modeli Kleina, dysku Poincaré i półpłaszczyzny Poincaré

3 Projekt systemu

W niniejszym rozdziale przedstawiony zostanie szczegółowy projekt systemu, jego matematyczną interpretację, zależności pomiędzy klasami oraz podstawowe algorytmy składające się na logikę funkcjonowania silnika.

3.1 Cykl pracy silnika

Głównym plikiem silnika jest `main.ts` znajdujący się w katalogu `/src`. Po załadowaniu programu, tworzy on instancje klasy `Canvas` odpowiedzialnej za rysowanie elementów na ekranie, ładuje konfigurację wyświetlanego programu i tworzy pętlę silnika poprzez wywołanie metody `createLoop()` klasy `Engine`.



Canvas	typeof Canvas
ctx	?:any
canvas	?:any
__constructor	?:Constructor
setupCanvas	?:() => void
drawOverlay	?:() => void
drawPoint	?:(point: Point) => vo
drawLine	?:(line: Line) => void
drawSection	?:(p: Point, q: Point)
drawCircle	?:(circle: Circle) =>
drawArc	?:(line: HypLine) => v
drawHypLine	?:(line: HypLine) => v
drawHypPolygon	polygon: HypPolygon
setColors	?:(color: string) => v
setLineWidth	?:(weight: number) =>

Rysunek 10: Diagram klasy Canvas

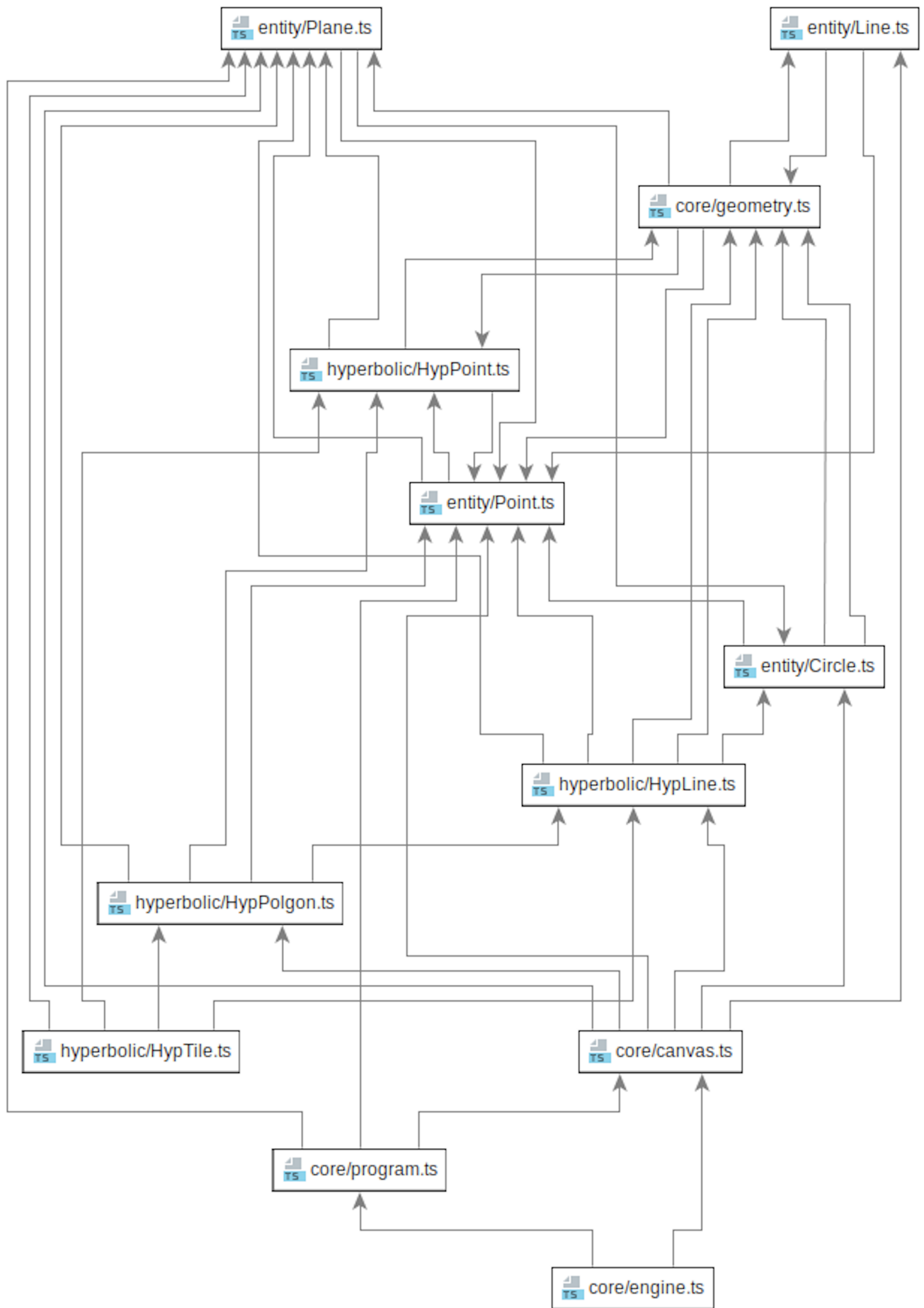
Moduł odpowiedzialny za renderowanie obrazu znajduje się w pliku `canvas.ts`. Konstruktor klasy `Canvas` przyjmuje element `canvas` ze strony oraz jego kontekst, oraz inicjuje się poprzez wywołanie funkcji `setupCanvas()`, która ustala szerokość i wysokość elementu. W każdym cyklu silnika, wywołana jest funkcja `drawOverlay()`, która resetuje element do podstawowego widoku. Kolejne funkcje klasy odpowiadają za rysowanie punktów, linii, łuków i wielokątów. Poza tym klasa udostępnia też funkcje zmiany koloru rysowanych elementów i grubości linii.

Klasa `Engine` przyjmuje konfigurację z pliku `/assets/config.json`, która ustala ilość FPS, wywołuje następnie metodę `drawOverlay()` klasy `Canvas` i odpala funkcję `onLoop()` z programu, konfigurację którego dostaje za pomocą *dependency injection* w parametrach konstruktora.

Odtwarzany program tworzony jest poprzez wywołanie instancji klasy programu, dziedziczącej po abstrakcyjnej klasie `Program`, udostępniającej metody takie jak `onLoop()`.

3.2 Typy obiektów renderowanych przez silnik

Każdy możliwy do narysowania obiekt jest instancją jednej z klas. W kodzie silnika istnieje wyraźny podział na klasy udostępniające obiekty rysowane w przestrzeni euklidesowej i hiperbolicznej. Wszystkie byty znajdują się w katalogu `/src/core/entity`. Kolejne rozdziały są poświęcone opisowi i interpretacji poszczególnych klas.



Engine	typeof Engine
interval	?:number
canvas	?:Canvas
__constructor	?:Constructor
createLoop	?:(program: Program) =
removeLoop	?:() => void

Rysunek 11: Diagram klasy Engine

Program	typeof Program
canvas	?:Canvas
plane	?:Plane
point	?:Point
__constructor	?:Constructor
onLoop	?:() => void

Rysunek 12: Diagram klasy Program

3.3 Obiekty geometrii Euklidesowej

Instancje klas opisanych poniżej są obiektami rysowanymi finalnie przez silnik. Zdefiniowanie ich jest konieczne, gdyż na płaskim ekranie całość sprowadza się do rysowania linii, łuków, kół i punktów w przestrzeni Euklidesowej. Każdy byt przestrzeni hiperbolicznej zawiera instancję przynajmniej jednej z poniższych klas i to właśnie one są interpretowane i rysowane przez klasę **Canvas**.

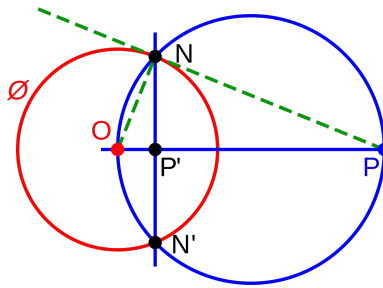
3.3.1 Klasa Point

Konstruktor klasy **Point** przyjmuje dwie zmienne typu **number**, które są reprezentacją bezwzględnych współrzędnych punktu na płótnie. Programista może skorzystać z metody **toHypPoint(plane: Plane): HypPoint**, która przyjmuje instancję klasy **Plane** i zwraca dla niej współrzędne punktu w interfejsie klasy **HypPoint**, oraz z metody **inversion(plane: Plane)**, zwracającej punkt będący inwersją względem płaszczyzny **Plane** (sfery hiperbolicznej). Funkcja **inversion** odgrywa ważną rolę w obliczaniu zakrzywień linii na przestrzeni hiperbolicznej.

3.3.2 Klasa Line

Konstruktor klasy **Line** przyjmuje dwie zmienne typu **number**. Programista może skorzystać z metody **at(x: number): number**, która zwraca wartość w punkcie **x** oraz **intersectPoint(line: Line): Point**, która zwraca punkt przecięcia tej linii z inną linią.

Alternatywnymi sposobami na stworzenie instancji klasy **Line** jest skorzystanie ze statycznych metod: **fromPoints(p: Point, q: Point)** tworzy linię z dwóch punktów, natomiast **fromPointSlope(p: Point, q: number)** do stworzenia linii potrzebuje podania punktu i kąta wyrażonego w radianach.



Rysunek 13: Inwersja punktu P względem okręgu

3.3.3 Klasa Circle

Konstruktor klasy `Circle` przyjmuje punkt centralny będący instancją klasy `Point` i średnicę typu `number`, oraz udostępnia metodę `intersectPoints(circle: Circle): [Point, Point]`, przyjmującą drugi okrąg i zwracającą parę punktów, w których przecinają się oba obiekty. Funkcja `fromPoints(p: Point, q: Point, r: Point)` umożliwia alternatywny sposób tworzenia okręgu z trzech obiektów klasy `Point`. Algorytm za to odpowiedzialny opisany jest poniżej.

3.3.4 Klasa Plane

Najważniejszym z pośród omawianych dotychczas bytów jest instancja klasy `Plane`, będąca singletonem i punktem odniesienia do wszystkich obiektów dla geometrii hiperbolicznej.

Klasa `Plane` dziedziczy po klasie `Circle`, podobnie jak ona posiada centrum i średnicę, liczone jednak są automatycznie na podstawie szerokości i wysokości ekranu przy pierwszym wywołaniu instancji klasy.

3.4 Obiekty geometrii hiperbolicznej

Kod źródłowy klas opisanych poniżej znajduje się w oddzielnym katalogu silnika - `/hyperbolic`. Każdy z tych obiektów opisuje byt geometrii hiperbolicznej rysowany następnie przez silnik w formie prostych linii, okręgów i łuków.

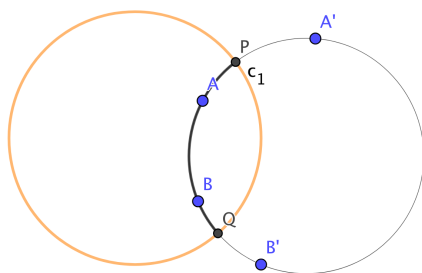
3.4.1 Klasa HypLine

Klasa `HypLine` jest pierwszą z pośród klas obiektów hiperbolicznych. Konstruktor klasy przyjmuje - podobnie jak ten klasy `Line`, dwa punkty oraz dodatkowo instancję klasy `Plane`.

Konstruktor klasy wywołuje metodę `calculateArc(p: Point, q: Point, plane: Plane): Circle`, która z pomocą algorytmu opisanego poniżej, zwraca instancję klasy `Circle`, będącą okręgiem, na obwodzie którego leży dana prosta hiperboliczna. Ustala jednocześnie punkty `p` i `q` wyznaczające końce odcinka, posługując się przy tym metodą `cutIfSticksOut(point: Point, circle: Circle, plane: Plane): Point`, sprawdzając, czy punkt nie leży poza granicą koła wyznaczonego przez obiekt klasy `Plane` i ewentualnie przesuwając go na punkt przecięcia obu okręgów używając do tego wspomnianej już metody `intersectPoints(circle: Circle): [Point, Point]`.

Niech A i B będą punktami na dysku Poincarégo, a punkty A' i B' będą ich inwersjami na płaszczyźnie `Plane`. Potrzebujemy okręgu przez A i B , który jest prostopadły do `Plane`.

Podczas konstruowania okręgu przez A i B , dowolny z odbijanych punktów A' lub B' może być użyty do zdefiniowania okręgu. Jeśli jeden z punktów ma współrzędne $(0, 0)$, należy



Rysunek 14: Kontrukcja linii w przestrzeni hiperbolicznej

użyć drugiego punktu. $((0, 0)$ odzwierciedla nieskończoność, która w tym kontekście jest niezdefiniowanym punktem).

Algorytm wyznaczania okręgu na podstawie dwóch punktów i płaszczyzny:

1. Sprawdź współrzędne punktu p : jeżeli są takie same jak współrzędną centrum Plane , przypisz q do zmiennej `validPoint`, w przeciwnym wypadku przypisz p .
2. Oblicz dwusieczną punktów p i q oraz q i `validPoint.inversion(plane)`.
3. Znajdź centrum nowego okręgu będące punktem przecięcia obu linii z pomocą funkcji `Line.intersectPoint(Line)`
4. Znajdź promień nowego okręgu licząc odległość euklidesową pomiędzy jednym z początkowych punktów a punktem przecięcia dwusiecznych.

Ostatnią nieomówioną funkcją jest `countAngle(circle: Circle)`, określającą na podstawie wcześniej obliczonych punktów, początkowy i końcowy kąt łuku oraz kierunek, w jakim rysowany będzie ten łuk. Ma to znaczenie dla klasy `Canvas` i pomaga w ustaleniu, gdzie znajduje się wnętrze rysowanej figury.

3.4.2 Klasa `HypPoint`

Klasa `HypPoint` to w rzeczywistości reprezentacja punktu względem płaszczyzny hiperbolicznej w dziedzinie $(-1, 1) \times (-1, 1) \in \mathbb{R} \times \mathbb{R}$.

Klasa udostępnia metodę `toCanvasCoords(): Point`, zwracającą instancję tego samego punktu, zdolną do wyświetlenia przez aplikację, funkcję `reflect(point: HypPoint): HypPoint` - zwracającą odbicie tego punktu względem innego, podanego w argumentach, co jest wymagane do poprawnego rysowania obiektów na przestrzeni i dwie prywatne, pomocnicze funkcje `times(point: HypPoint | number): HypPoint` oraz `over(point: HypPoint | number): HypPoint` służące kolejno do mnożenia lub dzielenia danego punktu przez stałą lub inny punkt.

Najważniejszą metodą tej klasy jest `moebius(point: HypPoint, t: number): HypPoint`. Aby zrozumieć jej działanie potrzebne będzie zdefiniowanie *Transformacji Möbiusa* i jej udziału w obliczaniu punktu na przestrzeni dysku Poincaré. Opisana jest ona na końcu tego rozdziału.

3.4.3 Klasa `HypPolygon`

Konstruktor klasy `HypPolygon` przyjmuje dwie zmienne typu `Point` oraz instancję klasy `Plane` i tworzy z nich wielokąt na przestrzeni hiperbolicznej.

Wielokąt może zostać rozszerzony o kolejne punkty z pomocą metody `addVerticle(point: Point)`. Funkcja `getCompletePolygonLines(): HypLine[]` zwraca wszystkie odcinki wchodzące w skład wielokąta, wraz z jednym dodatkowym odcinkiem, łączącym pierwszy i ostatni wierzchołek. Funkcje `moebius(point: HypPoint, t: number): HypPolygon` oraz `reflect(point: HypPoint): HypPolygon` wykonują kolejno transformację Möbiusa oraz odbicie względem punktu na wszystkich wierzchołkach wielokąta.

Programista może skorzystać ze statycznej metody `fromVertices(verts: Point[], plane: Plane): HypPolygon`, która przyjmuje tablicę punktów oraz instancję klasy `Plane` i zwraca gotowy wielokąt.

3.4.4 Klasa HypTile

Klasa `HypTile` jest nietypową na tle swoich poprzedniczek. Konstruktor tej klasy jest prywatny, a stworzenie jej instancji odbywa się za pomocą jednej z trzech metod statycznych:

- `fromPolygon(polygon: HypPolygon, center: HypPoint, plane: Plane): HypTile` - funkcja tworzy obiekt klasy `HypTile` wykorzystując do tego instancję obiektu klasy `HypPolygon`
- `createNKPolygon(n: number, k: number, center: HypPoint, plane: Plane): HypTile` - Tworzy n -ką o wielkości i kątach dobranych w ten sposób, by przy układaniu ich obok siebie, tworzyły przestrzeń będącą k -kątem (k = liczba n -gonów ‘spotykających się’ na każdym wierzchołku).
- `createRegularPolygon(numOfVerts: number, distance: number, center: HypPoint, plane: Plane, startAngle = 0): HypTile` - funkcja tworzy wielokąt foremny o podanych parametrach.

3.5 Funkcje dodatkowe

Plik `geometry.ts` zawiera zestaw funkcji wspólnych dla wielu obiektów, lub nie powiązanych bezpośrednio z żadnym z nich. Są to głównie funkcje czysto matematyczne - geometryczne, takie jak odległość Euklidesowa lub liczenie dwusiecznej.

3.6 Transformacja Möbiusa

Twierdzenie 1 *Transformacja Möbiusa jest funkcją na rozszerzonej płaszczyźnie zespolonej określoną równaniem*

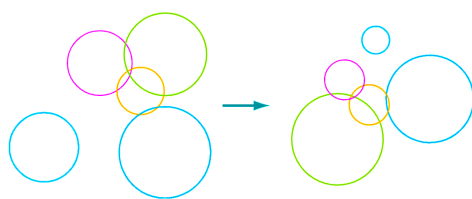
$$f(z) = \frac{az + b}{cz + d}, \text{ gdzie } ad - bc \neq 0$$

transformacja Möbiusa = złożenie inwersji = izometrie hiperboliczne

Hiperboliczne symetrie są modelowane jako przekształcenia Möbiusa: ³

Transformacje Möbiusa (zwane również homografiami) tworzą grupę geometryczną. Odwrócenie przestrzeni przez sferę ze środkiem w punkcie O i promieniu r , odwzorowuje na siebie wszystkie promienie pochodzące z tego, że iloczyn punktu na tym promieniu wraz z jego obrazem jest równy r^2 . Transformacje Möbiusa zachowują również kąty. Izometria geometrii hiperbolicznych to właśnie transformacje Möbiusa. W ten sposób, z ich pomocą możemy nawigować po przestrzeni hiperbolicznej, płynnie przesuując punkt widzenia modelu dysku Poincaré.

³HyperbolicTransformations, Chapter 17



Rysunek 15: Transformacja Möbiusa

4 Implementacja systemu

W niniejszym rozdziale omówiona zostanie technologia, konfiguracja oraz wdrożenie systemu wraz z krótkim opisem poszczególnych części systemu i kodu źródłowego.

4.1 Opis technologii

Do implementacji systemu użyto języka **TypeScript** w wersji 3.6.3, bundlera (transpilatora nowocześniejszych wersji języka **JavaScript** do wersji zrozumiałych dla przeglądarek) **webpack** w wersji 2.3.3 oraz **SCSS** i **HTML5** wraz z elementem `<canvas>` odpowiedzialnym za rysowanie grafiki na ekranie. Użyta została również funkcyjna biblioteka **ramda** w formie pomocniczej biblioteki *utilsowej*. Pełna lista wszystkich bibliotek wraz z ich wersjami znajduje się w pliku `package.json`, w katalogu głównym projektu.

4.2 Poszczególne składowe systemu

Aplikacja budowana jest ze źródeł z pomocą konfiguracji **webpackowej**. Kolejne paragrafy zawierają opisy i przeznaczenie poszczególnych plików oraz ogólny projekt całej aplikacji.

4.3 Konfiguracja systemu

Konfiguracja systemu potrzebna do zbudowania silnika znajduje się w całości w katalogu głównym.

4.3.1 Biblioteki projektu

Biblioteki potrzebne do zbudowania aplikacji wraz z ich wersjami znajdują się w pliku `package.json`. Instalują się one do katalogu `node_modules` po wpisaniu komendy `npm install`. Aby zbudować aplikację potrzebne jest połączenie z internetem.

4.3.2 Bundlowanie aplikacji

Do bundlowania aplikacji użyty został framework **webpack**. Jego konfiguracja znajduje się w pliku `webpack.config.js` w katalogu głównym. Określa ona, gdzie znajdują się pliki źródłowe, jakie mają rozszerzenia i w jaki sposób powinny być kompilowane. Do konfiguracji dołączone jest również rozszerzenie **style-loader**, które kompiluje pliki stylów o formacie **scss**.

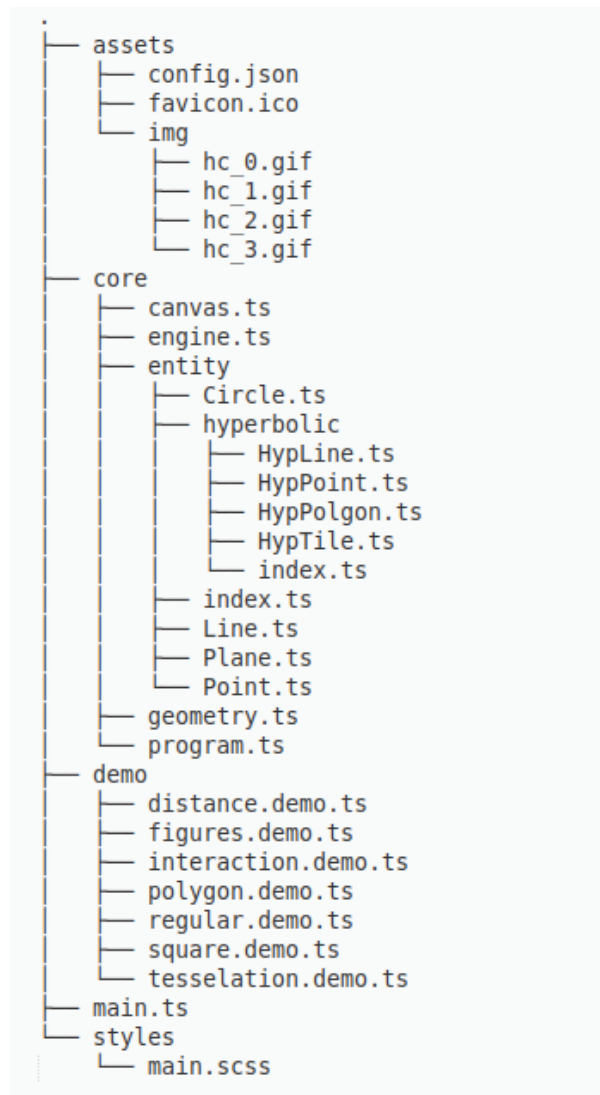
4.3.3 Konfiguracja języka

Język **Typescript** wymaga pliku `tsconfig.json` w katalogu głównym projektu. Plik `tsconfig.json` określa pliki główne i opcje kompilatora wymagane do skompilowania projektu.

4.4 Pliki źródłowe silnika

Źródła systemu umieszczone są w całości w katalogu `/src/core`. Opis poszczególnych klas i przepływ pracy programu znajduje się w poprzednim rozdziale. Katalog `styles` zawiera plik stylów, który budowany jest razem z resztą aplikacji z pomocą **webpacka**, natomiast folder `demo` zawiera programy demonstracyjne. Opis niektórych programów, co za tym idzie - możliwości silnika znajduje się poniżej. W katalogu `assets` znajduje się plik konfiguracyjny dla klasy `Canvas`.

Każdy program demonstracyjny dziedziczy po klasie `Program`. Klasa bazowa udostępnia metodę `onLoop()`, w której umieszcza się instrukcje do wykonania przez silnik oraz zmienna `point` definiująca aktualne położenie wskaźnika myszy. Instancja klasy `Canvas` dostarczana jest poprzez wzorzec `dependency injection`.



Rysunek 16: Schemat katalogów i plików źródłowych

4.4.1 Polygon Demo

Program **Polygon Demo** prezentuje możliwości rysowania linii i wielokątów na dysku Poincaré. Klasa zawiera zmienną globalną `polygon` typu `HypPolygon`, która definiowana jest po wybraniu dwóch punktów na dysku. Wybór punktu odbywa się poprzez kliknięcie lewym przyciskiem myszy na ekranie.

Funkcja `onLoop()` zawiera instrukcje rysowania wielokąta, co ogranicza się do wywołania metody `canvas.drawHypPolygon(this.polygon)`. Podobnie działa rysowanie punktów i linii. Programista nie musi znać wewnętrznych implementacji, jedynie api udostępniane przez klasy silnika.

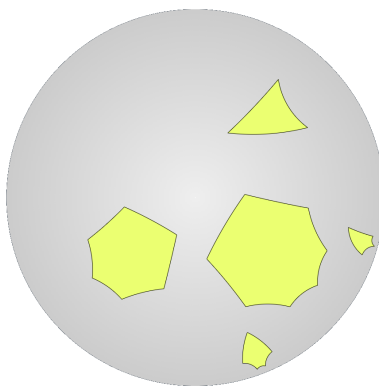
Program **Polygon** pokazuje również możliwości manipulowania grubością linii oraz kolorami płótna. W przykładzie jest to osiągnięte za pomocą wywołania funkcji klasy `Canvas` - `canvas.setColors("#FFF")`.



Rysunek 17: Wielokąt narysowany przy użyciu programu **Polygon Demo**

4.4.2 Interaction Demo

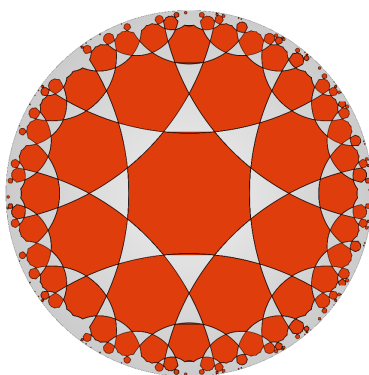
Program **Interaction Demo** zawiera wykorzystanie klasy `HypTile`. W każdym przebiegu pętli silnika, dookoła wskaźnika myszy zdefiniowanego zmienną `point`, z pomocą statycznej metody `createRegularPolygon()` tworzone są wielokąty foremne. Zmienna globalna `rotate`, definiuje kąt obrotu każdej z figur. Po narysowaniu wszystkich figur, zmienna ta jest inkrementowana, po czym wyświetlana jest następna klatka obrazu.



Rysunek 18: Przykład działania programu **Interaction Demo**

4.4.3 Tessellation Demo

Program `Tessellation Demo` różni się od innych demonstracji. Pętla silnika wyświetla raz już zdefiniowany obraz, rysując wszystkie kafelki umieszczone w tablicy `tiles` interfejsu `HypTile[]`. Konstruktor klasy wywołuje metodę `determineTiles()`, która tworzy pierwszy kafelek za pomocą statycznej metody `createNKPolygon()` a następnie, określoną ilość razy odbija jego obraz, co skutkuje wypełnieniem dysku przylegającymi do siebie kafełkami. Do tego celu została użyta opisana w poprzednim rozdziale funkcja `reflect()`.



Rysunek 19: ‘Kafelkowanie’ dysku wykonane przez program `Tessellation Demo`

4.5 Pliki źródłowe pracy

Katalog `/docs` zawiera źródła tej pracy, budowane za pomocą skryptu zamieszczonego w pliku `makefile` z wykorzystaniem programu `pandoc` i biblioteki `texlive`. Praca napisana jest w języku `markdown`. Katalog `/docs/figures` zawiera statyczne pliki. Strona tytułowa napisana jest w języku `latex` i budowana jest osobno.

5 Instalacja i wdrożenie

Rozdział ten zawiera informacje o sposobie zbudowania aplikacji w celu jej uruchomienia i opcjonalnie - wdrożenia na serwerze WWW.

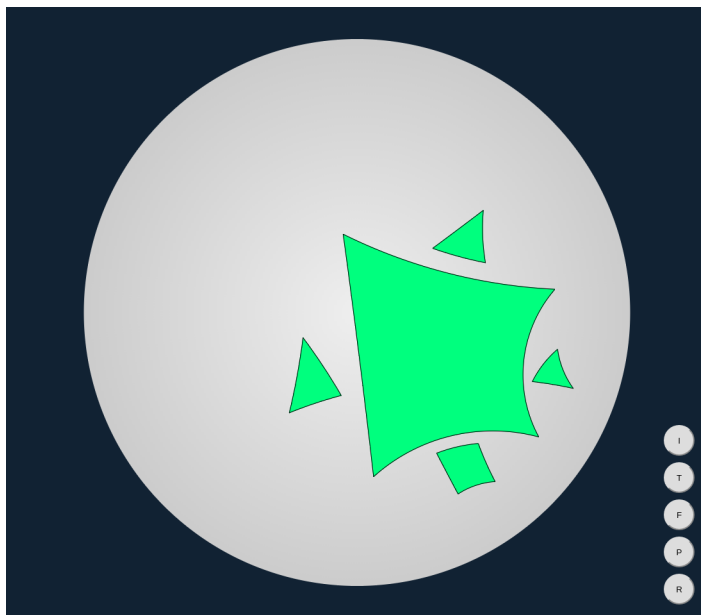
Do zbudowania aplikacji konieczny będzie menadżer pakietów `npm` w wersji przynajmniej 6.5.0 oraz środowisko uruchomieniowe języka JavaScript - `node.js` w wersji 10.6.0 lub nowszej. Instalacja wymaganych pakietów odbywa się poprzez wpisanie w konsoli polecenia

```
npm install
```

w katalogu głównym projektu. Następnie należy zbudować aplikację poleceniem

```
npm run build
```

Po zbudowaniu aplikacji, w katalogu głównym pojawi się folder `dist` z plikami, które wraz z plikiem `index.html` składają się na gotowy program możliwy do uruchomienia w przeglądarce.



Rysunek 20: Wygląd aplikacji po uruchomieniu

5.1 Serwer deweloperski

Aplikacja wspiera tryb deweloperski, w którym bieżące zmiany w kodzie automatycznie są budowane do plików wynikowych. Do uruchomienia trybu deweloperskiego potrzebne są te same pakiety instalowane poleceniem:

```
npm install
```

Wywołanie trybu odbywa się komendą:

```
npm run build-watch
```

5.2 Wdrożenie na serwerze WWW

Projekt można wystawić na serwerze WWW. Sposób wdrożenia zależy od posiadanego serwera. Nie należy jednak umieszczać na serwerze całego katalogu z projektem. Zalecane jest przede wszystkim usunięcie katalogu `node_modules`. Do poprawnego działania projektu wystarczy plik `index.html` oraz katalog `/dist` pojawiający się po zbudowaniu aplikacji.

6 Podsumowanie

Praca została napisana w oparciu o analizę zagadnienia. Zamierzony efekt pracy - to jest skonstruowanie silnika graficznego renderującego geometrię dysku Poincare udało się osiągnąć, na co wskazują programy demonstracyjne dla silnika. Jest to autorskie, unikalne rozwiązanie, pozwalające na kompleksową obsługę zadanego modelu. Użycie nadal niestandardowych technologii webowych takich jak silnie typowany język **Typescript** umożliwia przyjemną pracę z silnikiem, na co składa się również dobrze napisana warstwa renderująca grafikę, pozwalająca w sposób bezpośredni wyświetlić dowolny, wspierany interfejsami silnika byt lub figurę.

Projekt można w przyszłości rozszerzyć o wsparcie dla obrazków, co umożliwiłoby łatwą implementację grafik Eschera, co jednak jest możliwe nawet teraz z wykorzystaniem odbijanych względem siebie wielokątów, podobnie, jak zostało to osiągnięte w programie **Tesselation Demo**.

7 Bibliografia

- Joan Gómez, Tam, gdzie proste są krzywe, Geometrie nieeuklidesowe, RBA, 2010
- Martin Freiherr von Gagern, Creation of Hyperbolic Ornaments Algorithmic and Interactive Methods, Technischen Universität München
- Mateusz Kłeczek, Geometria hiperboliczna, Chrzanów 2016
- Bjørn Jahren, An introduction to hyperbolic geometry, MAT4510/3510
- Izabela Przedzink, Geometria Poincaré i Kleina. Skrypt do zajęć: Podstawy geometrii i elementy geometrii nieeuklidesowej, Wrocław 2010, Uniwersytet Wrocławski Wydział Matematyki i Informatyki Instytut Matematyczny
- Stefan Kulczycki Biblioteka Problemów Geometria Nieeuklidesowa Warszawa 1960, Państwowe Wydawnictwo Naukowe
- Marek Kordos, O różnych geometriach, Warszawa 1987, Wydawnictwa Alfa
- Caroline Series With assistance from Sara Maloni, Hyperbolic geometry MA448
- Marshall Bern, Optimal Möbius Transformation for Information Visualization and Meshing
- Steve Szidlik, Hyperbolic Constructions in Geometer's Sketchpad, December 21, 2001
- Douglas N. Arnold and Jonathan Rogness, Möbius Transformations Revealed
- Frank Nielsen and Richard Nock, Hyperbolic Voronoi diagrams made easy
- Marek Kordos, Geometria Bolyai–Łobaczewskiego, <http://www.deltami.edu.pl>, Sierpień 2018

8 Zawartość płyty CD

Płyta CD zawiera cały kod źródłowy programu, zbudowany w katalogu `/dist` projekt oraz katalog `/docs` zawierający źródła tej pracy oraz jej końcową wersję w postaci pliku `pdf`.