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

Spis treści

1 Wstęp

1.1 Wprowadzenie

Celem pracy jest zaprogramowanie i implementacja silnika graficznego służącego do generowania podstawowych obiektów geometrycznych w geometrii hiperbolicznej. Wybrano jej model - tak zwany dysk Poincaré. Biblioteka została napisana w języku `Typescript` (który jest wersją języka `JavaScript` zawierającą ponadto między innymi silne typowianie).

Praca swoim zakresem obejmie obsługę rysowania lini, okręgów, wielokątów na tejże płaszczyźnie oraz implementację przykładowych programów obejmujących wizualizacje bardziej skomplikowanych struktur. Na tle innych implementacji, aplikacja wyróżnia się dostarczanymi możliwościami 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 dysku 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.

Udało się zrealizować wszystkie postawione cele.

1.2 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. [a]

[a] Geometria euklidesowa. Encyklopedia PWN

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 postulatu w aksjomatyce – obecność postulatu wzięła się z pospiesznego 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. [a]
>
> [a] Najgłupiej postawiony problem matematyki. Marek Kordos - Delta, maj 2012

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.

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.

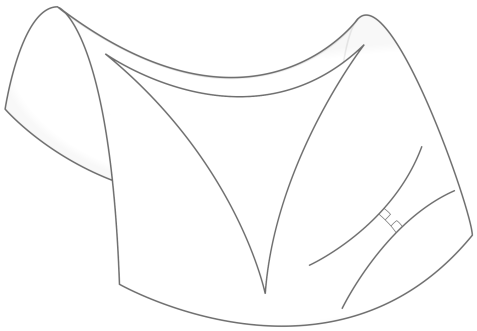
2 Analiza problemu

W tym 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 krzywiźnie, 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

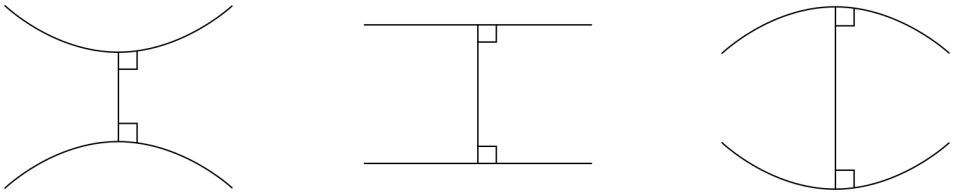
Sposobem opisania różnic między tymi geometriami jest rozważenie dwóch linii prostych rozciągniętych w nieskończoność w płaszczyźnie dwuwymiarowej, które są prostopadłe do trzeciej linii:

- W geometrii euklidesowej linie pozostają w stałej odległości od siebie (co oznacza, że linia narysowana prostopadle 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

Ta 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.

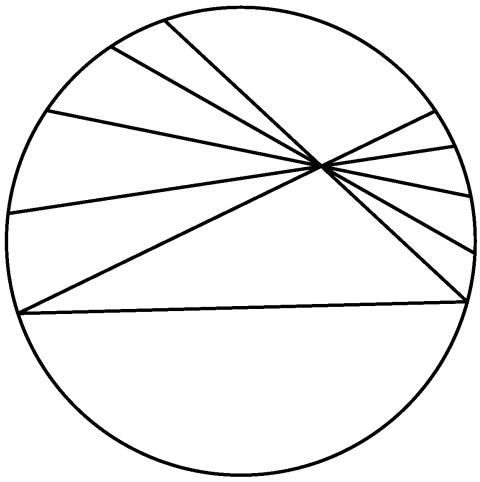
2.2 Popularne modele geometrii hiperbolicznej

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

2.2.1 Model Kleina

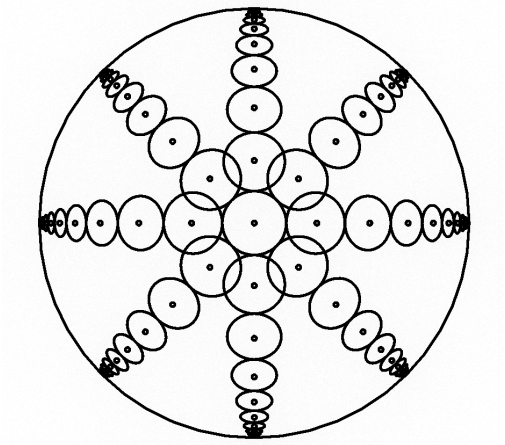
Model Kleina - a w zasadzie model dysku Beltrami–Kleina jest modelem geometrii hiperbolicznej, w którym punkty są reprezentowane przez punkty we wnętrzu dysku. Przyjmuje on następujące założenia:

- **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.

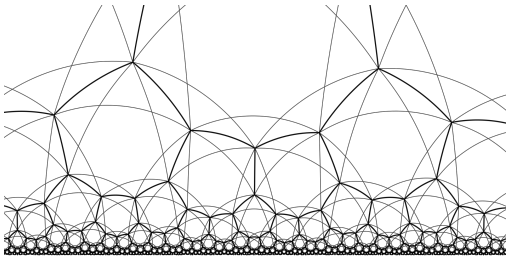


Rysunek 4: Graficzne zobrazowanie kół w modelu Kleina

2.2.2 Model półpłaszczyzny Poincaré

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

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



Rysunek 5: Przykład tesselacji w modelu półpłaszczyzny Poincaré

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.

2.2.3 Model dysku Poincaré

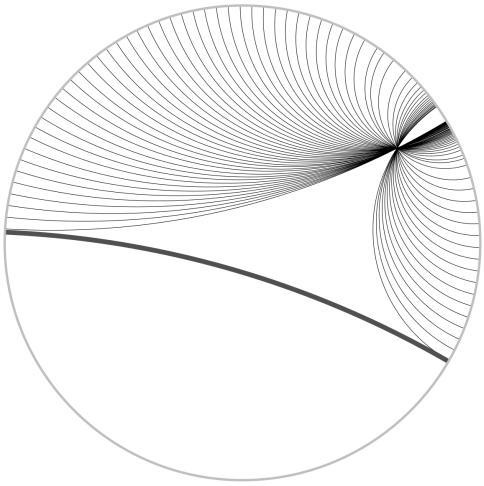
Model dysku Poincaré wykorzystuje wnętrze dysku jako model płaszczyzny hiperbolicznej.
W poniższych przykładach omawiany będzie dysk jednostkowy, który będzie również przedmiotem dalszych rozważań.

- **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 wzór:[1]

$$d_H(z_0, z_1) = ln(\frac{|1 - z_0\overline{z_1}| + |z_0 - z_1|}{|1 - z_0\overline{z_1}| - |z_0 - z_1|})$$

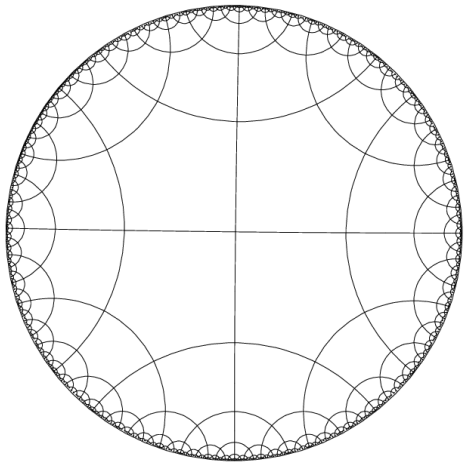
Ponieważ rozpatrywany jest dysk jednostkowy, powyższy wzór nie zawiera w zmiennej dla promienia.

[1] Hyperbolic Transformations, Chapter 17, Equation 17.2.13



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.

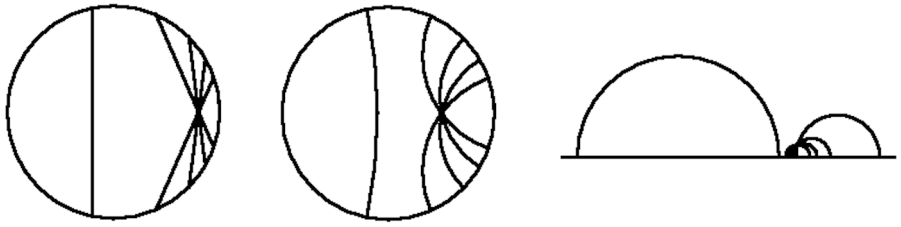
- **Punkty hiperboliczne** to punkty na półkuli południowej.
- **Linie hiperboliczne** to półkola powstałe z przecięcia półkuli południowej z płaszczyznami prostopadłymi do równika.

Rysunek 8: Rzut na dysk Poincaré (a) i projekcja do modelu Klein-Beltrami (b) [Martin Freiherr von Gagern, Creation of Hyperbolic Ornaments]

Wadą omawianego rozwiązania, jest dodatkowy wymiar, jaki należy rozpatrzeć 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

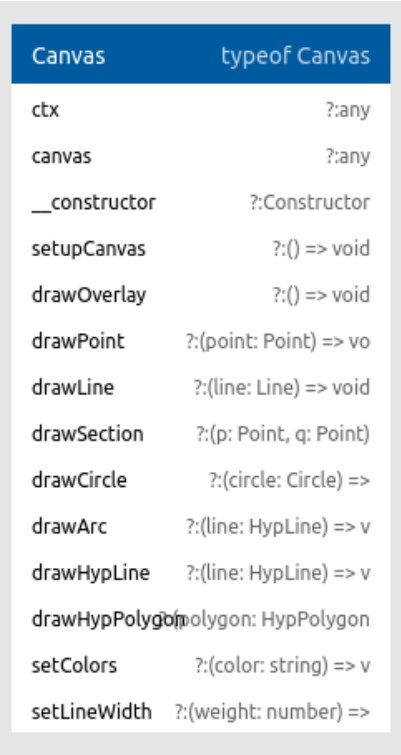
3 Projekt systemu

W tym rozdziale przedstawiony zostanie szczegółowy projekt systemu, jego matematyczna interpretacja, zależności pomiędzy klasami oraz podstawowe algorytmy składające się na logikę funkcjonowania silnika.

3.1 Cykl pracy silnika

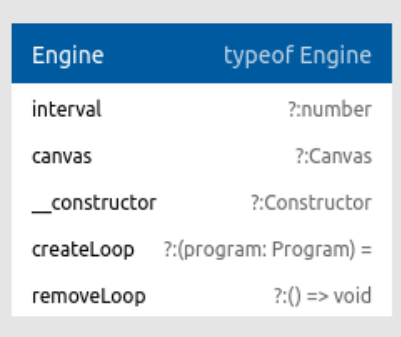
Głównym plikiem silnika jest `main.ts` znajdujący się w katalogu `/src`. Po uruchomieniu programu, tworzy on instancje klasy `Canvas` odpowiedzialnej za rysowanie elementów na ekranie, ładuje konfiguracje wyświetlanego programu i tworzy pętlę silnika poprzez wywołanie metody `createLoop()` klasy `Engine`.

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



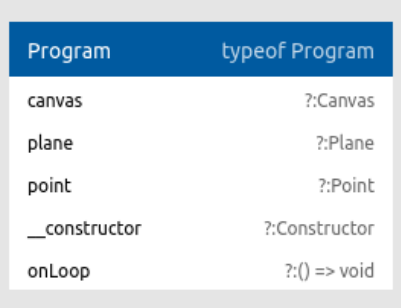
Rysunek 10: Diagram klasy Canvas

Klasa `Engine` przyjmuje konfigurację z pliku `/assets/config.json`, która ustala ilość FPS. Jednocześnie wywołuje metodę `drawOverlay()` klasy `Canvas` i uruchamia funkcję `onLoop()` z programu. Konfiguracja programu dostarczona jest z pomocą wzorca *dependency injection* w parametrach konstruktora.



Rysunek 11: Diagram klasy Engine

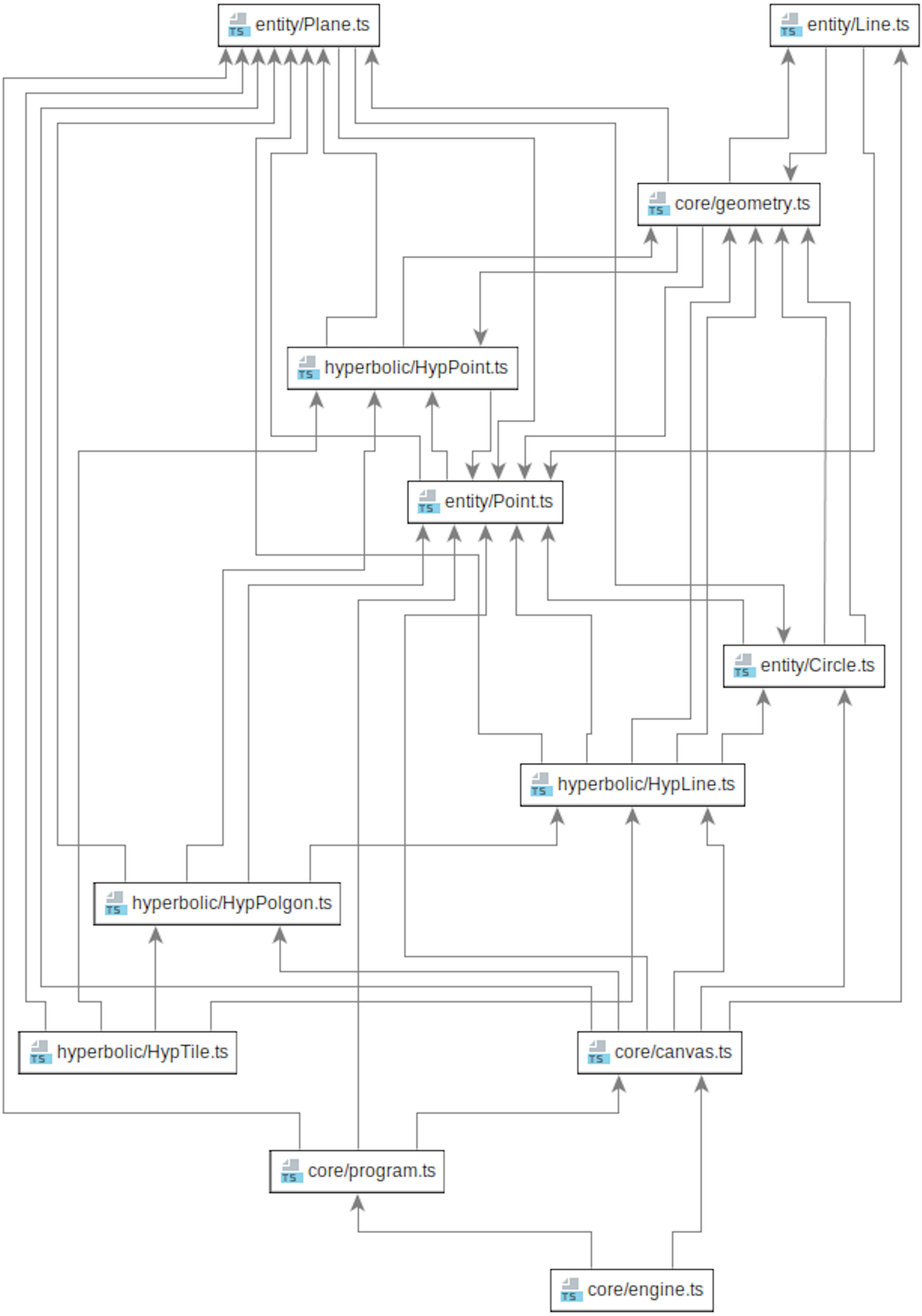
Odtwarzany program tworzony jest poprzez wywołanie instancji klasy, dziedziczącej po abstrakcyjnej klasie `Program`, która udostępnia metody takie jak `onLoop()`.



Rysunek 12: Diagram klasy Program

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. Źródła wszystkich elementów znajdują się w katalogu `/src/core/entity`. Kolejne rozdziały są poświęcone opisie i interpretacji poszczególnych klas.


entity/Plane.ts
entity/Line.ts
core/geometry.ts
hyperbolic/HypPoint.ts
entity/Point.ts
entity/Circle.ts
hyperbolic/HypLine.ts
hyperbolic/HypPolgon.ts
hyperbolic/HypTile.ts
core/canvas.ts
core/program.ts
core/engine.ts

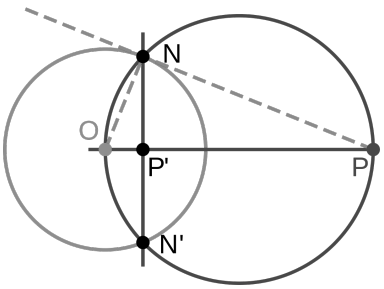
3.3 Obiekty geometrii Euklidesowej

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 element przestrzeni hiperbolicznej zawiera instancję przynajmniej jednej z poniższych klas. 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 koordynatów punktu na płótnie. Programista może skorzystać z metod `toHypPoint(plane: Plane): HypPoint` oraz `inversion(plane: Plane)`.

Funkcja `inversion(plane: Plane)` przyjmuje instancję klasy `Plane` (sfery hiperbolicznej) i zwraca dla niej koordynaty punktu w interfejsie klasy `HypPoint`, natomiast funkcja `inversion(plane: Plane)` zwraca punkt będący inwersją tego punktu względem płaszczyzny `Plane`. Funkcja `inversion` odgrywa ważną rolę w obliczaniu zakrzywień linii na przestrzeni hiperbolicznej.



Rysunek 13: Przykład inwersji punktu P względem okręgu

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 lini z inną linią.

Alternatywnymi sposobami na stworzenie instancji klasy `Line` jest skorzystanie ze statycznych metod:

- `fromPoints(p: Point, q: Point)` - Metoda tworzy linię z dwóch punktów.
- `fromPointSlope(p: Point, q: number)` - Metoda do stworzenia linii potrzebuje podania punktu i kąta wyrażonego w radianach.

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)` udostępnia 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 elementów jest klasa `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, które liczone są jednak 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

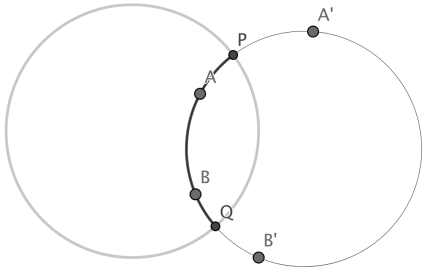
Klasa `HypLine` jest pierwszą z pośród klas obiektów hiperbolicznych. Konstruktor klasy przyjmuje - podobnie jak klasy `Line` - dwa punkty oraz dodatkowo instancję klasy `Plane`.

Klasa zezwala na następujące operacje:

- Funkcja `calculateArc(p: Point, q: Point, plane: Plane): Circle` za pomocą algorytmu opisanego poniżej, zwraca instancję klasy `Circle`. Jest ona w istocie opisem okręgu, na obwodzie którego leży dana prosta hiperboliczna.
- Funkcja `cutIfSticksOut(point: Point, circle: Circle, plane: Plane): Point` ustala punkty p i q wyznaczające końce odcinka oraz sprawdza, czy punkt nie leży poza granicą koła wyznaczonego przez obiekt klasy `Plane`. W takim przypadku przesuwa dany punkt na punkt przecięcia obu okręgów używając do tego wspomnianej już metody `intersectPoints(circle: Circle): [Point, Point]`.

Konstrukcja linii hiperbolicznej na podstawie dwóch punków:

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



Rysunek 14: Konstrukcja linii w przestrzeni hiperbolicznej

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

Algorytm wyznaczania okręgu na podstawie dwóch punktów i płaszczyny:

1. Sprawdź współrzędne punktu p: jeżeli są takie same jak współrzędne 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 `intersectPoint` klasy `Line`.
4. Znadź promień nowego okręgu licząc odległość euklidesową pomiędzy jednym z początkowych punków 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, nadającą się do wyświetlenia przez silnik. Funkcja `reflect(point: HypPoint): HypPoint` zwraca odbicie tego punktu względem innego, podanego w argumentach. Jest to wymagane do poprawnego rysowania obiektów na przestrzeni. Klasa zawiera również 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 jest zdefiniowanie *Transformacji Möbiusa* i jej udziału w obliczaniu punktu na przestrzeni dysku Poincaré. Opis zamieszono w punkcie **3.6**, na końcu tego rozdziału.

3.4.3 Klasa HypPolygon

Konstruktor klasy `HypPolygon` przyjmuje dwie zmienne typu `Point` oraz instancję klasy `Plane`. 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 przekazanego punktu na wszystkich wierzchołkach wielokąta.

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

3.4.4 Klasa HypTile

Klasa `HypTile` jest nietypowa 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 obiekty klasy `HypPolygon`

- `createNKPolygon(n: number, k: number, center: HypPoint, plane: Plane): HypTile` - Zwraca n-kąt o wielkości i kątach dobranych w ten sposób, by przy układaniu ich obok siebie, tworzyły przestrzeń będączą 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 matematyczno - geometryczne, takie jak odległość Euklidesowa lub liczenie dwusiecznej.

3.6 Transformacja Möbiusa

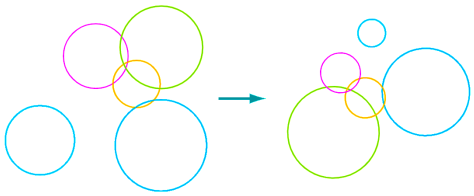
Twierdzenie 1 *Transformacja Möbiusa jest funkcją na rozszerzonej płaszczyźnie zespolonej określoną równaniem*

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

$$transformacja\ Möbiusa = złożenie\ inwersji = izometrie\ hiperboliczne$$

Hiperboliczne symetrie są modelowane jako przekształcenia Möbiusa: [2]

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 takie, 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 przesuwając punkt widzenia modelu dysku Poincaré.



Rysunek 15: Przykładowa transformacja Möbiusa [Marshall Bern, Optimal Möbius Transformation]

Na użytek aplikacji i wybranego modelu użyty został zmodyfikowany wzór Transformacji Möbiusa. [3]

Twierdzenie 2 *Transformacja wyrażona równaniem:*

$$f(z) = \beta\frac{z-\alpha}{\overline{\alpha}z-1},\ gdzie\ |\alpha|<1\ i\ |\beta|=1$$

zachowuje funkcje odległości Poincaré.

[2] Hyperbolic Transformations, Chapter 17, Definition 17.1
[3] Hyperbolic Transformations, Chapter 17, Corollary 17.12

4 Implementacja systemu

W tym rozdziale omówiona zostanie technologia, konfiguracja oraz wdrożenie systemu wraz z krótkim opisem jego poszczególnych składowych i kodu źródłowego.

4.1 Opis technologii

Do implementacji systemu użyto języka `TypeScript` w wersji `3.6.3`, bundlera (transpilatora nowoczesnych 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 na załączonej płycie CD.

4.2 Konfiguracja systemu

Konfiguracja systemu potrzebna do zbudowania silnika znajduje się w całości w katalogu głównym. Aplikacja budowana jest z plików źródłowych z pomocą konfiguracji webpackowej.
Poniżej zamieszczono opisy i przeznaczenie poszczególnych plików oraz ogólny projekt całej aplikacji.

4.2.1 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 aplikacje potrzebne jest połączenie z internetem.

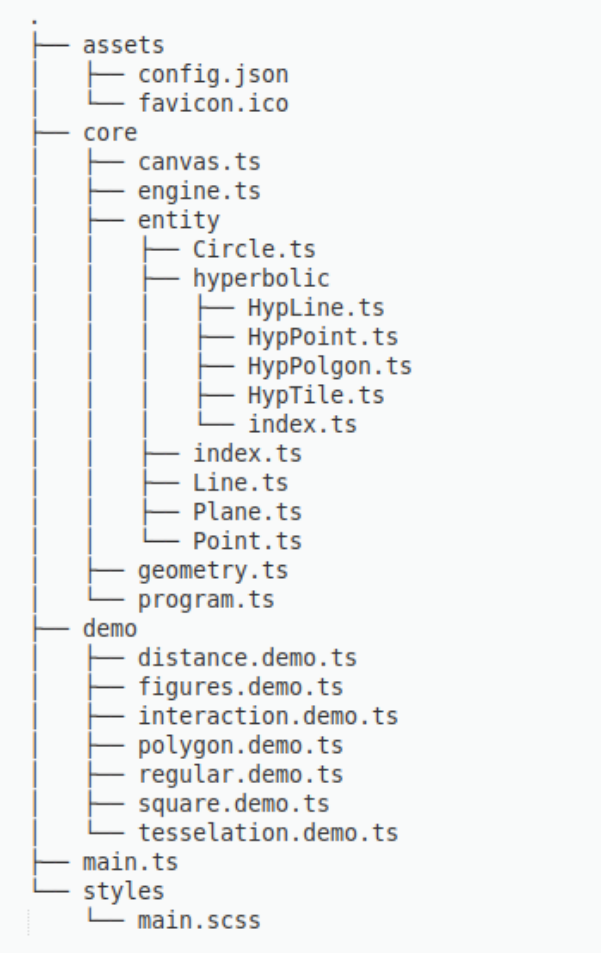
4.2.2 Bundlowanie aplikacji

Do bundolwania 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`.

4.2.3 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.

4.3 Pliki źródłowe silnika



Rysunek 16: Schemat katalogów i plików źródłowych

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 styli, 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`.

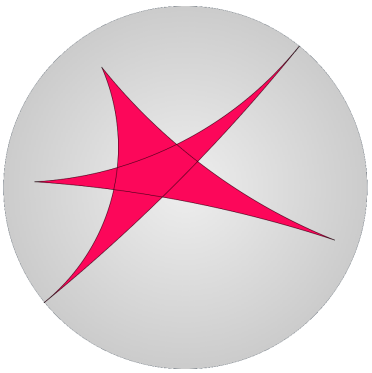
Każdy program demonstracyjny dziedziczy po klasie `Program`. Klasa bazowa udostępnia metodę `onLoop()`, w której umieszcza się instrukcje do wykonania przez silnik oraz zmienna point definiująca aktualne położenie wskaźnika myszy. Instancja klasy `Canvas` dostarczana jest poprzez wzorzec `dependecy injection`.

4.3.1 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 punków na dysku. Wybór punktu odbywa się poprzez klikniecie 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.

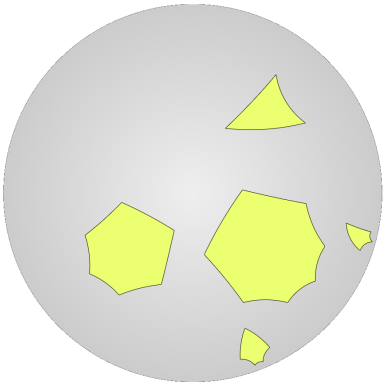
Program `Polygon` pokazuje również możliwości manipulowania grubością linii oraz kolorami płótna. W omawianym przykładzie jest to osiągnięte za pomocą wywołania funkcji klasy `Canvas` - `canvas.setColors("#FFF")`.



Rysunek 17: Przykład wielokąta narysowany przy użyciu programu Polygon Demo

4.3.2 Interaction Demo

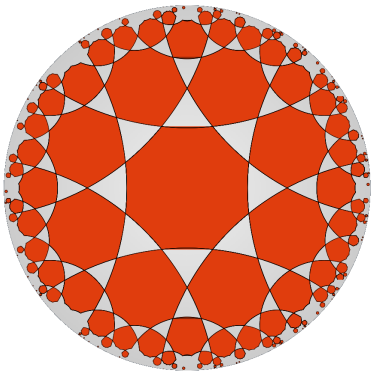
Program `Interaction Demo` zawiera wykorzystanie klasy `HypTile`. W każdym przebiegu pętli silnika, dookoła wskaźnika myszy zdefiniowanego zmienną `point`, z pomocą statycznej metody `createRegularPolygon()` tworzone są wielokąty foremne. Kąt obrotu każdej z figur jest zdefiniowany za pomocą zmiennej globalnej `rotate`. Po narysowaniu wszystkich figur, zmienna ta jest inkrementowana, następnie wyświetlana jest następna klatka obrazu.



Rysunek 18: Przykładowe figury narysowane za pomocą programu Interaction Demo

4.3.3 Tesselation Demo

Program `Tesselation 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 kafelkami. Do tego celu została użyta opisana w poprzednim rozdziale funkcja `reflect()`.



Rysunek 19: 'Kafelkowanie' dysku wykonane przez program Tesselation Demo

4.4 Pliki źródłowe pracy

Tekst tej pracy napisany został w języku `markdown`. Pliki źródłowe umieszczone są w katalogu `/docs` i budowane są za pomocą skryptu zamieszczonego w pliku `makefile` z wykorzystaniem programu `pandoc` i biblioteki `texlive`. Katalog `/docs/figures` zawiera statyczne pliki. Strona tytułowa napisana jest w języku `latex` i budowana jest osobno.

5 Instalacja i wdrożenie

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

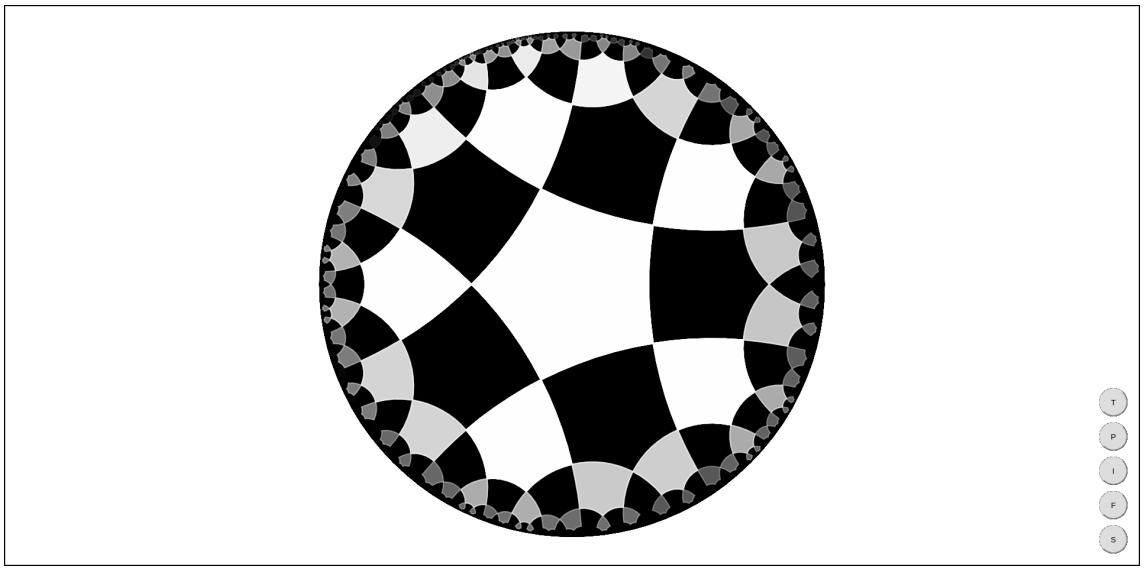
Do zbudowania aplikacji konieczny będzie menadżer pakietów `npm` w wersji przynajmniej `6.5.0` oraz środowisko uruchomieniowe języka `JavaScript` - `node.js` w wersji `10.6.0` lub nowszej. Instalacja wymaganych pakietów odbywa się poprzez wpisanie w konsoli polecenia

```
npm install
```

w katalogu głównym projektu. Skrypt umożliwiający zbudowanie aplikacji wykonuje się poleceniem:

```
npm run build
```

Skutkuje to pojawieniem się w katalogu głównym folderu `/dist` z plikami, które wraz z plikiem `index.html` składają się na gotowy program możliwy do uruchomiania w przeglądarce.



Rysunek 20: Wygląd aplikacji po uruchomieniu

5.1 Serwer deweloperski

Stworzony program wspiera tryb deweloperski, w którym bieżące zmiany w kodzie automatycznie są kompilowane do plików wynikowych. Do uruchomienia trybu deweloperskiego potrzebne

są pakiety instalowane poleceniem:

```
npm install
```

Wywołanie trybu odbywa się komendą:

```
npm run build-watch
```

5.2 Wdrożenie na serwerze WWW

Projekt można umieścić na serwerze WWW. Sposób wdrożenia zależy od posiadanego serwera. Ze względu na dużą objętość zaleca się umieszczenie projektu bez 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 Poincaré udało się osiągnąć. Wskazują na to programy demonstrujące działanie silnika. Zaprezentowana aplikacja to autorskie rozwiązanie, pozwalające na kompleksową obsługę modelu dysku Poincaré. Użycie nowoczesnych i technologi webowych takich jak silnie typowany język `Typescript`, umożliwia przyjemną pracę z silnikiem. Składają się na to również dobrze napisana warstwa renderująca grafikę, pozwalająca w sposób bezpośredni wyświetlić dowolny, wspierany interfejsami silnika figurę lub kształt.

Projekt można w przyszłości rozszerzyć o wsparcie dla obrazków, co ułatwiłoby implementację grafik Eschera. Jest to obecnie także możliwe, jednak należy w tym celu wykorzystać odbicia odpowiednich wielokątów, podobnie, jak zostało to osiągnięte w punkcie `4.3.3`, w programie `Tesselation Demo`.

7 Bibliografia

[1] Michael Hvidsten, *Exploring Geometry - Web Chapters*, 2017, Gustavus Adolphus College

[2] Martin Freiherr von Gagern, *Creation of Hyperbolic Ornaments - Algorithmic and Interactive Methods*, 2010, Technischen Universitat Munchen

[3] Joan Gómez, *Tam, gdzie proste są krzywe*, 2010, RBA Coleccionables. S. A.

[4] Bjørn Jahren, *An introduction to hyperbolic geometry*, MAT4510/3510, August 2010

[6] Marek Kordos, *O różnych geometriach*, Warszawa 1987, Wydawnictwa Alfa

[7] David Hilbert, *Foundations of Geometry*, Open Court Press, LaSalle, Illinois 1971

[8] Frank Nielsen1and and Richard Nock, *Hyperbolic Voronoi diagrams made easy*, Sony Computer Science Laboratories Inc

[9] Marshall Bern and David Eppstein, *Optimal Möbius Transformationfor Information Visualization and Meshing*, Univ. of California

[10] Marek Kordos, *Geometria Bolyaia–Łobaczewskiego*, DeltaMi, Sierpień 2018, Instytut Matematyki UW

[11] Steve Szydlik, *Hyperbolic Constructions in Geometer's Sketchpad*, December 21, 2001

[12] Douglas N. Arnold and Jonathan Rogness, *Möbius Transformations Revealed*, Minneapolis 2008, Institute for Mathematics and its Applications

[13] C. Series, *Hyperbolic Geometry*, MA448, 2008

[14] Izabela Przezdzink, *Geometria Poincaré i Kleina*, Wrocław 2010, Uniwersytet Wrocławski Wydział Matematyki i Informatyki - Instytut Matematyczny

8 Zawartość płyty CD

Do pracy dołączono płytę CD o następującej zawartości:

- kod źródłowy programu znajdujący się w folderze `/src`
- gotową, zbudowaną w katalogu `/dist` aplikację
- katalog `/docs` zawierający kod źródłowy tej pracy
- plik w formacie `pdf` zawierający pracę