Uniwersytet Mikołaja Kopernika w Toruniu
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Instytut Fizyki

Igor Kantorski
nr albumu: 258955

Praca magisterska
na kierunku fizyka techniczna

Obliczenia i symulacje biegu promieni świetlnych w kryształach scyntylacyjnych

Opiekun pracy dyplomowej
dr hab. Winicjusz Drozdowski, prof. UMK
Instytut Fizyki

Toruń 2017

Pracę przyjmuję i akceptuję

_____
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

_____
data i podpis pracownika dziekanatu

Spis treści

1. Wstęp ... 5

2. Algorytmy i metody ... **8**

2.1. Triangulacja powierzchni parametrycznych ... 10

2.2. Przecięcie prosta-trójkąt ... 11

2.3. Przecięcie płaszczyzna-trójkąt ... 14

2.4. Twierdzenie o punkcie wewnątrz wielokąta ... 16

2.5. Odbicie promienia od płaszczyzny ... 19

2.6. Rozkład jednostajny na sferze ... 21

2.7. Śledzenie promieni ... 24

3. Implementacja ... 27

3.1. Struktura kodu ... 28

3.2. Backend ... 28

3.2.1. Klasa `class DifferentialManifold(object)` ... 28

3.2.2. Klasa `class SurfaceModel(object)` ... 30

3.2.3. Klasa `class Ray(object)` ... 33

3.2.4. Funkcja `primitives(prim, params)` ... 35

3.2.5. Moduł `ctrace` ... 36

3.3. Frontend ... 42

3.3.1. Funkcja `unique_rows(arr)` ... 43

3.3.2. Funkcja `seedDirections(Npoints)` ... 44

3.3.3. Funkcja `meshDirections(Npoints)` ... 44

3.3.4. Funkcja `jitter(arr, size = 0.01)` ... 44

3.3.5. Funkcja `meshVolume(model, N)` ... 45

3.3.6. Funkcja `zeroPoly(model, zero = 0)` ... 48

3.3.7. Funkcja `axisEqual3D(ax)` ... 49

3.3.8. Klasa `class Scintillator(object)` ... 49

3.3.9. Funkcja `plotSurf(model, axes = False, z0 = True, color = 'g', alpha = 1)` ... 53

3.3.10. Funkcja `sampleTrace(model, pos, dirs = None, ax = None, fig = None, maxRecursion = 15)` ... 54

4. Symulacje i obliczenia ... 56

4.1. Pół-sfera 56
4.1.1. Przebieg trajektorii 56
4.1.2. Mapa intensywności 58
4.1.3. Obserwowana wydajność scyntylacji 60
4.2. Pół-torus 62
4.2.1. Przebieg trajektorii 62
4.2.2. Mapa intensywności 64
4.2.3. Obserwowana wydajność scyntylacji 65
4.3. Porównanie wydajności kryształów 67
5. Podsumowanie 69
6. Bibliografia 70

1. Wstęp

Pośród nauk przyrodniczych fizyka pełni rolę tej najbardziej pierwotnej, do której każda inna powinna się odwoływać. To z fizyki wyewoluowały takie nauki jak chemia kwantowa, informatyka czy biologia. Wszystkie one stosują metodę badawczą do opisu prawideł rządzących otaczającym nas światem. Jeżeli nazwać matematykę królową nauk to fizyka z pewnością zasługuję na miano matki wszelkiej nauki. Kiedy wyżej wymienione dziedziny samodzielnie rozrosły się na bardzo wielką skalę i duży wpływ na sam kierunek, w którym się udały miał utylitaryzm, fizyka pozostawała niezmienna, wciąż za główny cel mając poznanie. Mimo tego i w obrębie samej fizyki pojawiła się zdumiewająca różnorodność, czy to podejść, czy elementów przyrody, które opisuje. Dziś stwierdzić, że ktoś zajmuje się fizyką, jest tak samo niejednoznaczne jak powiedzieć, że jest się w średnim wieku. I tak mamy: termodynamikę czarnych dziur, optykę nieliniową, kwantową teorię pola, fizykę materiałów sypkich, materiałów laserowych. Można by w ten sposób wymieniać bardzo długo. Praca, której literały przegląda właśnie szanowny czytelnik, poświęcona jest zagadnieniu fizyki materiałowej – procesowi scyntylacji. Nie należy jednak sądzić, że jest ona pracą doświadczalną, choć takie najczęściej wychodzą spod pióra fizyków materiałowych. Zagadnienie scyntylacji zostało powzięte do opisu w sposób teoretyczny.

Aby postawić problem badawczy rozpatrywany w tej pracy, najpierw należy wyjaśnić czym jest scyntylacja. Scyntylacja jest pojęciem fizycznym, związanym ściśle z optyką i teorią emisji w układach kwantowych. Aby rozważać ten termin, należy najpierw wspomnieć nieco o promieniowaniu jonizującym. Jest ono szczególnym rodzajem promieniowania i generalnie terminem tym określa się każdy kwant promieniowania, który może, w oddziaływaniu z ośrodkiem materialnym, wywołać jego częściową jonizacje, a więc zmianę stanu elektronowego. To, czy dany rodzaj radiacji zostanie zakwalifikowany do tej grupy, zależy poniekąd od samego ośrodka (a więc sytuacji fizycznej) przez jaki zostaje przepuszczona, ale można generalnie stwierdzić, jakie kwanty zazwyczaj powodują jonizację materii. I tak, wyróżnia się promieniowanie, które jonizuje materię pośrednio, tj. poprzez akt wybicia elektronu w sposób komptonowski, tudzież w myśl zjawiska fotoelektrycznego. Taki sposób jonizacji charakterystyczny jest dla cząstek nienaładowanych (neutrony, fotony gamma lub rentgenowskie). Druga grupą do której można zakwalifikować promieniowanie ze względu na sposób w jaki oddziałuje, jest jonizacja bezpośrednia. Do tej grupy należą zazwyczaj cząstki obdarzone ładunkiem elektrycznym i w ich przypadku sam akt przebywania w obszarze ośrodka powoduje jonizacje, ale mogą także powodować lokalną zmianę gęstości ładunku – poprzez przebywanie w pobliżu innego pola elektromagnetycznego. Co do pojęcia scyntylacji – jeżeli promieniowanie jonizujące materię będzie przyczyną wywołania błysku świetlnego (emisji fotonu) to akt ten nazywamy właśnie scyntylacją. W myśl tego, scyntylatorem może być dowolny twór materialny. W praktyce zjawisko to można badać tylko w specyficznych obiektach, w których scyntylacja jest procesem zauważalnym i istotnym. Takie obiekty nazywane są scyntylatorami. Generalnie zarówno gaz, ciecz jak i ciało stałe mogą wykazywać własności scyntylacyjne i scyntylacja we wszystkich tych stanach skupienia znalazła swoje zastosowanie. Praca ta skupia się na opisie pewnej wielkości charakteryzującej scyntylatory, na którą wpływ ma ich kształt –

właściwość, która jest dobrze określona jedynie dla ciał stałych. Scyntylacja jest w swej istocie procesem niezwykle skomplikowanym, jej fizyczne podstawy są aktualnym tematem badań naukowych o wysokim stopniu zaawansowania, czy to pod względem teoretycznym, czy doświadczalnym [1,2]. W scyntylatorach stałych i nieorganicznych przyjęło się wyróżniać trzy fazy procesu scyntylacji – prowadzące od absorpcji kwantu jonizującego do emisji fotonu. Pierwszą z nich jest konwersja energii niesionej przez kwant jonizujący do formy energii, która może być na pewną chwilę zmagazynowana w krysztale. Tutaj kwant może zostać zaabsorbowany przez kryształ: wybić elektron z sieci (zjawisko Comptona, efekt fotoelektryczny) lub w wyniku oddziaływania z otoczeniem rozpaść się parę elektron – pozyton. Drugim etapem scyntylacji jest transport energii zmagazynowanej w krysztale. Generalnie nośnikiem energii w kryształach są elektrony lub kwazicząstki - ekscytony. Powstają one w procesie konwersji. Fizyka problemu pokazuje, że najbardziej efektywna scyntylacja obserwowana jest gdy kryształ nie jest idealny, czyli jest domieszkowany. Atomy domieszki w żargonie scyntylacyjnym nazywa się centrami luminescencji. Na etapie transportu nośnik przekazuje energię do centrów luminescencji, gdzie następuje ostatni i jednocześnie najlepiej poznany etap – luminescencja. Luminescencja jest procesem w którym energia zmagazynowana w krysztale zostaje wypromieniowana w postaci fotonu o mniejszej energii niż jonizująca cząstka. Energia ta jest zazwyczaj mniejsza nie tylko z powodu strat, ale z samej natury procesu, dlatego scyntylatory często przedstawia się jako materiały, które transformują kwanty wysokoenergetyczne (np. kwanty gamma) na wiele niskoenergetycznych fotonów.

W niniejszej pracy rozważać będziemy problem zachowania się światła w kryształach scyntylacyjnych o różnych geometriach. Ograniczeniem, które kładziemy na ową geometrię, będzie żądanie, by brzeg kryształu był kształtem gładkim, który da się przedstawić przez parametryzację. Rozważania te mają ostatecznie prowadzić do większego zrozumienia zależności między geometrią kryształu a jego *obserwowaną wydajnością scyntylacji*. Podane przed chwilą pojęcie jest jedną z charakterystyk kryształów scyntylacyjnych. W roku 1998 Dujardin i inni [4], w swojej pracy po raz pierwszy poruszyli problem zależności mierzonej wydajności kryształu scyntylacyjnego od jego rozmiarów. Zauważyli oni, że mierzona wydajność scyntylacji mocno zależy od wysokości próbki, a od wymiarów horyzontalnych nie zależy wcale, lub zależy w bardzo małym stopniu. Pierwszymi, którzy podjęli próbę wyjaśnienia tego zjawiska byli Wojtowicz wraz ze współpracownikami. W roku 2005 na konferencji SCINT na Krymie [3] przedstawili bardzo prosty model, nazywany modelem dwóch promieni (w skrócie – modelem 2R). W modelu tym założyli, że kryształ będzie charakteryzowała tylko jego wysokość (kryształ jest linią prostą), co znajduje uzasadnienie nie tylko w pracach Dujardina [4,5]. Błysk świetlny o początkowym natężeniu LY_0 który powstaje w takim krysztale na jakiejś wysokości y_0 może podróżować albo w dół, albo w górę. Jeżeli podróżuje w górę to odbija się i także dociera do okienka fotopowielacza. Droga, którą w ten sposób pokonuje determinuje straty w jego natężeniu, a straty te dane są prawem Lamberta-Beera. Wyjściowe natężenie jest uśredniane po każdym z możliwych położeń y_0 i po kierunku góra-dół. Efektem tego jest wzór z modelu dwóch promieni:

$$LY(H) = \frac{LY_0}{2H}\left(\int_0^H dy_0\left(e^{-\alpha(2H-y_0)}\right) + \int_0^H dy_0(e^{-\alpha y_0})\right) = LY_0\frac{1-e^{-2\alpha H}}{2\alpha H}$$

Model ten, mimo swojej prostoty, niezwykle dobrze tłumaczył większość obserwacji [3].

W roku 2015 autor niniejszej pracy opisał w swojej poprzedniej pracy dyplomowej model o wiele bardziej rozbudowany [6]. Bazował on wciąż na tej samej koncepcji – uśredniania natężenia błysku po jego położeniu początkowym i kierunkach początkowych. Brał jednak pod uwagę jednak to, że kryształ może być trójwymiarowy. Model ten, choć skomplikowany, doskonale zgadzał się z doświadczeniem, nie tylko pod względem przewidywania obserwowanej wydajności scyntylacji[7]. Wyniki bliższe doświadczeniu dawał także z przewidywaniem współczynnika absorbcji kryształu, czyli tam, gdzie model 2R zdecydowanie nie dawał sobie rady. W pracy tej przedstawiono jednak bardzo ogólny wzór, który będziemy tu wykorzystywać:

$$LY(\Omega) = \frac{LY_0}{4\pi|\Omega|}\int_\Omega d\mathrm{x}_0 dy_0 dz_0\left(\int_0^{2\pi} d\varphi\left(\int_0^{\pi} d\vartheta\left(|\sin(\vartheta)|R^{n(x_0,y_0,z_0,\varphi,\vartheta)}e^{-\alpha r(x_0,y_0,z_0,\varphi,\vartheta)}\right)\right)\right)$$

Gdzie $|\Omega| = \int_\Omega d\mathrm{x}_0 dy_0 dz_0$

Wzór ten obowiązuje dla kryształu o *dowolnej* geometrii, na którą ograniczenie zawarte jest w obszarze Ω. Widzimy w nim uśrednianie po wszystkich położeniach: $(\mathrm{x}_0, y_0, z_0) \in \Omega$ oraz po wszystkich kierunkach (ϑ, φ), wraz z wagą wynikającą z jakobianu - $\sin(\vartheta)$. Oprócz tego, wzór ten wprowadza obecność stałego współczynnika odbicia, który mówi o tym, ile światła pozostaje we wiązce po odbiciu od ściany kryształu. Wielkość ta podnoszona jest do potęgi $n(x_0, y_0, z_0, \varphi, \vartheta)$, która to jest liczbą odbić jakiej doświadczy wiązka zanim dotrze do detektora (okienka fotopowielacza). Następnym całkowanym wyrażeniem jest efekt osłabienia wiązki na drodze $r(x_0, y_0, z_0, \varphi, \vartheta)$, która jest drogą, jaką pokona wiązka zanim dotrze do płaszczyzny okienka fotopowielacza. Liczba odbić i droga zależą od geometrii kryształu i położeń początkowych. Wystarczy więc znaleźć tę zależność, i zastosować uśrednianie, by mieć zależność na obserwowaną wydajność scyntylacji. Właśnie tym zagadnieniem będziemy się zajmować w niniejszej pracy. Aby dokonać tego dla dowolnej geometrii danej parametrycznie, użyjemy technik numerycznych, znanych pod nazwą *śledzenia promieni*. Nazwa wybranej grupy algorytmów jest nieprzypadkowa, gdyż właśnie ze śledzeniem promieni mamy w tym zagadnieniu głównie do czynienia. Wspomniane numeryczne narzędzia stworzymy samodzielnie, wykorzystując możliwości jakie dają nam współczesne języki programowania.

2. Algorytmy i metody

Jak zostało zaznaczone w poprzednim rozdziale, celem niniejszej pracy jest koncepcyjne stworzenie i implementacja narządzi pozwalających na opis zarówno samej wydajności scyntylacji, jak i wielkości pobocznych, oraz na wizualizacje modelowego przebiegu promieni wewnątrz kryształów scyntylacyjnych o zadanej geometrii. Zanim sformułujemy zapotrzebowanie na algorytmy[8], musimy zdefiniować i opisać obszar w którym będziemy się poruszać.

Przede wszystkim, uściślijmy pojęcie *zadanej geometrii*. W pracach naukowych dotyczących zależności wydajności scyntylacji od kształtu kryształu zazwyczaj dyskutuje się kryształy, których brzeg jest rozmaitością topologiczną, a dokładniej – wielościanem. Zawężimy nieco pole poszukiwań, i w tej pracy dyskutować będziemy kryształy, których fragment brzegu będzie powierzchnią dwuwymiarową zadaną parametrycznie. Będziemy także posługiwać się pojęciem *nietrywialnego brzegu kryształu* i aby rozwiać potencjalne wątpliwości, definiujemy je jako ten fragment brzegu kryształu, który nie jest wspólny z płaszczyzną okienka fotopowielacza. W świetle powyższego, możemy powiedzieć, że:

Brzegiem kryształu ∂V nazwiemy powierzchnię zamkniętą, taką, że:

- $\vec{S}: \mathbb{R} \times \mathbb{R} \to \mathbb{R}^3$
- $\boldsymbol{S} = \vec{S}([u_{min}, u_{max}], [v_{min}, v_{max}]) \subset \mathbb{R}^3$,
- $\boldsymbol{\Gamma} = \boldsymbol{S} \cap \{[x, y, z]: z = 0\}$ jest *krzywą Jordana*,
- $\boldsymbol{Z}$ jest obszarem *wewnątrz* krzywej $\boldsymbol{\Gamma}$
- $\partial V = \boldsymbol{S} \cup \boldsymbol{Z}$,

Aby uniknąć zbędnych dyskusji na temat powyższych definicji, będziemy przyjmować także, że $\vec{S}(u, v)$ jest *dyfeomorfizmem*. Jednocześnie w powyższych wyrażeniach zdefiniowaliśmy także inne zbiory. $\boldsymbol{S}$ gra rolę *nietrywialnego brzegu kryształu* i to on jest potencjalnie najbardziej interesujący. $\{[x, y, z]: z = 0\}$ jest płaszczyzną okienka fotopowielacza, widzimy więc, że w samej definicji przyjęliśmy konwencję, że płaszczyzna $z = 0$ jest płaszczyzną okienka. **Konwencję tą będziemy stosować w całej pracy**, bez utraty jakiejkolwiek ogólności. Stosując taką definicję, jesteśmy więc zobowiązani dostarczyć takiej parametryzacji nietrywialnego brzegu kryształu, by przecinał on się z płaszczyzną $z = 0$ tworząc krzywą zamkniętą (silniej – krzywą Jordana). Analizując definicję brzegu kryształu okazuje się, że jest on dobrze zdefiniowany, gdy tylko dostarczymy żądanej parametryzacji $\vec{S}(u, v)$.

Powinno więc być już jasne, że cały kryształ reprezentujemy podając jego nietrywialny brzeg. W dalszym ciągu pracy, będziemy wymiennie traktować pojęcia $\boldsymbol{S}$ i $\vec{S}(u, v)$, z naciskiem na większą istotność samej parametryzacji $\vec{S}(u, v)$. Nietrywialny brzeg kryształu będzie podstawowym wejściem do całego algorytmu.

Bogatsi o dość ścisłą definicję wejścia całego algorytmu, możemy zastanowić się, jakie kroki przedsięwziąć, by móc uzyskać wielkości potrzebne do opisu wydajności scyntylacji. W pierwszym rozdziale stwierdziliśmy, że w tym celu należy:

1. Móc podać drogę l jaką pokona promień wyemitowany z punktu $\overrightarrow{r_0} = [x_0, y_0, z_0]$ (znajdującego się *wewnątrz* kryształu) w kierunku $\overrightarrow{d_0} = [d_{01}, d_{02}, d_{03}]$, zanim dotrze do płaszczyzny $z = 0$.
2. Móc podać liczbę odbić o, jakiej doświadczy promień wyemitowany z punktu $\overrightarrow{r_0} = [x_0, y_0, z_0]$ (znajdującego się *wewnątrz* kryształu) w kierunku $\overrightarrow{d_0} = [d_{01}, d_{02}, d_{03}]$, zanim dotrze do płaszczyzny $z = 0$.

Do opisu biegu promieni w krysztale będziemy stosować zasady optyki geometrycznej, czyli głównie *prawo odbicia*. Promień wewnątrz kryształu będzie poruszał się po prostych trajektoriach, których kierunek będzie się zmieniał tylko w momencie odbicia od jego brzegu.

W tym momencie jasno widać, że każdy z takich promieni musimy *śledzić*. W tym celu wykorzystamy algorytm znany pod nazwą *Ray Tracer*. Technika którą zastosujemy, jest ściśle nazywana *forward ray tracingiem* [14]. Słowo *forward* tyczy się tego, że promienie śledzimy zgodnie z kierunkiem ich propagacji, aż dotrą do ekranu (w naszym przypadku ekranem jest okienko fotopowielacza). Warto nadmienić, iż istnieje bliźniacza technika renderowania scen trójwymiarowych, nazywana *backward ray tracingiem* [14], która różni się tym, że promienie śledzone są wstecz, a poszukuje się punktów na scenie, z którymi promienie te się zderzają. *Forward ray tracing* jest metodą o wiele bardziej kosztowną obliczeniową, jednak ma pewną główną zaletę – jest to metoda fizyczna, wynika z praw optyki geometrycznej. Oprócz tego, używana przez nas odmiana *ray tracingu* jest znana jako najdokładniejsza istniejąca metoda renderowania scen trójwymiarowych na obraz 2D. Fakt, iż jest to w istocie narzędzie do generowania obrazu ze scen 3D, możemy wykorzystać na potrzeby wizualizacji biegu promieni w kryształach. Zanim przejdziemy do opisu wszystkich wykorzystywanych algorytmów, wymieńmy podstawowe sposoby prezentacji wyników działania budowanego narzędzia:

- Będziemy w stanie stworzyć obraz wnętrza kryształu oświetlonego źródłem punktowym.
- Będziemy w stanie wykreślić trójwymiarowe wykresy trajektorii promieni wewnątrz kryształu.
- Docelowo będziemy mogli obliczyć wartości parametru β, a więc i obserwowanej wydajności scyntylacji kryształu.
- Zmieniając parametry kształtu (np. promień dla sfery) będziemy mogli wykreślić zależność wydajności od tego parametru dla określonego kształtu.

Pozostało nam jeszcze omówić sposób reprezentacji kształtu brzegu jako obiektu w przestrzeni. Aby móc zastosować *ray tracing*, musimy móc określić punkt przecięcia się promienia z powierzchnią (będącą brzegiem kryształu). Istnieją numeryczne sposoby na rozwiązanie takiego zagadnienia, jednak w ogólności sprowadzają się one do rozwiązania układu trzech równań nieliniowych na trzy niewiadome. Jest to zagadnienie skomplikowane,

numerycznie trudne i w ogólności może nie wygenerować wszystkich rozwiązań. W naszym przypadku jest to niedopuszczalne, trzeba więc poszukać innej metody.

Typowym zabiegiem rozwiązującym powyższe problemy jest triangulacja powierzchni, czyli przybliżenie jej faktycznego kształtu wieloma trójkątami. Rozpoczniemy opis algorytmów od tego zagadnienia.

2.1. Triangulacja powierzchni parametrycznych

Samej triangulacji dokonywać będziemy tylko na nietrywialnym brzegu kryształu. Jak się później okaże, reszta powierzchni, czyli *de facto* płaszczyzna (okienka PMT), nie wymaga triangulacji, gdyż jest właśnie płaską powierzchnią. Dla płaszczyzn znalezienie ich przecięcia z (pół)prostą jest wykonalne symbolicznie.

Musimy również zaznaczyć, że nie wykonujemy całej triangulacji samemu. Skorzystamy z zaimplementowanej już w Pythonie metody triangulacji powierzchni płaskich, żeby wygenerować zbiór trójkątów w przestrzeni trójwymiarowej.

1. Wygenerujmy płaską siatkę (zbiór punktów) na przestrzeni parametrów:

Nietrywialny brzeg dany jest zbiorem:

$$\boldsymbol{S} = \{\mathbb{R}^3 \ni \vec{S}(u,v) : u \in [u_{min}, u_{max}], v \in [v_{min}, v_{max}]\}$$

Stwórzmy następujący zbiór:

$$M_{flat} = \left\{\left(u_{min} + n\frac{u_{max} - u_{min}}{N}, v_{min} + m\frac{v_{max} - v_{min}}{N}\right) : n, m \in \{0,1,\dots,N\}\right\}$$

Teraz M_{flat} jest listą punktów $(u,v) \in \mathbb{R}^2$, które wypełniają zakres podany w definicji $\boldsymbol{S}$. Liczba wygenerowanych punktów jest równa $(N+1)^2$. Gęstość siatki rośnie wraz z N.

2. Na tak przygotowanym zbiorze punktów dokonujemy triangulacji płaskiej, metodą Delanuay'a[8]:

Triangulacja Delanuay'a działa w ten sposób, że z danego zbioru punktów M_{flat} tworzy takie trójkąty (trójki punktów), że nie istnieje żaden punkt ze zbioru M_{flat} będący wewnątrz okręgu opisanego na danym trójkącie. Można pokazać, że taka metoda maksymalizuje najmniejszy z kątów w każdym z trójkątów (na płaszczyźnie). Nie zajmujemy się implementacją algorytmu Delanuay'a, gdyż jest ona dostępna w jednej ze standardowych bibliotek Pythona.

Triangulacja M_{flat} daje w wyniku uporządkowany zbiór trójek, będących wierzchołkami trójkątów:

$$M_{tri} = \{(\boldsymbol{p_0^0}, \boldsymbol{p_0^1}, \boldsymbol{p_0^2}), \dots, (\boldsymbol{p_i^0}, \boldsymbol{p_i^1}, \boldsymbol{p_i^2}), \dots\} = DL(M_{flat})$$

Gdzie $\boldsymbol{p_i^j} \in M_{flat}$.

3. Każdy z trójkątów $(\boldsymbol{p_i^0}, \boldsymbol{p_i^1}, \boldsymbol{p_i^2}) \in M_{tri}$, wynosimy na powierzchnię nietrywialnego brzegu za pomocą parametryzacji $\vec{S}(u, v)$:

Tworzymy uporządkowany zbiór trójkątów dla nietrywialnego brzegu:

$$S_{tri} = \left\{\left(\vec{S}(\boldsymbol{p_0^0}), \vec{S}(\boldsymbol{p_0^1}), \vec{S}(\boldsymbol{p_0^2})\right), \dots, \left(\vec{S}(\boldsymbol{p_i^0}), \vec{S}(\boldsymbol{p_i^1}), \vec{S}(\boldsymbol{p_i^2})\right), \dots\right\}$$

S_{tri} jest więc teraz zbiorem trójek współrzędnych w $\mathbb{R}^3$ które składają się na wierzchołki trójkątów. Jako, że w zbiorze M_{flat} pokryliśmy cały zakres zmienności parametrów *u* i *v*, to możemy być pewni, że S_{tri} pokryje cały zakres zmienności $\boldsymbol{S}$.

W ten sposób dokonaliśmy triangulacji powierzchni parametrycznej. Warto zwrócić uwagę, że ten sposób zadziała tylko dla powierzchni parametrycznych, gdyż tak naprawdę właściwej triangulacji dokonaliśmy na przestrzeni jej parametrów. Następnym rozważanym algorytmem będzie znajdowanie punktu przecięcia prostej z trójkątem.

2.2. Przecięcie prosta-trójkąt

Obecnie nasza powierzchnia jest już przybliżana zbiorem trójkątów:

$$S_{tri} = \left\{(\boldsymbol{P_0^0}, \boldsymbol{P_0^1}, \boldsymbol{P_0^2}), \dots, (\boldsymbol{P_i^0}, \boldsymbol{P_i^1}, \boldsymbol{P_i^2}), \dots\right\}$$

gdzie $\boldsymbol{P_j^i} \in \mathbb{R}^3$

W procesie *ray tracingu* będziemy musieli znaleźć punkt w którym prosta przecina płaszczyznę trójkąta, oraz *rozstrzygnąć czy punkt przecięcia należy do trójkąta*. Najpierw zajmiemy się znalezieniem punktu przecięcia. W tym celu wprowadzimy konwencję opisu promienia i płaszczyzny w $\mathbb{R}^3$.

Promieniem w $\mathbb{R}^3$ będziemy nazywać funkcję:

$$\vec{r}: \mathbb{R} \to U \subset \mathbb{R}^3,$$

taką, że:

$$\vec{r}(t) = \overrightarrow{r_0} + \vec{d} \cdot t = \begin{bmatrix} r_0^0 \\ r_0^1 \\ r_0^2 \end{bmatrix} + \begin{bmatrix} d^0 \\ d^1 \\ d^2 \end{bmatrix} t, \qquad t > 0$$

Płaszczyzną w $\mathbb{R}^3$ będziemy nazywać funkcję:

$$\vec{X}: \mathbb{R}^2 \to \mathbb{R}^3,$$

taką, że:

$$\vec{X}(u, v) = \overrightarrow{P_2} + \left(\overrightarrow{P_1} - \overrightarrow{P_2}\right) \cdot u + \left(\overrightarrow{P_0} - \overrightarrow{P_2}\right) \cdot v$$

gdzie $\overrightarrow{P_i} \in \mathbb{R}^3$ oraz $\det\left(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2}\right) \neq 0$

Widzimy, że promień jest dobrze zdefiniowany przez podanie jego *punktu początkowego* $\overrightarrow{r_0}$ oraz jego *kierunku* $\vec{d}$. Gdy kierunek promienia jest znormalizowany, to zmienna t jest jednocześnie długością wektora kierunku (w sensie euklidesowym).

Z kolei definicja płaszczyzny jest kompletna, gdy podamy (niezdegenerowany) zestaw trzech punktów do niej należących. Formalnie, taka płaszczyzna jest *podprzestrzenią afiniczną* dla $\mathbb{R}^3$. Wtedy wektory: $\left(\overrightarrow{P_1} - \overrightarrow{P_2}\right), \left(\overrightarrow{P_0} - \overrightarrow{P_2}\right)$ wyznaczają bazę jej *przestrzeni stowarzyszonej*.

Dla nas jest jednak najistotniejsze, że wymienione wyżej punkty mogą być traktowane jako współrzędne wierzchołków trójkątów.

Zauważmy bardzo istotny fakt, mianowicie takie $\vec{X}$ jest *bijekcją*, czyli każdemu punktowi z płaszczyzny (u, v) możemy przyporządkować jednoznacznie $\vec{X}(u, v)$ i odwrotnie, wyczerpując zakres zmienności parametrów. Wykorzystamy ten fakt za chwilę.

Aby znaleźć punkt przecięcia promienia z płaszczyzną, należy rozwiązać układ równań:

$$\vec{X}(u, v) = \vec{r}(t)$$
$$\overrightarrow{P_2} + \left(\overrightarrow{P_1} - \overrightarrow{P_2}\right) \cdot u + \left(\overrightarrow{P_0} - \overrightarrow{P_2}\right) \cdot v = \overrightarrow{r_0} + \vec{d} \cdot t$$
$$\left(\overrightarrow{P_1} - \overrightarrow{P_2}\right) \cdot u + \left(\overrightarrow{P_0} - \overrightarrow{P_2}\right) \cdot v - \vec{d} \cdot t = \overrightarrow{r_0} - \overrightarrow{P_2}$$

Jak widzimy, jest to układ równań liniowych na u, v, t. Możemy go rozwiązać wykorzystując metodę wyznaczników. W tym celu rozpiszmy go jako układ równań skalarnych:

$$\begin{bmatrix} (P_1^0 - P_2^0) & (P_0^0 - P_2^0) & -d^0 \\ (P_1^1 - P_2^1) & (P_0^1 - P_2^1) & -d^1 \\ (P_1^2 - P_2^2) & (P_0^2 - P_2^2) & -d^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \begin{bmatrix} r_0^0 - P_2^0 \\ r_0^1 - P_2^1 \\ r_0^2 - P_2^2 \end{bmatrix}$$

Wobec tego, jego rozwiązanie u_c, v_c, t_c:

$$u_c = \frac{\begin{vmatrix} r_0^0 - P_2^0 & (P_0^0 - P_2^0) & -d^0 \\ r_0^1 - P_2^1 & (P_0^1 - P_2^1) & -d^1 \\ r_0^2 - P_2^2 & (P_0^2 - P_2^2) & -d^2 \end{vmatrix}}{\begin{vmatrix} (P_1^0 - P_2^0) & (P_0^0 - P_2^0) & -d^0 \\ (P_1^1 - P_2^1) & (P_0^1 - P_2^1) & -d^1 \\ (P_1^2 - P_2^2) & (P_0^2 - P_2^2) & -d^2 \end{vmatrix}}$$

$$v_c = \frac{\begin{vmatrix} (P_1^0 - P_2^0) & r_0^0 - P_2^0 & -d^0 \\ (P_1^1 - P_2^1) & r_0^1 - P_2^1 & -d^1 \\ (P_1^2 - P_2^2) & r_0^2 - P_2^2 & -d^2 \end{vmatrix}}{\begin{vmatrix} (P_1^0 - P_2^0) & (P_0^0 - P_2^0) & -d^0 \\ (P_1^1 - P_2^1) & (P_0^1 - P_2^1) & -d^1 \\ (P_1^2 - P_2^2) & (P_0^2 - P_2^2) & -d^2 \end{vmatrix}}$$

$$t_c = \frac{\begin{vmatrix} (P_1^0 - P_2^0) & (P_0^0 - P_2^0) & r_0^0 - P_2^0 \\ (P_1^1 - P_2^1) & (P_0^1 - P_2^1) & r_0^1 - P_2^1 \\ (P_1^2 - P_2^2) & (P_0^2 - P_2^2) & r_0^2 - P_2^2 \end{vmatrix}}{\begin{vmatrix} (P_1^0 - P_2^0) & (P_0^0 - P_2^0) & -d^0 \\ (P_1^1 - P_2^1) & (P_0^1 - P_2^1) & -d^1 \\ (P_1^2 - P_2^2) & (P_0^2 - P_2^2) & -d^2 \end{vmatrix}}$$

które istnieje o ile wyznacznik macierzy układu jest różny od zera, co z kolei jest zapewnione przez definicję powierzchni.

Pozostało nam rozstrzygnięcie, czy takie rozwiązanie leży wewnątrz trójkąta o wierzchołkach $(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2})$. Skorzystajmy z faktu, że $\vec{X}$ jest *bijekcją*. W związku z tym dowolny punkt na obrazie $\vec{X}$, ma swoje odpowiedniki na przestrzeni argumentów (u, v). Zamiast sprawdzać w zawiły sposób czy rozwiązanie należy do trójkąta w przestrzeni trójwymiarowej, sprowadzimy problem do zwykłej przestrzeni dwuwymiarowej, w dodatku euklidesowej, jaką tworzą wymienione argumenty (w końcu płaszczyzna to płaszczyzna, nie ma znaczenia jak ją reprezentujemy, tak długo jak długo istnieje wzajemna jednoznaczność między reprezentacjami). Wiemy, że wierzchołki trójkąta to $(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2})$, zapytajmy, jakie mają współrzędne w bazie (u, v)? Sprawdźmy, podstawiając za $\vec{X}(u, v)$, kolejno: $\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2}$.

$$\overrightarrow{P_0} = \vec{X}(u, v) = \overrightarrow{P_2} + (\overrightarrow{P_1} - \overrightarrow{P_2}) \cdot u + (\overrightarrow{P_0} - \overrightarrow{P_2}) \cdot v$$

Widzimy tutaj, że jest to potencjalnie sprzeczny układ (dwie niewiadome, trzy równania), jednak liniową zależność dwóch z nich zapewnia nam fakt, że punkt $\overrightarrow{P_0}$ na pewno leży na tej płaszczyźnie. Gdyby podstawić za $(u, v) \rightarrow (0,1)$, otrzymujemy:

$$\vec{X}(0,1) = \overrightarrow{P_2} + (\overrightarrow{P_1} - \overrightarrow{P_2}) \cdot 0 + (\overrightarrow{P_0} - \overrightarrow{P_2}) \cdot 1 = \overrightarrow{P_0}$$

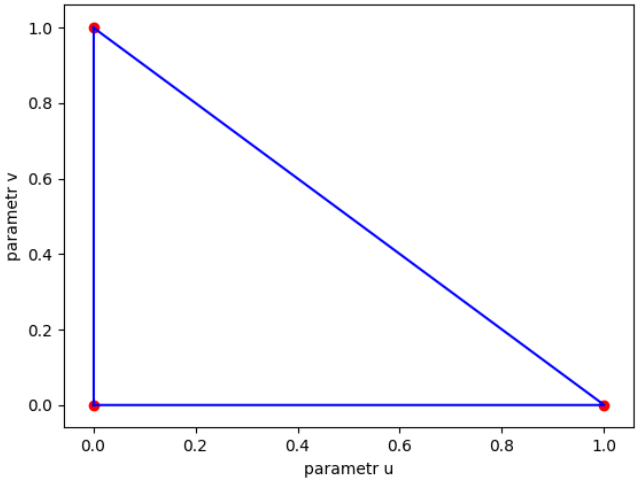
Podobnie dla dwóch następnych wierzchołków:

$$\vec{X}(1,0) = \overrightarrow{P_2} + (\overrightarrow{P_1} - \overrightarrow{P_2}) \cdot 1 + (\overrightarrow{P_0} - \overrightarrow{P_2}) \cdot 0 = \overrightarrow{P_1}$$
$$\vec{X}(0,0) = \overrightarrow{P_2} + (\overrightarrow{P_1} - \overrightarrow{P_2}) \cdot 0 + (\overrightarrow{P_0} - \overrightarrow{P_2}) \cdot 0 = \overrightarrow{P_2}$$

Mamy więc wzajemną odpowiedniość:

$$\overrightarrow{P_0} \leftrightarrow (u, v) = (0,1)$$
$$\overrightarrow{P_1} \leftrightarrow (u, v) = (1,0)$$
$$\overrightarrow{P_2} \leftrightarrow (u, v) = (0,0)$$

Widzimy, że w bazie (u, v) taki trójkąt jest równoramienny. Rysunek 2.1 przedstawia umiejscowienie wierzchołków $(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2})$ w bazie (u, v), oraz odcinki które reprezentują jego ramiona. Z samego rysunku



Rys. 2.1. Krawędzie i wierzchołki trójkąta $(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2})$ w układzie współrzędnych (u, v).

wynika, że punkt (u_c, v_c) leży wewnątrz trójkąta, gdy tylko:

- $0 \leq u_c \leq 1$
- $0 \leq v_c \leq 1$
- $u_c + v_c \leq 1$

Oprócz tego, aby być pewnym że znaleźliśmy właściwe rozwiązanie, należy sprawdzić, czy:

- $t_c > 0$

Wtedy i tylko wtedy promień $\vec{r}(t)$ przecina trójkąt $(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2})$ w punkcie:

$$\overrightarrow{X_c} = \overrightarrow{P_2} + (\overrightarrow{P_1} - \overrightarrow{P_2}) \cdot u_c + (\overrightarrow{P_0} - \overrightarrow{P_2}) \cdot v_c$$

Mamy więc sposób na sprawdzenie czy promień przecina się z trójkątem i w jakim punkcie następuje przecięcie. Będzie to nasza podstawowa metoda przy implementacji *ray tracera*. Pozostając w tonie znajdowania przecięć, następną dyskutowaną metodą będzie znajdowanie przecięcia płaszczyzny z trójkątem.

2.3. Przecięcie płaszczyzna-trójkąt

Znalezienie przecięcia płaszczyzny z trójkątem, którego efektem ma być odcinek, nie jest potrzebne do implementacji *ray tracingu*, natomiast będzie potrzebne przy próbie całkowania po objętości kryształu. W dodatku jedyne przecięcie, jakie będzie wykorzystywane to przecięcie trójkąta płaszczyzną *z = const*, dlatego tylko taki przypadek rozważymy.

Niech dana będzie płaszczyzna $\overrightarrow{X_\iota}$ opisana równaniem $z = z_i$, wtedy jej postać parametryczna:

$$\overrightarrow{X_\iota}(x, y) = z_i \cdot \hat{z} + \hat{x} \cdot x + \hat{y} \cdot y = \begin{bmatrix} x \\ y \\ z_i \end{bmatrix}$$

oraz niech dany będzie trójkąt $(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2})$, wtedy jego płaszczyzna:

$$\vec{X}_{tri}(u, v) = \overrightarrow{P_2} + (\overrightarrow{P_1} - \overrightarrow{P_2}) \cdot u + (\overrightarrow{P_0} - \overrightarrow{P_2}) \cdot v$$

Ich przecięcie znajdujemy rozwiązując układ równań:

$$\overrightarrow{X_\iota}(x, y) = \vec{X}_{tri}(u, v)$$

I oczekujemy znaleźć za jego pomocą funkcję $x = x(v)$, wobec tego układ można zapisać następująco:

$$\overrightarrow{X_\iota}(x(v), y) = \vec{X}_{tri}(u, v)$$

który ma już rozwiązanie. Praktycznie jego rozwiązanie jest proste. Zapiszmy równania skalarnie:

$$\begin{cases}(P_1^0 - P_2^0)\cdot u + (P_0^0 - P_2^0)\cdot v + P_2^0 = x \\ (P_1^1 - P_2^1)\cdot u + (P_0^1 - P_2^1)\cdot v + P_2^1 = y \\ (P_1^2 - P_2^2)\cdot u + (P_0^2 - P_2^2)\cdot v + P_2^2 = z_i\end{cases}$$

Z ostatniego równania otrzymujemy zależność $u(v)$:

$$u(v) = \frac{z_i - P_2^2}{(P_1^2 - P_2^2)} - \frac{P_0^2 - P_2^2}{(P_1^2 - P_2^2)}v$$

Skąd otrzymujemy (wstawiając do pierwszego lub drugiego równania):

$$x(v) = (P_1^0 - P_2^0)\cdot\left(\frac{z_i - P_2^2}{(P_1^2 - P_2^2)} - \frac{P_0^2 - P_2^2}{(P_1^2 - P_2^2)}v\right) + (P_0^0 - P_2^0)\cdot v + P_2^0$$

$$y(v) = (P_1^1 - P_2^1)\cdot\left(\frac{z_i - P_2^2}{(P_1^2 - P_2^2)} - \frac{P_0^2 - P_2^2}{(P_1^2 - P_2^2)}v\right) + (P_0^1 - P_2^1)\cdot v + P_2^1$$

Proste opisane równaniami $\big(x(v), y(v)\big), u(v)$ są tymi samymi prostymi, zapisanymi w różnej bazie. Docelowo chcemy znaleźć dwa punkty we współrzędnych (x, y), które będą tworzyć odcinek. Punkty te mają być punktami przecięcia płaszczyzny $\overrightarrow{X_i}(x, y)$ z trójkątem $\big(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2}\big)$. Aby je znaleźć, problem rozwiążemy w bazie (u, v), w której mamy już prostą $u(v)$.

Należy rozważyć przecięcia prostej $u(v)$ z prostymi tworzącymi trójkąt w bazie (u, v), są to proste: $u = 0, v = 0, v = 1 - u$.

1) Rozważmy przecięcie $u(v)$ z $u = 0$, mamy wtedy:

$$\frac{z_i - P_2^2}{(P_1^2 - P_2^2)} - \frac{P_0^2 - P_2^2}{(P_1^2 - P_2^2)}v_{c1} = 0$$

$$v_{c1} = \frac{z_i - P_2^2}{P_0^2 - P_2^2}$$

   Jeżeli takie $v_{c1} \in [0,1]$, to znaleźliśmy jeden z punktów przecięcia

2) Rozważmy przecięcie $u(v)$ z $v = 0$, mamy wtedy:

$$u_{c2} = \frac{z_i - P_2^2}{(P_1^2 - P_2^2)}$$

   Jeżeli takie $u_{c2} \in [0,1]$, to znaleźliśmy jeden z punktów przecięcia

3) Rozważmy przecięcie $u(v)$ z $v = 1 - u$, mamy wtedy:

$$v_{c3} = 1 - \left(\frac{z_i - P_2^2}{(P_1^2 - P_2^2)} - \frac{P_0^2 - P_2^2}{(P_1^2 - P_2^2)}v_{c3}\right)$$

$$v_{c3}\left(1-\frac{P_0^2-P_2^2}{(P_1^2-P_2^2)}\right)=1-\frac{z_i-P_2^2}{(P_1^2-P_2^2)}$$

$$v_{c3}=\frac{1-\frac{z_i-P_2^2}{(P_1^2-P_2^2)}}{1-\frac{P_0^2-P_2^2}{(P_1^2-P_2^2)}}=\frac{P_1^2-z_i}{P_1^2-P_0^2}$$

$$u_{c3}=1-\frac{P_1^2-z_i}{P_1^2-P_0^2}=\frac{z_i-P_0^2}{P_1^2-P_0^2}$$

Jeżeli takie $v_{c3} \in [0,1]$, to znaleźliśmy jeden z punktów przecięcia

Należy obliczyć wszystkie: v_{c1}, u_{c2}, v_{c3} i wykonać sprawdzenie:

Jeżeli ($v_{c1} \in [0,1] \wedge u_{c2} \in [0,1]$) to:

$$(u_1, v_1)=\left(0, \frac{z_i-P_2^2}{P_0^2-P_2^2}\right)$$
$$(u_2, v_2)=\left(\frac{z_i-P_2^2}{(P_1^2-P_2^2)}, 0\right)$$

W przeciwnym wypadku, jeżeli ($v_{c1} \in [0,1] \wedge v_{c3} \in [0,1]$) to:

$$(u_1, v_1)=\left(0, \frac{z_i-P_2^2}{P_0^2-P_2^2}\right)$$
$$(u_2, v_2)=\left(\frac{z_i-P_0^2}{P_1^2-P_0^2}, \frac{P_1^2-z_i}{P_1^2-P_0^2}\right)$$

W przeciwnym wypadku, jeżeli ($u_{c2} \in [0,1] \wedge v_{c3} \in [0,1]$) to:

$$(u_1, v_1)=\left(\frac{z_i-P_2^2}{(P_1^2-P_2^2)}, 0\right)$$

$$(u_2, v_2)=\left(\frac{z_i-P_0^2}{P_1^2-P_0^2}, \frac{P_1^2-z_i}{P_1^2-P_0^2}\right)$$

W przeciwnym wypadku, płaszczyzna nie przecina trójkąta.

Współrzędne końców odcinka w przestrzeni $\mathbb{R}^3$ wyglądają wtedy następująco:

$$(\overrightarrow{w_1}, \overrightarrow{w_2})=\left(\vec{X}_{tri}(u_1, v_1), \vec{X}_{tri}(u_2, v_2)\right)$$

Kolejnym algorytmem, jaki omówimy, będzie algorytm rozstrzygający, czy punkt na płaszczyźnie znajduje się wewnątrz danego wielokąta.

2.4. Twierdzenie o punkcie wewnątrz wielokąta

Poprzez wielokąt będziemy rozumieć zbiór odcinków, jednoznacznie reprezentowany poprzez zbiór wierzchołków tych odcinków. Od wielokąta żądamy, by był krzywą Jordana, czyli krzywą zamkniętą, która się nie przecina sama ze sobą. Wtedy też obowiązuje *twierdzenie o krzywej Jordana*:

> *Krzywa Jordana dzieli zbiór na dwa podzbiory (A i B) o wspólnym brzegu, będącym tą krzywą.*

Rozważmy teraz półprostą o początku w punkcie wewnątrz krzywej, czyli w punkcie należącym do *A*. Jako że jest to półprosta, to znajdziemy na niej taki punkt (w nieskończoności), który zawsze leży w podzbiorze B. Widać, że w takim przypadku, półprosta musiała się przecinać z brzegiem przestrzeni nieparzystą liczbę razy, a więc przecięła wielokąt nieparzystą liczbę razy.

Z drugiej strony, jeśli rozważymy półprostą o początku w punkcie na zewnątrz krzywej Jordana, czyli należącym do *B*, to również na tej półprostej zawsze istnieje punkt (w nieskończoności) który należy do *B*. Oznacza to, że tym razem półprosta musiała przeciąć wielokąt *parzystą* liczbę razy.

Możemy sformułować więc twierdzenie:

> *Punkt leży wewnątrz wielokąta wtedy i tylko wtedy gdy jakakolwiek półprosta wyprowadzona z tego punktu przecina wielokąt nieparzystą liczbę razy.*

Algorytm który będzie sprawdzał, czy punkt (x_0, y_0) leży wewnątrz wielokąta można opisać następująco:

Dla każdej pary sąsiednich punktów $(\overrightarrow{w_1}, \overrightarrow{w_2})$ należącej do zbioru końców odcinków wielokąta:

1) Wyzerować rejestr przecięć.
2) Rozwiązać równanie:

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} t = \begin{bmatrix} w_2^0 \\ w_2^1 \end{bmatrix} + \begin{bmatrix} w_1^0 - w_2^0 \\ w_1^1 - w_2^1 \end{bmatrix} t'$$

   co daje nam punkt, w którym przecięły się proste:

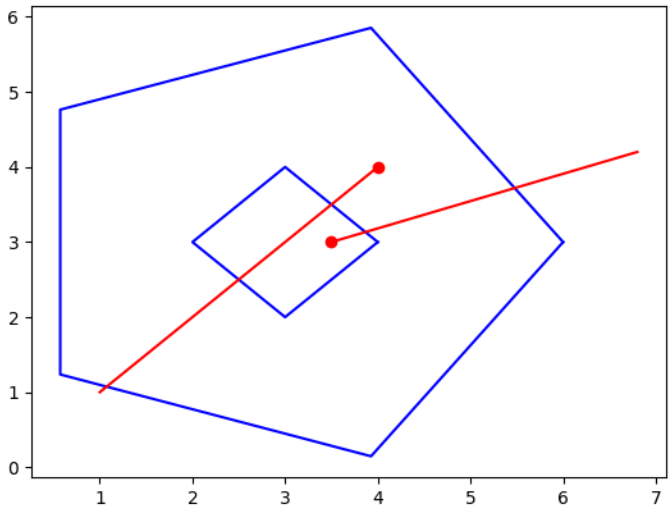
$$t_c' = -\frac{b\, w_2^0 - a\, w_2^1 - b\, x_0 + a\, y_0}{b\, w_1^0 - b\, w_2^0 - a\, w_1^1 + a\, w_2^1}$$
$$t_c = \frac{w_2^0 - x_0 + (w_1^0 - w_2^0) t_c'}{a}$$

3) Rozstrzygnąć, czy *półprosta* przecięła się z *odcinkiem*:

   Jeżeli $t_c' \in [0,1) \wedge t_c > 0$ to punkt przecięcia należy do odcinka $(\overrightarrow{w_1}, \overrightarrow{w_2})$.

4) Jeżeli punkt przecięcia należy do odcinka $(\overrightarrow{w_1}, \overrightarrow{w_2})$, zwiększyć o 1 rejestr przecięć

5) Przyjąć następny odcinek $(\overrightarrow{w_1}, \overrightarrow{w_2})$ i wrócić do punktu 2)



Rys. 2.2. Wizualizacja problemu w poruszanego w algorytmie 2.4. Czerwone kropki są punktami o których należy rozstrzygnąć czy są wewnątrz czy na zewnątrz wielokąta. Linie niebieskie reprezentują wielokąty, a linie czerwone – półproste testujące.

Jeżeli po wyczerpaniu odcinków, rejestr przecięć jest liczbą nieparzystą, to punkt (x_0, y_0) leży wewnątrz wielokąta. Jeżeli natomiast rejestr przecięć jest parzysty, to punkt (x_0, y_0) leży na zewnątrz wielokąta. Warto zwrócić uwagę, że podany algorytm działa równie dobrze, gdy dyskutujemy kilka wielokątów. Jeżeli zbiór końców odcinków $(\overrightarrow{w_1}, \overrightarrow{w_2})$ tworzy nie jedną, ale *kilka* krzywych Jordana, to podany algorytm jest w stanie przesądzić, czy punkt leży wewnątrz któregokolwiek z wielokątów. Co więcej, nawet gdyby zbiór końców odcinków zawierał dwie krzywe Jordana, z których jedna *w całości* leży wewnątrz drugiej (np. okrąg w okręgu), to algorytm da odpowiedź twierdzącą (*'punkt jest wewnątrz'*) gdy punkt leży między dwiema tymi krzywymi, a przeczącą (*'punkt jest na zewnątrz'*) gdy punkt leży „wewnątrz obydwu krzywych" albo całkowicie na zewnątrz obydwu. Sytuację tę obrazuje rysunek 2.2. Widzimy na nim, że algorytm zaklasyfikuje punkt między czworokątem a pięciokątem, jako punkt *'wewnątrz'* (trzy przecięcia, nieparzysta liczba), natomiast punkt wewnątrz czworokąta uznany zostanie za punkt *'na zewnątrz'* (dwa przecięcia, parzysta liczba). Zachowanie takie jest w naszym przypadku pożądane z jednego konkretnego powodu. Wyobraźmy sobie, że nietrywialnym brzegiem kryształu ustalimy torus. Jednym ze sposobów ustalenia, czy punkt leży wewnątrz kryształu, jest przecięcie nietrywialnego brzegu płaszczyzną $z = const$ (na takiej wysokości, by ta płaszczyzna zawierała rozważany punkt) i zapisanie krzywych, jakie przy takim przecięciu powstają. W przypadku torusa, mogą być to dwa okręgi, jeden wewnątrz drugiego. Wtedy wystarczy sprawdzić, podanym tutaj algorytmem, czy punkt leży 'wewnątrz', czy 'na zewnątrz' podanych krzywych (problem jest wtedy płaski, rozgrywa się na przecinającej płaszczyźnie).

Podany algorytm będzie pomocny przy numerycznym całkowaniu po objętości kryształu.

2.5. Odbicie promienia od płaszczyzny

Najbardziej podstawowym algorytmem używanym przy implementacji *ray tracingu* jest algorytm obliczający kierunek promienia odbitego od płaszczyzny. Etap ten jest tym, co czyni z *ray tracera* jedną z najdokładniejszych metod renderowania scen trójwymiarowych, gdyż to on jest fizyczną podstawą tej metody. W miejscu tym, w najprostszej postaci algorytmu,

korzystamy z prawa odbicia znanego z optyki geometrycznej. Najkrócej można je przedstawić w następującym stwierdzeniu:

Kąt między promieniem padającym a płaszczyzną jest taki sam jak kąt między promieniem odbitym a płaszczyzną.

W takiej jednak postaci jest ono mało przydatne. Przeformułujemy je na postać równań matematycznych, które będziemy mogli zaprzęgnąć w implementacji metody. Najpierw jednak określimy, jakie wielkości będziemy podawać na wejście algorytmu, a jakie chcemy przy jego pomocy otrzymać. Na wejścia będziemy podawać:

- Kierunek promienia wejściowego, reprezentowany przez wektor $\overrightarrow{r_d} \in \mathbb{R}^3$ (jest to ten sam wektor określający kierunek w parametryzacji prostej)
- Zbiór punktów $\left(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2}\right)$ określający płaszczyznę od której ma nastąpić odbicie

Natomiast na wyjściu algorytmu chcemy otrzymać:

- Kierunek promienia odbitego, reprezentowanego przez wektor $\overrightarrow{r_d}' \in \mathbb{R}^3$

Płaszczyznę jednoznacznie reprezentuje wektor do niej prostopadły. Zamiast więc posługiwać się trzema wektorami do opisu płaszczyzny (afinicznej), rozważymy jedynie jeden wektor, opisujący płaszczyznę i zapomnimy o jej umiejscowieniu w przestrzeni (nie gra to żadnej roli w dyskutowanym problemie).

Wektor prostopadły (ortogonalny) $\vec{n}$ do płaszczyzny do której należą trzy punkty: $\left(\overrightarrow{P_0}, \overrightarrow{P_1}, \overrightarrow{P_2}\right)$ wyraża się następująco:

$$\vec{n} = \left(\overrightarrow{P_0} - \overrightarrow{P_2}\right) \times \left(\overrightarrow{P_1} - \overrightarrow{P_2}\right)$$

Wektory $\left(\overrightarrow{P_0} - \overrightarrow{P_2}\right)$ i $\left(\overrightarrow{P_1} - \overrightarrow{P_2}\right)$ rozpinają przestrzeń liniową, więc ich iloczyn wektorowy jest wektorem prostopadłym do tej przestrzeni (jest prostopadły do dwóch rożnych wektorów do niej należących, co implikuje, że jest prostopadły do każdego). Oczywiście wektor taki nie musi mieć euklidesowej normy równej jedności, ale też na razie nie wymagamy od niego tej własności.

Sformułujmy teraz równania, które narzucają ograniczenia prawa odbicia na wektory: $\overrightarrow{r_d}'$ i $\overrightarrow{r_d}$. W użytej przez nas notacji, $\langle \vec{a} | \vec{b} \rangle$ oznacza iloczyn skalarny wektorów $\vec{a}$ i $\vec{b}$:

$$\begin{cases} \langle \overrightarrow{r_d} | \overrightarrow{r_d} \rangle = \left\langle \overrightarrow{r_d}' \middle| \overrightarrow{r_d}' \right\rangle \\ \langle \overrightarrow{r_d} | \vec{n} \rangle = -\left\langle \overrightarrow{r_d}' \middle| \vec{n} \right\rangle \\ \left\langle \overrightarrow{r_d} \middle| \vec{t} \right\rangle = \left\langle \overrightarrow{r_d}' \middle| \vec{t} \right\rangle \\ \langle \vec{t} | \vec{n} \rangle = 0 \end{cases}$$

W pierwszym równaniu żądamy, aby długości wektora wejściowego i wyjściowego były takie same. W drugim równaniu żądamy, aby rzut wektora padającego na wektor prostopadły do płaszczyzny był przeciwny do rzutu wektora odbitego na wektor prostopadły. Jest to konsekwencją prawa odbicia. Wektor po odbiciu od płaszczyzny zmienia jedynie składową prostopadłą do tej płaszczyzny na przeciwną. Skoro zmienia jedynie tą składową, to reszta składowych, czyli składowe styczne (równoległe) do płaszczyzny, mają pozostać niezmienione. Fakt ten wyraża trzecie równanie. Równanie czwarte to definicja wektora $\vec{t}$ jako dowolnego wektora, który jest prostopadły do wektora prostopadłego do płaszczyzny, czyli właśnie wektora do niej równoległego. Proces rozwiązywania tego układu równań jest żmudny i miejscami skomplikowany. Zamiast tego podamy proponowane (przewidywane) rozwiązanie i pokażemy, że spełnia ono wymagany układ równań:

$$\overrightarrow{r_d'} = \overrightarrow{r_d} - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\vec{n}$$

Najpierw sprawdźmy pierwsze równanie, długość (do kwadratu) wektora odbitego:

$$\left\langle\overrightarrow{r_d'}\middle|\overrightarrow{r_d'}\right\rangle = \left\langle\overrightarrow{r_d} - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\vec{n}\middle|\overrightarrow{r_d} - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\vec{n}\right\rangle = \langle\overrightarrow{r_d}|\overrightarrow{r_d}\rangle + 4\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle^2}{\langle\vec{n}|\vec{n}\rangle^2}\langle\vec{n}|\vec{n}\rangle - 4\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\langle\vec{n}|\overrightarrow{r_d}\rangle$$
$$= \langle\overrightarrow{r_d}|\overrightarrow{r_d}\rangle$$

Następnie rzut wektora odbitego na wektor prostopadły do płaszczyzny:

$$\left\langle\overrightarrow{r_d'}\middle|\vec{n}\right\rangle = \left\langle\overrightarrow{r_d} - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\vec{n}\middle|\vec{n}\right\rangle = \langle\overrightarrow{r_d}|\vec{n}\rangle - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\langle\vec{n}|\vec{n}\rangle = -\langle\overrightarrow{r_d}|\vec{n}\rangle$$

oraz rzut wektora odbitego na samą płaszczyznę:

$$\left\langle\overrightarrow{r_d'}\middle|\vec{t}\right\rangle = \left\langle\overrightarrow{r_d} - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\vec{n}\middle|\vec{t}\right\rangle = \langle\overrightarrow{r_d}|\vec{t}\rangle - 2\frac{\langle\vec{n}|\overrightarrow{r_d}\rangle}{\langle\vec{n}|\vec{n}\rangle}\langle\vec{n}|\vec{t}\rangle$$

Korzystając z czwartego równania ($\langle\vec{t}|\vec{n}\rangle = 0$) otrzymujemy rozwiązanie:

$$\left\langle\overrightarrow{r_d'}\middle|\vec{t}\right\rangle = \langle\overrightarrow{r_d}|\vec{t}\rangle$$

co pokazuje, że przewidywane rozwiązanie jest poprawne. Zaznaczamy, że wektor prostopadły do płaszczyzny nie musi być znormalizowany. Normalizacja jest dokonywana już we wzorze na kierunek promienia odbitego.

Mimo, że powyższe rozwiązanie jest jedynym jakiego potrzebujemy, warto zaznaczyć, jakiej procedury należałoby użyć, aby obliczyć promień odbity od dowolnej powierzchni, nie tylko płaszczyzny. Powyższe równania rozwiązaliśmy w oparciu o formalizm wektora prostopadłego. Wnioski są więc poprawne tak długo, jak długo takim wektorem dysponujemy. W przypadku powierzchni parametrycznych, danych równaniem promienia

wodzącego $\vec{X}(u,v)$, wektor prostopadły (w pewnym punkcie powierzchni) można obliczyć w następujący sposób:

$$\overrightarrow{N_X}(u,v) = \partial_u \vec{X}(u,v) \times \partial_v \vec{X}(u,v)$$

Problemem pozostaje natomiast znalezienie punktu w którym promień będzie odbijał się od takiej powierzchni. W przypadku powierzchni, którą przybliżamy zbiorem trójkątów, rozwiązaniem tego problemu jest iterowanie po każdym trójkącie i korzystanie z metody podanej w akapicie 2.2. Gdybyśmy chcieli znaleźć punkt przecięcia prostej z powierzchnią daną *explicite*, musielibyśmy się zmagać za każdym razem z rozwiązaniem układu trzech równań nieliniowych. Oczywiście, istnieją metody pozwalające znajdować takie rozwiązania, natomiast często oparte są one o metodę największych gradientów, w związku z czym nie gwarantują znalezienia wszystkich rozwiązań układu. Oprócz tego, są one dość kosztowne obliczeniowo. Te dwie wady tego podejścia dyskwalifikują je prawie całkowicie. W grafice komputerowej w zasadzie wszystkie obiekty przybliża się siatką trójkątów. Wyjątek stanową podstawowe obiekty, takie jak sfera czy elipsa. Dla nich istnieją analityczne rozwiązania, zupełnie tak, jak w przypadku przecięcia prosta – trójkąt.

W miejscu tym powinniśmy jeszcze zaznaczyć, jakie struktury danych wykorzystujemy przy kalkulacji promieni odbitych. Tak jak zostało wspomniane, implementując *ray tracer* będziemy iterować po każdym trójkącie ze zbioru powstałego w wyniku triangulacji. Każdorazowe wyliczanie wektora normalnego dla danego trójkąta byłoby marnotrawstwem zasobów CPU. W obecnych czasach dysponujemy wystarczającymi zasobami pamięci, by móc statycznie przechowywać wektor normalny dla każdego z trójkątów. Tak też w rzeczywistości postąpimy. W momencie przygotowywania siatki stworzymy uporządkowaną listę trójkątów, a następnie dla każdego z tych trójkątów obliczymy wektor prostopadły do niego. Wektory te będziemy przechowywać w kolejnej uporządkowanej liście, tak by istniała wzajemna jednoznaczność między trójkątem a jego wektorem prostopadłym. W algorytmie *ray tracera* pozostanie nam odwołanie się do pamięci, co zajmuje znacznie mniej czasu procesora niż wykonanie iloczynu wektorowego i jego normalizacji.

2.6. Rozkład jednostajny na sferze

Jednym z końcowych etapów metody będzie całkowanie pewnej wielkości po przestrzeni zajmowanej przez kryształ oraz po wszystkich kierunkach w przestrzeni trójwymiarowej. Na potrzeby takiego całkowania musimy przyjąć, że kierunki te mają rozkład jednostajny na sferze. Oznacza to, że prawdopodobieństwo emisji w każdym z kierunków jest takie samo. Całkowanie będziemy wykonywać numerycznie i na siatce. Oznacza to, że każdy z przedziałów parametrów po których zamierzamy całkować przybliżymy jako zbiór liczb mieszczących się w tym przedziale. Dla każdej kombinacji wartości parametrów obliczymy wartość funkcji podcałkowej, wartości te zsumujemy i podzielimy przez liczbę otrzymanych kombinacji. Zastępujemy więc ciągłą całkę poprzez dyskretną średnią. Przybliżenie takie jest oczywiście tym lepsze, im więcej punktów w każdym z przedziałów całkowania wybierzemy.

Jeśli kierunek w przestrzeni chcielibyśmy wyrażać poprzez współrzędne sferyczne, to zagęszczenie poziomic (linii stałych parametrów) nie byłoby takie samo w okolicy każdego

punktu na sferze. Rozsianie punktów poprzez podzielenie wartości parametrów na równe odstępy skutkuje większą gęstością punktów w okolicy biegunów sfery, a mniejszą w okolicy równika. Należy więc dobrać funkcję, wedle której będziemy dokonywać parametryzacji. Załóżmy, że wychodzimy od standardowej parametryzacji sfery:

$$\vec{r}(\theta,\phi) = [\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\phi)], \theta \in [0,\pi], \phi \in (0,2\pi]$$

Chcemy znaleźć takie funkcje *u* i *v*, że następująca parametryzacja:

$$\vec{r}\big(u(\theta), v(\phi)\big)$$

daje jednorodny rozkład przedziałów $\theta \in [0,\pi], \phi \in (0,2\pi]$ na sferze. Formułując ściśle nasze żądanie, mówimy, że stosunek elementu powierzchni w przestrzeni parametrów: $d\theta d\phi$ do elementu powierzchni parametryzowanej: dS, jest stały, czyli nie zależy od wartości parametrów θ, ϕ:

$$\frac{d\theta d\phi}{dS} = const$$

Aby znaleźć rozwiązanie tego zagadnienia, zajmiemy się najpierw wyrażeniem elementu powierzchni parametryzowanej poprzez element powierzchni przestrzeni parametrów.

Przede wszystkim, dla dowolnej powierzchni $\vec{r}(u,v)$ zadanej parametrycznie, możemy rozważyć wektory styczne do linii siatki:

$$\overrightarrow{\Delta w_u} = \overrightarrow{w_1} - \overrightarrow{w_0} = d\vec{r}(u \to u + du)$$
$$\overrightarrow{\Delta w_v} = \overrightarrow{w_2} - \overrightarrow{w_0} = d\vec{r}(v \to v + dv)$$

Sytuacja ta została zobrazowana na rysunku 2.3. Element powierzchni rozpinany przez te wektory dany jest iloczynem wektorowym:

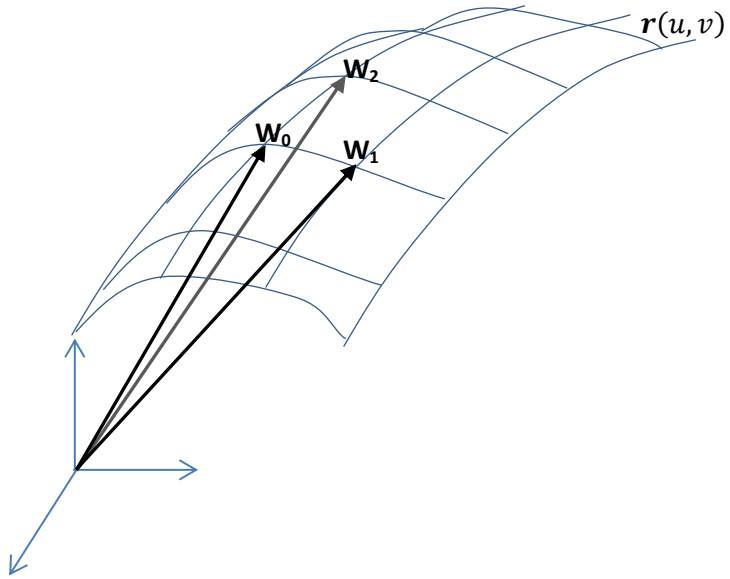
$$dS = \left|\overrightarrow{\Delta w_u} \times \overrightarrow{\Delta w_v}\right|.$$

Natomiast elementy $d\vec{r}$ mamy z tw. Taylora (o różniczce zupełnej):

$$d\vec{r}(u \to u + du) = \frac{\partial \vec{r}}{\partial u} du + \frac{\partial \vec{r}}{\partial v} 0 = \partial_u \vec{r}\, du$$

$$d\vec{r}(v \to v + dv) = \frac{\partial \vec{r}}{\partial u} 0 + \frac{\partial \vec{r}}{\partial v} dv = \partial_v \vec{r}\, dv$$

Równocześnie, dla dowolnych dwóch wektorów $\vec{a}$ i $\vec{b}$ mamy:



Rys. 2.3. Koncepcyjne przedstawienie powierzchni i linii stałych parametrów.

$$\left|\vec{a}\times\vec{b}\right|^2 = |\vec{a}|^2\left|\vec{b}\right|^2 sin^2(\vartheta)$$
$$|\vec{a}\times\vec{b}|^2 = |\vec{a}|^2\left|\vec{b}\right|^2(1-cos^2(\vartheta))$$
$$|\vec{a}\times\vec{b}|^2 = |\vec{a}|^2|\vec{b}|^2 - |a|^2|b|^2\,cos^2(\vartheta)$$

A ostatecznie zachodzi równość:

$$|\vec{a}\times\vec{b}|^2 = |\vec{a}|^2|\vec{b}|^2 - \left(\vec{a}\cdot\vec{b}\right)^2$$

Wracając do elementu powierzchni, wyrażamy go przez iloczyn wektorowy i rozpisujemy:

$$dS = \left|\overrightarrow{\Delta w_u}\times\overrightarrow{\Delta w_v}\right| = \partial_u\vec{r}\,du\times\partial_v\vec{r}\,dv = \sqrt{|\partial_u\vec{r}\,du|^2|\partial_v\vec{r}\,dv|^2 - (\partial_u\vec{r}\,du\cdot\partial_v\vec{r}\,dv)^2}$$
$$= \sqrt{|\partial_u\vec{r}|^2|\partial_v\vec{r}|^2 - (\partial_u\vec{r}\cdot\partial_v\vec{r})^2}\,du\,dv.$$

Powyższe wyrażenie można przepisać w języku tensora metrycznego. Z jego definicji otrzymujemy następujący wynik:

$$dS = \sqrt{\boldsymbol{g}_{uu}\boldsymbol{g}_{vv} - (\boldsymbol{g}_{uv})^2}dudv$$

Traktując metrykę Riemanna jako macierz, powyższe wyrażenie upraszcza się:

$$dS = \sqrt{|\det(\boldsymbol{g})|}\,dudv$$

Dla sfery metryka wygląda następująco:

$$\boldsymbol{g} = \begin{bmatrix}\partial_\theta\vec{r}\cdot\partial_\phi\vec{r} & \partial_\theta\vec{r}\cdot\partial_\phi\vec{r}\\ \partial_\theta\vec{r}\cdot\partial_\phi\vec{r} & \partial_\phi\vec{r}\cdot\partial_\phi\vec{r}\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & \sin^2(\theta)\end{bmatrix}$$

Tak więc poszukiwany element powierzchni sprowadza się do:

$$dS = |\sin(\theta)|d\theta d\phi$$

Wynik ten pozwala zauważyć, że stosunek $\frac{d\theta d\phi}{dS}$ nie jest stały. Okazuje się, że aby znaleźć funkcje skalujące parametry, należy rozwiązać układ równań różniczkowych cząstkowych. Nie jest to proste zagadnienie, a my proponujemy po raz kolejny metodę przewidywań. Dla następujących funkcji skalujących:

$$u(\theta) = \arccos(2\theta-1)\,, \qquad \theta\in[0,1]$$
$$v(\phi) = 2\pi\phi, \qquad \phi\in(0,1]$$

parametryzacja sfery przyjmuje postać:

$$\vec{r}\big(u(\theta),v(\phi)\big) = \left[\cos(2\pi\phi)\sqrt{1-(2\theta-1)^2},\sin(2\pi\phi)\sqrt{1-(2\theta-1)^2},2\theta-1\right]$$

Obliczamy jej pochodne cząstkowe i otrzymujemy wynik:

$$\partial_\theta \vec{r}\big(u(\theta), v(\phi)\big) = \left[\cos(2\pi\phi)\frac{-2(2\theta-1)}{\sqrt{1-(2\theta-1)^2}}, \sin(2\pi\phi)\frac{-2(2\theta-1)}{\sqrt{1-(2\theta-1)^2}}, 2\right]$$

$$\partial_\phi \vec{r}\big(u(\theta), v(\phi)\big) = \left[-2\pi\sin(2\pi\phi)\sqrt{1-(2\theta-1)^2}, 2\pi\cos(2\pi\phi)\sqrt{1-(2\theta-1)^2}, 0\right]$$

Wtedy metryka Riemanna wygląda następująco:

$$\boldsymbol{g} = \begin{bmatrix} \partial_\theta\vec{r}\cdot\partial_\phi\vec{r} & \partial_\theta\vec{r}\cdot\partial_\phi\vec{r} \\ \partial_\theta\vec{r}\cdot\partial_\phi\vec{r} & \partial_\phi\vec{r}\cdot\partial_\phi\vec{r} \end{bmatrix} = \begin{bmatrix} \frac{1}{\theta-\theta^2} & 0 \\ 0 & 16\pi^2(\theta-\theta^2) \end{bmatrix}$$

Obliczając element powierzchni zadanej taką parametryzacją:

$$dS = \sqrt{|\det(\boldsymbol{g})|}\, d\theta d\phi = 4\pi d\theta d\phi$$

Widzimy, że jest on stały i nie zależy od wartości parametrów, podobnie jak poszukiwany stosunek:

$$\frac{d\theta d\phi}{dS} = \frac{1}{4\pi} = const$$

Udało się zatem pokazać, że takie skalowanie zmiennych dla standardowej parametryzacji sfery daje jednorodny rozkład gęstości na sferze.

Aby zasiać punkty jednorodnie na sferze, należy więc zasiać losowo punkty θ i ϕ w przedziałach [0,1] i dla każdego zasianego punktu obliczyć wartość następującego wyrażenia:

$$\vec{r}\big(u(\theta), v(\phi)\big) = \left[\cos(2\pi\phi)\sqrt{1-(2\theta-1)^2}, \sin(2\pi\phi)\sqrt{1-(2\theta-1)^2}, 2\theta-1\right]$$

Otrzymamy w ten sposób zbiór wektorów trójwymiarowych o jednostkowej długości i losowym kierunku.

2.7. Śledzenie promieni

Opis algorytmu *ray-tracingu* zostawiliśmy na sam koniec, aby móc odwoływać się do poznanych już metod. Wejściem dla tego algorytmu będą:

- Półprosta parametryczna:

  $$\vec{r}(t>0) = \overrightarrow{r_0} + \overrightarrow{r_d}\, t$$

  Czyli w zasadzie dwa wektory: $\overrightarrow{r_0}$ i $\overrightarrow{r_d}$. Półprosta reprezentuje śledzony promień.

- Siatka powierzchni wewnątrz której dokonujemy śledzenia, czyli zbiór trójkątów, którego opis znajduje się w akapicie 2.1.
- Zbiór wektorów prostopadłych do siatki, opisany pod koniec akapitu 2.5.

Na wyjściu oczekujemy uzyskać zbiór punktów które tworzą krzywą łamaną – trajektorię śledzonego promienia.

Algorytm śledzenia promieni podzielimy na dwie części:

1) Znajdowanie promienia odbitego
2) Faktyczne śledzenie promienia

Na wejście pierwszej części algorytmu podajemy te same wielkości, co do całego algorytmu oraz dodatkowo wielkość którą nazwiemy roboczo *lastRefIdx*. Indeksuje ona trójkąt od którego nastąpiło ostatnie odbicie.

1) Ustal $i = 0$
2) Dla *i-tego* trójkąta w siatce, sprawdź, czy podana półprosta przecina się z tym trójkątem (metoda z akapitu 2.1.)
3) Jeżeli się przecina, oraz *i* jest różne od *lastRefIdx*, to:
    - Dodaj do słownika przecięć klucz będący rozwiązaniem t_c (akapit 2.1.)
    - Dodaj do słownika przecięć wartość dla powyższego klucza będącą punktem przecięcia (akapit 2.1.)
    - Dodaj do słownika przecięć wartość *i* dla powyższego klucza
4) Jeżeli istnieje następny trójkąt w siatce to zwiększ *i* o 1 i wróć do punktu 2)
5) Posortuj słownik przecięć rosnąco względem kluczy i wypierz zerowy element
6) Rozważ przecięcie promienia z płaszczyzną $z = 0$
7) Jeżeli promień przecina się z płaszczyzną $z = 0$ wcześniej niż trójkątem odpowiadającym pierwszemu elementowi słownika ($t_c[0] > t_{z=0} > 0$), to:
    - Zwróć położenie punktu przecięcia z $z = 0$ oraz wystaw flagę *z0_reached*
8) W przeciwnym wypadku, jeżeli słownik przecięć jest pusty i przecięcie z $z = 0$ jest poprawne ($t_{z=0} > 0$), to:
    - Zwróć położenie punktu przecięcia z $z = 0$ oraz wystaw flagę *z0_reached*
9) W przeciwnym wypadku:
    - Odbij promień od płaszczyzny odpowiadającej pierwszemu elementowi posortowanego słownika (akapit 2.5.)
    - Zwróć: punkt przecięcia jako $\overrightarrow{r_0'}$, wektor odbity jako $\overrightarrow{r_d'}$, indeks trójkąta.

Dysponując powyższym algorytmem, formułujemy cały *ray tracer*:

1) Ustal *lastRefIdx* na N, gdzie N jest liczbą trójkątów w siatce
2) Ustal rejestr aktualnej pozycji na $\overrightarrow{r_0}$
3) Ustal rejestr aktualnego kierunku na $\overrightarrow{r_d}$
4) Ustal $i = 0$
5) Dodaj do listy punktów trajektorii stan rejestru aktualnej pozycji
6) Znajdź promień odbity i ustal rejestr aktualnej pozycji jako $\overrightarrow{r_0'}$ i rejestr aktualnego kierunku jako $\overrightarrow{r_d'}$
7) Inkrementuj *i* o *1*
8) Jeżeli wystawiono *z0_reached* lub $i \geq maxRef$, to przerwij.
9) W przeciwnym wypadku, wróć do 5)

Tak skonstruowany algorytm powinien z powodzeniem śledzić zadany promień. Można zauważyć, że wprowadzona została zmienna o nazwie *maxRef*, służy ona przerwaniu działania algorytmu, gdy liczba odbić wiązki będzie zbyt duża. Wspominaliśmy o potrzebie użycia takiego podejścia na początku tego rozdziału.

3. Implementacja

Algorytmy typu *ray tracer* posiadają wysoki potencjał do zrównoleglenia obliczeń. Jest tak z powodu braku wzajemnej zależności przy śledzeniu dwóch i więcej promieni. Nie ma potrzeby wymiany danych między procedurami śledzącymi dwa różne promienie, jak i wynik śledzenia promienia nie jest zależy od przebiegu śledzenia pozostałych promieni. Daje to możliwość rozdystrybuowania procedury śledzenia na wiele wątków, dla każdego należałoby jedynie dostarczyć zmienne definiujące promienie. Potencjał ten stał się jedną z przyczyn dla których tak wielką popularność zyskały tak zwane procesory graficzne (GPU). Architektura GPU zakłada wiele odrębnych modułów przetwarzających dane. Instrukcje obsługiwane przez architekturę GPU są specyficzne ze względu na typ danych wejściowych – podstawowe zadanie które mają spełniać to możliwość przetwarzania wektorów danych.

Wracając jednak do istoty problemu postawionego w tej pracy – algorytmy opisane w poprzednim rozdziale muszą zostać zaimplementowane. Od implementacji tej nie będziemy żądać najwyższej wydajności, istotna dla nas będzie natomiast jej klarowność. W związku z tym, językiem programowania jaki został wybrany na te potrzeby jest Python (w wersji 2.7 [12]). Python, jeśli chodzi o wydajność programów w nim napisanych, plasuje się bardzo nisko. Główną przyczyną tego faktu jest to, że jest on językiem interpretowanym. Przewagę zdobywa jednak w momencie, w którym chodzi o szybkość produkcji kodu i wydajność procesu implementacji. Implementacja zaawansowanych algorytmów w Pythonie potrafi zająć do kilkudziesięciu razy mniej czasu niż w języku C. Nie bez znaczenia jest także klarowność jego składni. Kod napisany w Pythonie jest zazwyczaj samotłumaczący się. Projektanci Pythona za jeden z głównych celów postawili sobie przejrzystość projektowanego kodu tak, by szczegóły implementacyjne, zależne często od architektury, zostały zredukowane do absolutnego minimum.

Jednak w trakcie implementacji *ray tracera* w Pythonie okazało się, że mimo tych zalet, wydajność kodu jest tak niska, że osiągnięcie nawet miernych rezultatów jest niemożliwe. Okazuje się jednak, że projektanci Pythona dostarczają wraz z jego dystrybucją szeregu narzędzi developerskich, w tym biblioteki do języka C, pozwalające na zbudowanie interfejsu między tymi dwoma językami. W trakcie badań nad wydajnością kodu okazało się, że tzw. *Bottleneckiem*, czyli fragmentem kodu który pochłaniał najwięcej zasobów procesora, była procedura odbicia promienia od powierzchni. Podjęta została więc decyzja o próbie zaimplementowania tego fragmentu w języku C, za pomocą bibliotek API (application programming interface). Okazało się, że zabieg ten był wystarczający, by osiągnąć rezultaty pozwalające na swobodną pracę z programem. Czas poświęcany na obsługę tej procedury spadł blisko 1700 razy.

Implementacja mogłaby obejmować użycie wątków wirtualnych systemu operacyjnego, jednak z powodu zabiegu zastosowanego w najbardziej popularnej i niezawodnej dystrybucji Pythona (CPython) o nazwie GIL (Global Interpreter Lock), zrównoleglenie kodu nie spowoduje wzrostu wydajności [9].

3.1.Struktura kodu

Wynikowy kod obejmuje w sumie dwa moduły. W jednym zaimplementowane zostały algorytmy służące do reprezentacji powierzchni kryształu oraz do śledzenia promieni. Jako część tego modułu traktujemy także podmoduł napisany w języku C, którego jedynym zadaniem jest dostarczenie interfejsu procedury odbijania promienia. Pierwszy moduł można więc traktować go jako tzw. *Backend*. W drugim module dostarczamy narzędzi specyficznych do postawionego w pracy problemu dotyczącego wydajności scyntylacji. Wprowadzone zostały w nim funkcje pozwalające na takie manipulowanie obiektami z modułu pierwszego, by można było w prosty sposób liczyć wydajność scyntylacji, konstruować wykresy i renderować obraz. Można myśleć o nim jak o tzw. *Frontendzie*.

Podczas prezentacji i omawiania kodu pominięte zostaną notki dokumentacyjne i niektóre komentarze. Nie będziemy zamieszczać i opisywać niektórych fragmentów samego kodu, jeżeli ten nie jest wykorzystywany. Fragmenty takie istnieją jako zaplecze do dalszych prac nad aplikacją. Nie są jednak niezbędne do zrealizowania celów tej pracy.

3.2. Backend

Backendem jest moduł o nazwie `fwdraytracing.geo`. Jak widać, jest to także podmoduł zbiorczej biblioteki o nazwie `fwdraytracing`. Zawiera on trzy klasy dziedziczące jedynie po szablonie standardowego obiektu:

```
01 class Ray(object):
02     #...
03 class SurfaceModel(object):
04     #...
05 class DifferentialManifold(object):
06     #...
```

oraz jedną funkcję:

```
01 def primitives(prim, params):
02     #...
```

Moduł korzysta z następujących bibliotek, importowanych na samym początku pliku:

```
01 import sympy as sym
02 import numpy as np
03 from scipy.optimize import minimize
04 from scipy.spatial import Delaunay
05 import ctrace as ct
```

3.2.1. Klasa `class DifferentialManifold(object)`

Omawianie kodu rozpoczniemy od klasy `class DifferentialManifold(object)`. Zawiera ona atrybuty i metody definiujące parametryzację powierzchni, a uściślając to do terminologii wprowadzonej w drugim rozdziale, reprezentuje ona nietrywialny brzeg kryształu.

```
01  class DifferentialManifold(object):
02
03      def __init__(self, args_str, param_str, ranges_tpl):
04          self.u, self.v = sym.symbols(args_str)
05          self.parameterization = sym.Matrix(sym.sympify(param_str)).T
06          self.u_range = ranges_tpl[0]
07          self.v_range = ranges_tpl[1]
08
09          self.nR = sym.lambdify((self.u, self.v), self.parameterization[0,:])
010
011     def bring(self, u, v):
012         if not ((self.u_range[0] <= u <= self.u_range[1])
013                 and
014                 (self.v_range[0] <= v <= self.v_range[1])):
015             raise ValueError(
016             'Arguments out of range: u:{0}; v:{1}'.format(self.u_range,self.v_range)
    )
017
018         evParameterization = self.parameterization.evalf(subs = {self.u : u, self.v
    : v})
019         return np.array(evParameterization.tolist()).astype(np.float64)
020
021     def tangent_bundle(self):
022         DuR = sym.diff(self.parameterization, self.u)
023         DvR = sym.diff(self.parameterization, self.v)
024         return sym.Matrix([DuR,DvR])
025
026     def normal_bundle(self):
027         DuR = sym.diff(self.parameterization, self.u)
028         DvR = sym.diff(self.parameterization, self.v)
029         normal = DuR.cross(DvR)
030         return sym.simplify(normal)
031
032     def hesse_matrix(self):
033         DuR = sym.diff(self.parameterization, self.u)
034         DvR = sym.diff(self.parameterization, self.v)
035         DuuR = sym.simplify(sym.diff(DuR, self.u))
036         DvvR = sym.simplify(sym.diff(DvR, self.v))
037         DuvR = sym.simplify(sym.diff(DuR, self.v))
038         H = sym.Matrix([[DuuR,DuvR],[DuvR,DvvR]])
039         return H
040     def riemann_metric(self):
041         DuR = sym.diff(self.parameterization, self.u)
042         DvR = sym.diff(self.parameterization, self.v)
043         guu = DuR.dot(DuR)
044         gvv = DvR.dot(DvR)
045         guv = DuR.dot(DvR)
046         g = sym.Matrix([
047             [guu, guv],
048             [guv, gvv]]
049             )
050         return sym.simplify(g)
```

Konstruktor powyższej klasy, czyli metoda `__init__(self, args_str, param_str, ranges_tpl)`, przyjmuje trzy argumenty:

- `args_str` – string zawierający nazwy argumentów parametryzacji, oddzielone spacjami, np. `args_str = 'u v'`
- `param_str` – string zawierający parametryzację nietrywialnego brzegu, np. `param_str = '[sin(u), cos(v), 2*v**2]'`

- `ranges_tpl` – krotka dwóch krotek, zawierająca zakresy zmiennych parametryzacji, np. `ranges_tpl = ((u_min, u_max), (v_min, v_max))`

W konstruktorze uzupełniamy odpowiadające argumentom atrybuty oraz dokonujemy przekształcenia stringa parametryzacji na obiekt będący macierzą modułu *sympy* (linia `05`).

Następną metodą jest metoda `bring(self, u, v)`, która spełnia oczywistą funkcjonalność, czyli zwraca wartość wektora parametryzacji dla podanych argumentów - `u, v`. W linii `010` rozpoczynamy sprawdzanie, czy żądane wartości argumentów mieszczą się w przedziale podanym do konstruktora. Jeżeli argumenty te nie mieszczą się w przedziale, to podnoszony jest błąd typu `ValueError`. Parametryzacja `self.parameterization` jest obiektem modułu *sympy*, który jest implementacją systemów algebry symbolicznej. Operuje on na pojęciu symbolu i aby wywołać numeryczną wartość parametryzacji, musimy dokonać podstawienia typowego dla formalizmu *sympy*. Dokonywane to jest w linii `018`. W następnej linii zwracana jest lista przekonwertowana na tablice modułu *numpy*. Gdy następnym razem będziemy używali sformułowania *tablica*, będziemy mieć na myśli tablicę modułu *numpy*. Jednak na potrzeby obliczeń numerycznych należy ekwiwalentnie używać wywołania `.nR(u,v)`.

W metodzie `tangent_bundle(self)` dokonujemy symbolicznego różniczkowania po parametrach *u* i *v*. Zwracana jest symboliczna macierz która w swoich dwóch wierszach ma wektory styczne do linii stałych parametrów. W języku geometrii różniczkowej można o takiej strukturze mówić jak o wiązce stycznej.

W metodzie `normal_bundle(self)` także dokonujemy różniczkowania po parametrach, z tą różnicą, że wyniki mnożymy ze sobą wektorowo. Daje to w wyniku wektor prostopadły do powierzchni, o długości proporcjonalnej do krzywizn głównych w danym punkcie. Wektor ten jest zwracany jako symboliczna macierz.

W metodach `hesse_matrix(self)` i `riemann_metric(self)` także dokonujemy różniczkowania parametryzacji po jej parametrach, a na wynikach tych operujemy tak, by uzyskać odpowiednio macierz Hessego i metrykę Riemanna dla danej powierzchni.

W obecnej wersji kodu nie korzystamy z czterech ostatnich metod. Zostały one jednak przedstawione, by zaprezentować, jak w łatwy sposób używać Pythona jako systemu algebry symbolicznej. Na przedstawionym listingu zostały zapisane najważniejsze pola i metody. Faktyczna jej implementacja zawiera o wiele więcej zarejestrowanych pól i metod. Nie korzystamy z nich jednak w tworzonym oprogramowaniu – na razie. W przyszłości ich obecność może być przydatna, o ile autor zdecyduje się na dalsze rozwijanie projektu.

Podsumowując, na razie klasa `class DifferentialManifold(object)` służy jako kontener przechowujący symboliczne informacje o nietrywialnym brzegu kryształu.

3.2.2. Klasa `class SurfaceModel(object)`

W klasie `class SurfaceModel(object)` przechowujemy numeryczny model nietrywialnego brzegu kryształu. Została ona zaprojektowana tak, by można ją było później ulepszać. Z tego powodu, niektóre metody zawierają jedynie jedną instrukcję.

```
01 class SurfaceModel(object):
02 
03     def __init__(self, precision):
04 
05         self.precision = precision
06 
07     def addSheet(self, manifold):
08 
09         self.sheet = manifold
010 
011     def bakeMesh(self):
012 
013         #lets set the entries
014         uTable = np.linspace(self.sheet.u_range[0],self.sheet.u_range[1],self.precis
    ion)
015         vTable = np.linspace(self.sheet.v_range[0],self.sheet.v_range[1],self.precis
    ion)
016         uLen = len(uTable)
017         vLen = len(vTable)
018         pts = np.zeros((uLen*vLen,3))
019         uvLUT = np.zeros((uLen*vLen,2))
020 
021         #next, lets calculate a mesh points
022         for i in xrange(uLen):
023             for j in xrange(vLen):
024                 uvLUT[vLen*i + j, 0] = uTable[i]
025                 uvLUT[vLen*i + j, 1] = vTable[j]
026                 pts[vLen*i + j,:] = self.sheet.nR(uTable[i], vTable[j])[0]
027 
028         #now perform Delaunay triangulation on flat parameter space:
029         uvDelTri = Delaunay(uvLUT)
030         uvTri = uvLUT[uvDelTri.simplices]
031         triLen = len(uvTri)
032         surfTri = np.zeros((triLen,3,3))
033         normalGrid = np.zeros((triLen,3))
034         for i in xrange(triLen):
035             surfTri[i,0,:] = self.sheet.nR(*uvTri[i,0,:])[0]
036             surfTri[i,1,:] = self.sheet.nR(*uvTri[i,1,:])[0]
037             surfTri[i,2,:] = self.sheet.nR(*uvTri[i,2,:])[0]
038             normalGrid[i,:] = np.cross((surfTri[i,2,:] - surfTri[i,0,:]),
039                                        (surfTri[i,2,:] - surfTri[i,1,:]))
040             normalGrid[i,:] = normalGrid[i,:]/np.linalg.norm(normalGrid[i,:])
041         self.mesh = surfTri
042         self.normal = normalGrid
043 
044     def fromRawMesh(self, raw):
045 
046         surfTri = raw.copy()
047         triLen = len(surfTri)
048         normalGrid = np.zeros((triLen, 3))
049         for i in xrange(triLen):
050             normalGrid[i,:] = np.cross((surfTri[i,2,:] - surfTri[i,0,:]),
051                                        (surfTri[i,2,:] - surfTri[i,1,:]))
052             normalGrid[i,:] = normalGrid[i,:]/np.linalg.norm(normalGrid[i,:])
053         self.mesh = surfTri
054         self.normal = normalGrid
```

Konstruktor przedstawionej klasy, metoda `__init__(self, precision)`, przyjmuje tylko jeden parametr – `precision`. Jego wartość jest liczbą punktów, na które zostanie podzielony przedział zmienności każdej ze zmiennych parametryzujących nietrywialny brzeg kryształu. Im jego wartość jest wyższa, tym siatka dokładniej odwzorowuje rzeczywistą powierzchnię.

Metoda `addSheet(self, manifold)` wykonuje jedynie jedną instrukcję – rejestruje w polu klasy jej argument `manifold`. Argument ten musi być obiektem klasy `class DifferentialManifold(object)`. W ten sposób przekazujemy omawianej klasie obiekt do numerycznej obróbki.

Metoda `bakeMesh(self)` jest główną metoda omawianej klasy. Zajmuje się ona numeryczną obróbką symbolicznej powierzchni przekazywanej do metody `addSheet(self, manifold)`. W liniach `014` i `015` dzielimy przedziały zmienności rozmaitości na tablicę dyskretnych punktów. Ich liczba dana jest przez argument przekazywany do konstruktora, a przechowywana jest w polu `self.precision`. Na wypadek zmian architektury tej klasy, w obrębie metody `bakeMesh(self)` posługujemy się liczbą punktów na które został podzielony przedział zmienności parametru, która przechowywana jest dla każdego parametru oddzielnie i liczona jest już po zasianiu punktów w przedziałach (linie `016` i `017`). W pętli `022` dokonujemy specyficznego zabiegu, który można generalnie nazwać iloczynem kartezjańskim zbiorów, reprezentowanych przez tablice `uTable`, `vTable`. Zabieg indeksowania w sposób: `uvLUT[vLen*i + j, 0]` powinien być znany programistom C. Jeżeli indeks `i` indeksuje wiersz macierzy, a indeks `j` jej kolumnę, to wyrażenie `vLen*i + j` mówi jaki indeks będzie miała macierz spłaszczona (w sensie C, nie Fortrana) o liczbie kolumn równej `vLen`. Możemy więc sobie wyobrazić, że indeksujemy po wierszach i kolumnach macierzy (po punktach z tablicy *u*, potem po punktach z tablicy *v*), a macierz złożoną ze wszystkich możliwych par punktów *(u, v)* spłaszczamy i wpisujemy do tablicy `uvLUT`. W ten sposób, we wierszach tej zmiennej znajdują się wszystkie możliwe kombinacje par *(u, v)*. Z kolei wiersze tablicy `pts` przechowują wartości wektora parametryzacji odpowiadające parametrom *(u, v)* zawartym w wierszach tablicy `uvLUT`. Tablica `pts` jest więc zbiorem punktów siatki płaszczyzny parametrów *u, v*. W linii `029` dokonujemy triangulacji Delanuay'a tej siatki. Jak wspomnieliśmy w rozdziale drugim, procedura ta jest dostępna w module *scipy* i z niej właśnie korzystamy. Zwraca ona obiekt, o którym można przeczytać więcej w dokumentacji technicznej pakietu *scipy* [11]. Nam wystarczy wiedzieć, że zmienna `uvTri` zdefiniowana tak jak w linii `030` będzie tablicą trójwymiarową, której strony (zerowy numer indeksu) przechowują macierze, której wiersze zawierają współrzędne wierzchołków trójkąta. Strona tablicy macierzy jest więc interpretowana jako trójkąt. W kolejnej pętli (linia `034`) dokonujemy wyniesienia wierzchołków trójkątów (które leżą na płaszczyźnie parametrów) w przestrzeń trójwymiarową, poprzez zastosowanie parametryzacji na tych punktach. Iterujemy po stronach (trójkątach) i dla każdego wierzchołka trójkąta na płaszczyźnie *u, v*, obliczamy odpowiadający wierzchołek powierzchni parametrycznej. Wynik zapisywany jest w tablicy trójwymiarowej o nazwie `surfTri`. Jej struktura jest taka sama, jak struktura tablicy `uvTri`, z tą różnicą, że wiersze jej stron mają trzy elementy, a nie dwa (teraz są trzy współrzędne, a były dwa parametry). W tablicy `normalGrid` zapisujemy wektor prostopadły do danego trójkąta. Pętla ta jest kwintesencją implementacji metod opisywanych w rozdziale 2.1. Oprócz tego, w rozdziale drugim zaznaczyliśmy, że wektora normalnego dla trójkąta nie będziemy liczyć za każdym razem gdy następować będzie odbicie, ale będziemy przechowywać je dla każdego trójkąta w statycznej zmiennej. Ten właśnie zabieg dokonywany jest na końcu omawianej pętli. Implementacja liczenia wektora normalnego jest zgodna z procedurą opisaną na

początku rozdziału 2.5. Zmienne `surfTri` i `normalGrid` są rejestrowane w polach klasy: `.mesh` i `.normal`.

Metoda `fromRawMesh(self, raw)` służy do zarejestrowania pól: `.mesh` i `.normal`, w przypadku, w którym nietrywialny brzeg kryształu miałby być dany po prostu zbiorem trójkątów. Przyjmuje ona jako argument zmienną, która ma taką samą strukturę, jak zmienna `surfTri`. Na razie nie jest ona wykorzystywana, ale można jej użyć, gdyby chcieć wygenerowaną wcześniej siatkę odczytywać np. z pliku.

3.2.3. Klasa `class Ray(object)`

Klasa jest modelem promienia, który może podlegać stopniowemu odbijaniu. Posiada metody pozwalające promień odbić, śledzić i obliczać jego charakterystyki.

```
01 class Ray(object):
02 
03     def __init__(self, init_pos, init_dir, regCnt = 100):
04         self.init_pos = init_pos
05         self.init_dir = init_dir
06         self.pos = init_pos.copy()
07         self.dir = init_dir.copy()
08         self.path = np.zeros((regCnt,3))
09         self.reflectCnt = 0
010 
011         self.model = None
012         self._lastRefIdx = None
013         self._triLen = None
014         self._regCnt = regCnt
015 
016     def placeEnviorment(self, model):
017         self.model = model
018         self._lastRefIdx = len(model.mesh)
019         self._triLen = len(self.model.mesh)
020 
021     def reflect(self):
022 
023         cres = ct.reflect(self.pos, self.dir, self.model.mesh, self.model.normal, se
    lf._lastRefIdx)
024         if not (cres is None):
025             self.pos = cres[0]
026             self.dir = cres[1]
027             self._lastRefIdx = cres[2]
028             return cres[0], cres[1]
029 
030     def trace(self, maxRecursion = 100):
031 
032         if maxRecursion >= self._regCnt:
033             raise ValueError("maxRecursion can't be greater than regCnt (%d)" % self
    ._regCnt)
034 
035         self.path[0,:] = self.pos.copy()
036         self.reflectCnt = 0
037         for i in xrange(maxRecursion):
038             self.reflect()
039             self.path[i+1,:] = self.pos.copy()
040 
041             if np.isclose(self.pos[2], 0):
042                 break
043             self.reflectCnt += 1
044 
045         return self.reflectCnt
```

```
046
047     def pathLength(self):
048
049         divs = self.path[:(self.reflectCnt+1),:] -
    self.path[1:(self.reflectCnt+2),:]
050         return np.sqrt(np.square(divs).sum(1)).sum()
```

Konstruktor obiektu, metoda `__init__(self, init_pos, init_dir, regCnt = 100)` przyjmuje trzy argumenty. Pierwszy - `init_pos` – jest tablicą jednowymiarową o trzech elementach. Reprezentuje współrzędne początkowe wiązki. Drugi - `init_dir` – także jest tablicą jednowymiarową o trzech elementach. Reprezentuje on kierunek początkowy wiązki. Argument `regCnt` został wprowadzony ze względu na wydajność. Mówi on o tym, ile maksymalnie odbić można rozważać w ramach danej instancji wiązki. Wartość domyślna, czyli `100` jest w zupełności wystarczająca na nasze potrzeby.

Konstruktor rejestruje położenie i kierunek początkowy wiązki w polach `.init_pos` i `.init_dir`. Klasa przechowuje aktualny punkt startowy wiązki i jej kierunek w polach: `.pos` i `.dir`. Są one aktualizowane gdy wiązka ulegnie odbiciu. Z oczywistych względów są one inicjalizowane wartościami przekazanymi w argumentach konstruktora.

Klasa przechowuje informacje o poprzednich kierunkach wiązki i jej startowych pozycjach w polu `.path`. Jest ona inicjalizowana zerami, na długość przekazywaną w argumencie `regCnt`. Zmienn `.path` jest tablicą która w swoich wierszach przechowuje kolejne pozycje początkowe prostej łamanej (trajektorii wiązki). Ze względów bezpieczeństwa, argument `regCnt` jest rejestrowany w polu pseudo-prywatnym.

Metoda `placeEnviorment(self, model)` służy do zarejestrowania środowiska w którym odbija się wiązka, czyli nietrywialnego brzegu. Rejestrowany jest model rozmaitości, czyli obiekt klasy `class SurfaceModel(object)`. Tego typu także musi być argument metody.

Metoda `reflect(self)` ma za zadanie odbić wiązkę o aktualnej pozycji startowej `.pos` i kierunku `.dir` od brzegu `.model`. Wywołuje ona metodę `.reflect(c_pos, c_dir, c_mesh, c_normal, c_lastRefIdx)` modułu `ctrace` który na początku został zarejestrowany w przestrzeni nazw jako `ct`. Przyjmuje ona argumenty nienazwane, dlatego należy podać je w kolejności:

- `c_pos` – pozycja początkowa promienia który ma być odbity, struktura taka jak w polu `.pos`
- `c_dir` – kierunek promienia który ma być odbity, struktura taka jak w polu `.dir`
- `c_mesh` – pole `.mesh` obiektu klasy `class SurfaceModel(object)`
- `c_normal` – pole `.normal` obiektu klasy `class SurfaceModel(object)`
- `c_lastRefIdx` – liczba całkowita, indeksująca trójkąt od którego nastąpiło ostatnie odbicie wiązki opisywanej przez instancje klasy `class Ray(object)`. Jeżeli ma być to pierwsze odbicie, to argument ten powinien być równy liczbie trójkątów w siatce.

Moduł `ctrace` jest rozszerzeniem do Pythona napisanym w języku C. Jego jedynym zadaniem jest dostarczenie metody `ctrace.reflect(...)`. Metoda ta zwraca krotkę, która przypisywana jest do zmiennej `cres`. Krotka ta ma następującą strukturę:

- `cres[0]` – pozycja początkowa promienia po odbiciu
- `cres[1]` – kierunek promienia po odbiciu
- `cres[2]` – indeks trójkąta od którego nastąpiło odbicie

Jeżeli przechwycenie danych z modułu `ctrace` się powiodło, to odpowiednie pola klasy są uzupełniane.

Metoda `trace(self, maxRecursion = 100)` dokonuje śledzenia promienia z maksymalną liczbą odbić równą `maxRecursion`. Jeżeli przekazano argument, który przekracza długość rejestru odbić, to podnoszony jest błąd typu `ValueError`. Pole `.path` jest uzupełniane o pozycję startową i rejestr liczby odbić jest zerowany. W pętli wywoływana jest metoda `reflect(self)` i aktualizowana jest tablica `.path`. W momencie, w którym składowa *z-owa* położenia początkowego po odbiciu osiąga wartość zera, pętla jest przerywana. Jeżeli tak się nie dzieje, to licznik odbić jest inkrementowany. Jest to implementacja algorytmu opisanego w rozdziale 2.7.

3.2.4. Funkcja `primitives(prim, params)`

Funkcja ta służy do zamodelowania nietrywialnego brzegu kryształu, którego nie da się opisać za pomocą rozmaitości pseudoriemannowskich. Będzie ona rozwijana w miarę potrzeb, ale na razie jest w stanie zwrócić jedynie siatkę trójkątów dla prostopadłościanu. Wyjście z tej funkcji może być przekazane bezpośrednio do metody `fromRawMesh(self, raw)` obiektu klasy `class SurfaceModel(object)`.

```
01 def primitives(prim, params):
02     if prim == 'cuboid':
03         A = params[0] #x
04         L = params[1] #y
05         H = params[2] #z
06 
07         p =   np.array([[0.5*A, -.5*L, 0],
08                         [0.5*A, 0.5*L, 0],
09                         [-.5*A, 0.5*L, 0],
010                        [-.5*A, -.5*L, 0],
011                        [0.5*A, -.5*L, H],
012                        [0.5*A, 0.5*L, H],
013                        [-.5*A, 0.5*L, H],
014                        [-.5*A, -.5*L, H]])
015         tris = np.array([[p[0], p[3], p[4]], [p[7], p[4], p[3]],
016                          [p[0], p[1], p[4]], [p[5], p[4], p[1]],
017                          [p[1], p[2], p[5]], [p[6], p[5], p[2]],
018                          [p[7], p[6], p[2]], [p[3], p[2], p[7]],
019                          [p[5], p[4], p[6]], [p[4], p[7], p[6]]])
020         return tris.copy()
```

W zależności od argumentu `prim`, docelowo wybierane będą kształty nieróżniczkowe. W obecnej wersji, funkcja obsługiwać będzie jedynie kształt prostopadłościenny. W sekwencji przełączającej, sprawdzane będą wartości zmiennej `prim`. Obecnie, dostępna wartość to string `'cuboid'`. Argument `params` to krotka, której zawartość będzie specyficzna dla danego typu prymitywu. Dla prostopadłościanu, krotka wypełniona jest długościami boków prostopadłościanu kolejno w kierunku: *x*, *y*, i *z*. Zmienna `p` przechowuje konieczne punkty, a w zmiennej `tris` przechowywane są trójkąty. Zmienna ta jest zwracana. Jej struktura jest

zgodna ze strukturą siatki trójkątów, wobec czego może być użyta jako surowe wejście dla jednej z metod klasy `class SurfaceModel(object)`, konstruującej numeryczny model nietrywialnego brzegu.

3.2.5. Moduł `ctrace`

Moduł ten traktujemy jako *backend*. Tak jak zostało opisane w rozdziale 3.2.3., wykonuje on procedurę odbicia promienia od brzegu kryształu. Został on napisany w języku C z powodów wyjaśnionych na początku tego rozdziału.

```
01 #include "include\Python\Python.h"
02 #include "include\numpy\arrayobject.h"
03
04 double det3(double *arr)
05 {
06   double ret;
07
08   ret = ((arr[0*3 + 0] * (arr[1*3 + 1]*arr[2*3 + 2] - arr[2*3 + 1]*arr[1*3 + 2]))
09        - (arr[0*3 + 1] * (arr[1*3 + 0]*arr[2*3 + 2] - arr[2*3 + 0]*arr[1*3 + 2]))
010       + (arr[0*3 + 2] * (arr[1*3 + 0]*arr[2*3 + 1] -
     arr[2*3 + 0]*arr[1*3 + 1])));
011
012   return ret;
013 }
014
015 int comp_fcn(const void *elem1, const void *elem2)
016 {
017   double e1 = (double) *((double*)elem1 + 2 );
018   double e2 = (double) *((double*)elem2 + 2 );
019   if (e1 > e2) return 1;
020   if (e1 < e2) return -1;
021   return 0;
022 }
023
024 void sort_by_3rd(double *arr, int rowNum, double *out_arr, int colNum)
025 {
026 //  double *out_arr;
027 //  out_arr = (double*) malloc(rowNum*3*sizeof(double));
028   memcpy(out_arr, arr, rowNum*colNum*sizeof(double));
029
030   qsort(out_arr, rowNum, colNum*sizeof(double), comp_fcn);
031
032 }
033
034
035 PyObject* ctrace_reflect(PyObject* self, PyObject* args)
036 {
037   // In args
038   PyObject *argPos_in = NULL, *argDir_in = NULL, *argMesh_in = NULL, *argNormal_in =
     NULL;
039   int lastRefIdx_in;
040   npy_intp *triShape;
041
042   // args
043   PyObject *Pos_in = NULL, *Dir_in = NULL, *Mesh_in = NULL, *Normal_in = NULL;
044   double *Pos_data, *Dir_data, *Mesh_data, *Normal_data;
045
046   PyObject *z_sol;  npy_intp z_sol_shape[1];   double *z_sol_data;
047
048   PyObject *new_Dir = NULL, *new_Pos = NULL;
049   double *new_Dir_data = NULL, *new_Pos_data = NULL;
050
```

```
051   char isNewCollision = 0;  int triNum = 0; double *tri;
052   double M[9];  double Mu[9]; double Mv[9]; double Mt[9];
053   double detM = 0;
054   double sol_uvt[3];
055   double *sol_buffer = NULL, *sorted_sol = NULL;
056   int solCnt = 0, closestIdx = 0;
057   double alpha_norm = 0;
058
059   //ret
060   PyObject *retval = NULL;
061
062   z_sol_shape[0] = 3;
063   if (!PyArg_ParseTuple(args, "OOOOi", &argPos_in, &argDir_in, &argMesh_in, &argNorm
    al_in, &lastRefIdx_in))
064     return NULL;
065
066   Pos_in = PyArray_FROM_OTF(argPos_in, NPY_DOUBLE, NPY_IN_ARRAY);
067   if (Pos_in == NULL) return NULL;
068   Dir_in = PyArray_FROM_OTF(argDir_in, NPY_DOUBLE, NPY_IN_ARRAY);
069   if (Dir_in == NULL) goto fail;
070   Mesh_in = PyArray_FROM_OTF(argMesh_in, NPY_DOUBLE, NPY_IN_ARRAY);
071   if (Mesh_in == NULL) goto fail;
072   Normal_in = PyArray_FROM_OTF(argNormal_in, NPY_DOUBLE, NPY_IN_ARRAY);
073   if (Normal_in == NULL) goto fail;
074   new_Dir = PyArray_SimpleNew(1, PyArray_DIMS(Dir_in), NPY_DOUBLE);
075   new_Pos = PyArray_SimpleNew(1, PyArray_DIMS(Pos_in), NPY_DOUBLE);
076
077   Pos_data = (double*) PyArray_DATA(Pos_in);
078   Dir_data = (double*) PyArray_DATA(Dir_in);
079   Mesh_data = (double*) PyArray_DATA(Mesh_in);
080   Normal_data = (double*) PyArray_DATA(Normal_in);
081   new_Pos_data = (double*) PyArray_DATA(new_Pos);
082   new_Dir_data = (double*) PyArray_DATA(new_Dir);
083
084   triShape = PyArray_DIMS(Mesh_in);
085
086   z_sol = PyArray_SimpleNew(1, z_sol_shape, NPY_DOUBLE);
087   z_sol_data = (double*) PyArray_DATA(z_sol);
088   if (!((Dir_data[2] < 1.0E-05) && (Dir_data[2] > -1.0E-05)))
089   {
090     z_sol_data[0] = Pos_data[0] - Dir_data[0]*(Pos_data[2]/Dir_data[2]);
091     z_sol_data[1] = Pos_data[1] - Dir_data[1]*(Pos_data[2]/Dir_data[2]);
092     z_sol_data[2] = (-1)*(Pos_data[2]/Dir_data[2]);
093   }
094   else
095   {
096     z_sol_data[0] = -1.0;
097     z_sol_data[1] = -1.0;
098     z_sol_data[2] = -1.0;
099   }
0100
0101         for(triNum = 0; triNum < triShape[0]; triNum++)
0102         {
0103           tri = (Mesh_data + 9*triNum);
0104           ////
0105           M[0*3 + 0] = tri[1*3 + 0] - tri[2*3 + 0]; M[0*3 + 1] = tri[0*3 + 0] -
     tri[2*3 + 0]; M[0*3 + 2] = (-1)*Dir_data[0];
0106           M[1*3 + 0] = tri[1*3 + 1] - tri[2*3 + 1]; M[1*3 + 1] = tri[0*3 + 1] -
     tri[2*3 + 1]; M[1*3 + 2] = (-1)*Dir_data[1];
0107           M[2*3 + 0] = tri[1*3 + 2] - tri[2*3 + 2]; M[2*3 + 1] = tri[0*3 + 2] -
     tri[2*3 + 2]; M[2*3 + 2] = (-1)*Dir_data[2];
0108           ////
0109           detM = det3(M);
0110           if (!((detM < 1.0E-05) && (detM > -1.0E-05)))
0111           {
0112             ////replace column 0 in M by constant b
```

```
0113            Mu[0*3 + 0] = Pos_data[0] - tri[2*3 + 0]; Mu[0*3 + 1] = tri[0*3 + 0] -
    tri[2*3 + 0]; Mu[0*3 + 2] = (-1)*Dir_data[0];
0114            Mu[1*3 + 0] = Pos_data[1] - tri[2*3 + 1]; Mu[1*3 + 1] = tri[0*3 + 1] -
    tri[2*3 + 1]; Mu[1*3 + 2] = (-1)*Dir_data[1];
0115            Mu[2*3 + 0] = Pos_data[2] - tri[2*3 + 2]; Mu[2*3 + 1] = tri[0*3 + 2] -
    tri[2*3 + 2]; Mu[2*3 + 2] = (-1)*Dir_data[2];
0116            ////
0117            ////replace column 1 in M by constant b
0118            Mv[0*3 + 0] = tri[1*3 + 0] - tri[2*3 + 0]; Mv[0*3 + 1] = Pos_data[0] -
    tri[2*3 + 0]; Mv[0*3 + 2] = (-1)*Dir_data[0];
0119            Mv[1*3 + 0] = tri[1*3 + 1] - tri[2*3 + 1]; Mv[1*3 + 1] = Pos_data[1] -
    tri[2*3 + 1]; Mv[1*3 + 2] = (-1)*Dir_data[1];
0120            Mv[2*3 + 0] = tri[1*3 + 2] - tri[2*3 + 2]; Mv[2*3 + 1] = Pos_data[2] -
    tri[2*3 + 2]; Mv[2*3 + 2] = (-1)*Dir_data[2];
0121            ////
0122            ////replace column 2 in M by constant b
0123            Mt[0*3 + 0] = tri[1*3 + 0] - tri[2*3 + 0]; Mt[0*3 + 1] = tri[0*3 + 0] -
    tri[2*3 + 0]; Mt[0*3 + 2] = Pos_data[0] - tri[2*3 + 0];
0124            Mt[1*3 + 0] = tri[1*3 + 1] - tri[2*3 + 1]; Mt[1*3 + 1] = tri[0*3 + 1] -
    tri[2*3 + 1]; Mt[1*3 + 2] = Pos_data[1] - tri[2*3 + 1];
0125            Mt[2*3 + 0] = tri[1*3 + 2] - tri[2*3 + 2]; Mt[2*3 + 1] = tri[0*3 + 2] -
    tri[2*3 + 2]; Mt[2*3 + 2] = Pos_data[2] - tri[2*3 + 2];
0126            ////
0127            sol_uvt[0] = det3(Mu)/detM;  //solve for u
0128            sol_uvt[1] = det3(Mv)/detM;  //solve for v
0129            sol_uvt[2] = det3(Mt)/detM;  //solve for t
0130
0131            if ((sol_uvt[2] > 0) && (triNum != lastRefIdx_in) && (0 <= sol_uvt[0])

0132                && (sol_uvt[0] <= 1) && (0 <= sol_uvt[1]) && (sol_uvt[1] <= 1)
0133                && ((sol_uvt[0] + sol_uvt[1]) <= 1))
0134            {
0135              isNewCollision = 1;
0136              solCnt++;
0137              sol_buffer = (double*) realloc(sol_buffer, solCnt*4*sizeof(double));

0138              sol_buffer[4*(solCnt-1) + 0] = sol_uvt[0]; //u
0139              sol_buffer[4*(solCnt-1) + 1] = sol_uvt[1]; //v
0140              sol_buffer[4*(solCnt-1) + 2] = sol_uvt[2]; //t
0141              sol_buffer[4*(solCnt-
    1) + 3] = (double) (triNum); //for later lut use
0142            }
0143          }
0144        }
0145
0146        sorted_sol = (double*) malloc(solCnt * 4 * sizeof(double));
0147        sort_by_3rd(sol_buffer, solCnt, sorted_sol, 4);
0148        closestIdx = (int) (sorted_sol[4*0 + 3] + 0.4);
0149        tri = (Mesh_data + 9*closestIdx);
0150
0151        if (isNewCollision && ((z_sol_data[2] > sorted_sol[2]) || (z_sol_data[2] <=
    0)))
0152        {
0153          // found a coliding triangle and triangle appears before z=0 plane
0154          new_Pos_data[0] = tri[2*3 + 0] + (tri[1*3 + 0] -
    tri[2*3 + 0])*sorted_sol[0] + (tri[0*3 + 0] - tri[2*3 + 0])*sorted_sol[1];
0155          new_Pos_data[1] = tri[2*3 + 1] + (tri[1*3 + 1] -
    tri[2*3 + 1])*sorted_sol[0] + (tri[0*3 + 1] - tri[2*3 + 1])*sorted_sol[1];
0156          new_Pos_data[2] = tri[2*3 + 2] + (tri[1*3 + 2] -
    tri[2*3 + 2])*sorted_sol[0] + (tri[0*3 + 2] - tri[2*3 + 2])*sorted_sol[1];
0157
0158          alpha_norm = 2.0*((Normal_data[closestIdx*3 + 0]*Dir_data[0] + Normal_dat
    a[closestIdx*3 + 1]*Dir_data[1] + Normal_data[closestIdx*3 + 2]*Dir_data[2])
0159                            /(Normal_data[closestIdx*3 + 0]*Normal_data[closestIdx*3
    + 0] + Normal_data[closestIdx*3 + 1]*Normal_data[closestIdx*3 + 1] + Normal_data[clo
    sestIdx*3 + 2]*Normal_data[closestIdx*3 + 2]));
```

```
0160
0161          new_Dir_data[0] = Dir_data[0] -
    (alpha_norm*Normal_data[closestIdx*3 + 0]);
0162          new_Dir_data[1] = Dir_data[1] -
    (alpha_norm*Normal_data[closestIdx*3 + 1]);
0163          new_Dir_data[2] = Dir_data[2] -
    (alpha_norm*Normal_data[closestIdx*3 + 2]);
0164
0165          retval = Py_BuildValue("OOi", new_Pos, new_Dir, closestIdx);
0166          Py_DECREF(new_Pos);
0167          Py_DECREF(new_Dir);
0168          Py_DECREF(z_sol);
0169          Py_DECREF(Pos_in);
0170          Py_DECREF(Dir_in);
0171          Py_DECREF(Mesh_in);
0172          Py_DECREF(Normal_in);
0173          free(sorted_sol);
0174          free(sol_buffer);
0175
0176          return retval;
0177        }
0178        else if (z_sol_data[2] > 0)
0179        {
0180          new_Pos_data[0] = z_sol_data[0];
0181          new_Pos_data[1] = z_sol_data[1];
0182          new_Pos_data[2] = 0.0;
0183
0184          new_Dir_data[0] = Dir_data[0];
0185          new_Dir_data[1] = Dir_data[1];
0186          new_Dir_data[2] = (-1)*Dir_data[2];
0187
0188          retval = Py_BuildValue("OOi", new_Pos, new_Dir, *(PyArray_DIMS(Mesh_in)))
    ;
0189          Py_DECREF(new_Pos);
0190          Py_DECREF(new_Dir);
0191          Py_DECREF(z_sol);
0192          Py_DECREF(Pos_in);
0193          Py_DECREF(Dir_in);
0194          Py_DECREF(Mesh_in);
0195          Py_DECREF(Normal_in);
0196          free(sorted_sol);
0197          free(sol_buffer);
0198
0199          return retval;
0200        }
0201        else
0202        {
0203          Py_DECREF(new_Pos);
0204          Py_DECREF(new_Dir);
0205          Py_DECREF(z_sol);
0206          Py_DECREF(Pos_in);
0207          Py_DECREF(Dir_in);
0208          Py_DECREF(Mesh_in);
0209          Py_DECREF(Normal_in);
0210          free(sorted_sol);
0211          free(sol_buffer);
0212          Py_INCREF(Py_None);
0213
0214          return Py_None;
0215        }
0216
0217
0218
0219        fail:
0220        Py_XDECREF(Pos_in);
0221        Py_XDECREF(Dir_in);
```

```
0222          Py_XDECREF(Mesh_in);
0223          Py_XDECREF(Normal_in);
0224          return NULL;
0225
0226
0227        }
0228
0229
0230
0231        static PyMethodDef globalMethodsTable[] =
0232        {
0233          /*{"PythonFuncName", cfunction,
0234            ARGS_MODIFIERS,
0235            "Doc String"},*/
0236            {"reflect", ctrace_reflect,
0237             METH_VARARGS,
0238             "reflect(argPos_in, argDir_in, argMesh_in, argNormal_in, lastRefIdx_in)
    -> retTuple\n retTuple = (newPos, newDir, lastrefidx)"},
0239            {NULL, NULL, 0, NULL}
0240        };
0241
0242        PyMODINIT_FUNC initctrace(void)
0243        {
0244          (void) Py_InitModule("ctrace", globalMethodsTable);
0245          import_array();
0246        }
```

Na początku programu importujemy biblioteki API Pythona i *numpy*. Pierwsza funkcja, `double det3(double *arr)`, przyjmuje wskaźnik na zmienną typu `double`, a zajmuje się obliczaniem wyznacznika macierzy 3x3. Jako, że tylko wyznacznika takiej macierzy potrzeba, funkcja ta oblicza go korzystając z gotowego wzoru. Implementacja ogólnej metody obliczania wyznacznika jest niepotrzebna, a wręcz w tym przypadku byłaby niewskazana. Jakiekolwiek pętle i rekurencje nie mają szans równać się z szybkością formuły wprowadzonej jako zwykłe operacje arytmetyczne.

Funkcja `int comp_fcn(const void *elem1, const void *elem2)` jest funkcją porównującą dwa wektory (wskaźniki na wiersze). W implementacji algorytmu odbicia okaże się, że potrzebna będzie funkcja, która sortuje wiersze macierzy po drugiej (licząc od zera) kolumnie. Funkcja ta porównuje dwa wiersze, porównując ich drugie (licząc od zera) elementy.

Funkcja `void sort_by_3rd(double *arr, int rowNum, double *out_arr, int colNum)` sortuje tablicę o wymiarach `rowNum x colNum` po drugiej kolumnie (licząc od zera) i zapisuje wynik w zmiennej `out_arr`. W dalszej części tego rozdziału wszystkie indeksy będą w domyśle liczone od zera.

Funkcja `PyObject* ctrace_reflect(PyObject* self, PyObject* args)` jest główną funkcją modułu. To ona porozumiewa się z Pythonem poprzez jego API, oraz to ona wykonuje obliczenia niezbędne do przeprowadzenia odbicia. Argumenty i typ wartości wyjściowej są typowe dla standardu API Pythona. Więcej o tym czym są wskaźniki na struktury `PyObject` można przeczytać w dokumentacji Pythona [12]. Nam wystarczy myśleć o nich jak o obiektach Pythonowych, które przekazywane są w postaci wskaźników.

W liniach `037-057` przeprowadzane są niezbędne deklaracje zmiennych. W liniach `057-087` dokonywane jest parsowanie argumentów i ich konwersja do typów języka C.

W instrukcji warunkowej z linii `088` sprawdzamy, czy wiązka ma szanse przeciąć się z płaszczyzną $z = 0$, sprawdzając czy składowa *z-owa* jej kierunku jest istotnie różna od zera. Jeśli tak, to przypisujemy od razu do zmiennej `z_sol_data` współrzędne tego przecięcia (patrz: rozdział 2.3.).

W pętli z linii `0101` iterujemy po numerze trójkąta w siatce. Najpierw do tablicy `M` przypisujemy współczynniki takie jak w algorytmie opisanym w rozdziale 2.2. Macierz ta jest macierzą główną układu równań na *u,v,t*, przedstawioną we wspomnianym rozdziale. W linii `0109` obliczamy jej wyznacznik, a następnie sprawdzamy czy jest różny od zera. Jeśli tak, to układ posiada rozwiązanie. Następnie, aż do linii `0126` definiujemy macierze z zastąpionymi kolumnami przez prawą stronę układu równań z rozdziału 2.2. Posiadając te macierze, rozwiązanie *u,v,t* przecięcia prosta – trójkąt dane jest przez odpowiednie ilorazy, tak jak opisano we wspomnianym rozdziale. Rozwiązanie to jest przechowywane w kolejności w tablicy `sol_uvt`. Następnie sprawdzane są odpowiednie warunki, czy punkt przecięcia leży w trójkącie. Oprócz tego, sprawdzamy, czy nie próbujemy znaleźć przecięcia z trójkątem, od którego ostatnio nastąpiło odbicie.

Jeżeli warunki są spełnione, to:

- wystawiana jest flaga `isNewCollision`
- zwiększany jest licznik poprawnych rozwiązań `solCnt`
- w liniach `0138-0141` tworzona jest dowiązanie do tzw. listy z dowiązaniami (`sol_buffer`). W języku Python odpowiada to metodzie `list.append()` – na koniec listy `sol_buffer` dodawane jest obecne rozwiązanie, wraz z indeksem trójkąta dla którego to rozwiązanie jest poprawne

Po wyjściu z pętli, w linii `0147` sortujemy listę `sol_buffer` po drugiej kolumnie, czyli po rozwiązaniu *t*. Po posortowaniu, zerowy wiersz listy `sol_buffer` odpowiada takiemu przecięciu z trójkątem, dla którego parametr *t* jest najmniejszy, więc jest to przecięcie z *najbliższym* trójkątem z siatki. Indeks tego trójkąta zapisywany jest w zmiennej `closestIdx`.

W linii `0151` sprawdzamy czy w ogóle trójkąt został znaleziony i czy pojawia się przed płaszczyzną $z = 0$. Jeśli tak (najbardziej typowa sytuacja), to do odpowiednich zmiennych (nowa pozycja początkowa, nowy kierunek, indeks trójkąta od którego nastąpiło odbicie) są wypełniane. Nowy kierunek jest obliczany na podstawie wektora normalnego do danego trójkąta, zgodnie z algorytmem z rozdziału 2.5. Zadania te są wykonywane aż do linii `0163`. W linii `0165` budujemy krotkę wyjściową za pomocą funkcji z API. Następnie odbywa się zarządzanie Pythonowym *Garbage Collectorem* i zwalnianie pamięci zarezerwowanej dla dynamicznie alokowanych zmiennych. W linii `0176` zwracamy krotkę.

Jeżeli omawiane warunki nie zostały spełnione, to sprawdzamy, czy rozwiązanie przecięcia wiązki z płaszczyzną $z = 0$ ma parametr *t* większy od zera (wiązka rzeczywiście biegnie w kierunku tej płaszczyzny). Jeżeli tak, to wiązka musiała się przeciąć tylko i wyłącznie z płaszczyzną $z = 0$ i aktualizujemy nowe pozycje startowe oraz kierunki, oraz zwracamy je w zbudowanej krotce (linia `0199`). Przed tym przeprowadzone zostało zarządzanie *Garbage Collectorem*, dokładnie w ten sam sposób co poprzednio. Zwracamy uwagę, że w tym

przypadku, jako indeks trójkąta od którego nastąpiło odbicie, zwracana jest długość listy trójkątów, czyli wartość o 1 większa niż indeks ostatniego trójkąta.

W przeciwnym wypadku, procedura się nie powiodła, więc zarządzamy *Garbage Collectorem* i zwracamy Pythonowe *None*.

Od linii `0231` do `0240` zajmujemy się interfejsem C – Python, czyli tworzymy tablicę w której rejestrujemy metody modułu. Nasz moduł posiada tylko jedną metodę której interfejs chcemy obsługiwać, więc tylko ją rejestrujemy.

W linii `0242` deklarujemy i definiujemy funkcję inicjalizująca moduł.

Więcej informacji na temat rozszerzeń Pythona w języku C i interfejsu API zarówno samego Pythona, jak i *numpy*, można znaleźć w dokumentacji technicznej Pythona i *numpy* [10,11].

Jak widać, napisanie tego rozszerzenia w C zajęło nam blisko 250 linii kodu. Ta sama funkcjonalność napisana w Pythonie nie zajmowała więcej niż 80 linii kodu. Jest to ponad 3 razy więcej pracy, ale zysk czasowy, który został zbadany na przykładowych danych, jest nieoceniony. Procedura ta, napisana w języku C, wykonywała się zawsze około 1700 razy szybciej niż jej Pythonowy odpowiednik (łącznie z odwołaniem do skompilowanej biblioteki .dll).

Na wprowadzeniu fragmentu w C cierpi jednak przenośność kodu. Moduł napisany w C trzeba skompilować do biblioteki .dll zawsze, gdy zmieniamy system operacyjny, na którym uruchamiamy to oprogramowanie. Proces kompilacji rozszerzeń do Pythona jest skomplikowany, ale częściowo opisany w dokumentacji technicznej [10]. Kompilowanie rozszerzeń pod systemami z rodziny UNIX jest w zasadzie bezproblemowe, jednak pod systemami z rodziny Windows może nastręczyć problemów. Producent Pythona zaleca kompilowanie rozszerzeń przy użyciu kompilatorów firmy Microsoft i oprogramowania Visual Studio. Proces ten jest jednak na tyle skomplikowany, że autor zdecydował się użyć dystrybucji kompilatora GCC pod system Windows. Dostawca Pythona nie gwarantuje w takiej sytuacji, że skompilowane rozszerzenie będzie pracować poprawnie, jednak okazuje się, że w tym konkretnym przypadku (jak i we wielu innych) – pracuje poprawnie.

3.3. Frontend

Za frontend uważamy tę część kodu, która jest przeznaczona do budowania interfejsu człowiek - maszyna. Funkcje i klasy zdefiniowane w module `fwdraytracing.utils` czynią za dość tej definicji. Moduł ten składa się z 9. funkcji oraz z jednej klasy:

```
01 def unique_rows(arr):
02     #...
03
04 def seedDirections(Npoints):
05     #...
06
07 def meshDirections(Npoints):
08     #...
```

```
09
010 def jitter(arr, size = 0.01):
011     #...
012
013 def meshVolume(model, N):
014     #...
015
016 def zeroPoly(model, zero = 0):
017     #...
018
019 def axisEqual3D(ax):
020     #...
021
022 class Scintillator(object):
023     #...
024
025 def plotSurf(model, axes = False, z0 = True, color = 'g', alpha = 1):
026     #...
027
028 def sampleTrace(model, pos, dirs = None, ax = None, fig = None, maxRecursion = 15):

029     #...
030
031
```

Moduł importuje następujące biblioteki:

```
01 import numpy as np
02 from geo import *
03 import matplotlib as mpl
04 import matplotlib.pyplot as plt
05 from mpl_toolkits.mplot3d import Axes3D
```

Jako, że moduł służy za interfejs do funkcjonalności backendu - importuje także moduł `fwdraytracing.geo`.

3.3.1. Funkcja `unique_rows(arr)`

Ciało tej funkcji wygląda następująco:

```
01 def unique_rows(arr):
02     ind = np.lexsort(arr.T)
03     ind = ind[np.concatenate(([True],np.any(np.logical_not(np.isclose(arr[ind[1:]],a
   rr[ind[:-1]])),axis=1)))]
04     return arr[ind].copy()
```

Nie jest ona obszerna, a jest wykorzystywana w pewnym etapie tworzenia bardziej zaawansowanych procedur. Funkcja zwraca tablicę podaną na jej wejście, ale bez duplikatów wierszy.

3.3.2. Funkcja `seedDirections(Npoints)`

Argumentem tej funkcji powinna być liczba całkowita. Służy ona do wylosowania zbioru wektorów o jednostkowej długości, które wskazują na punkty które mają jednostajny rozkład gęstości na sferze.

```
01  def seedDirections(Npoints):
02
03      n = int(np.sqrt(Npoints))
04      phiv = 2*np.pi*np.random.rand(n,n)
05      thetav = np.arccos(2*np.random.rand(n,n) - 1)
06
07      x_table = (np.sin(thetav)*np.cos(phiv)).flatten()
08      y_table = (np.sin(thetav)*np.sin(phiv)).flatten()
09      z_table = np.cos(thetav).flatten()
010
011     return np.array([x_table, y_table, z_table]).T.copy()
```

Na początku, funkcja pierwiastkuje swoje wejście i wynik zaokrągla do najbliższej mniejszej liczby całkowitej. Dopiero kwadrat *tej* liczby będzie faktyczną liczbą zasianych punktów. Następnie tworzone są tablice dwuwymiarowe, zgodnie z procedurą opisaną w rozdziale 2.6. Tak przeskalowane liczby losowe służą do obliczenia wartości współrzędnych *x*, *y*, *z*, które są parametryzacją sfery. We wspomnianym rozdziale udowodniliśmy, że takie skalowanie zmiennych daje w wyniku rozkład jednostajny. Zwracana jest tablica, której wiersze są wektorami o rozkładzie jednostajnym na sferze.

3.3.3. Funkcja `meshDirections(Npoints)`

Funkcja ta spełnia bardzo podobną funkcjonalność do omawianej przed chwilą, kod prezentuje się następująco:

```
01  def meshDirections(Npoints):
02
03      n = int(np.sqrt(Npoints))
04      phi = 2*np.pi*np.linspace(0,1,n)
05      theta = np.arccos(2*np.linspace(0.001,1-0.001,n) - 1)
06      phiv, thetav = np.meshgrid(phi,theta)
07      x_table = (np.sin(thetav)*np.cos(phiv)).flatten()
08      y_table = (np.sin(thetav)*np.sin(phiv)).flatten()
09      z_table = np.cos(thetav).flatten()
010
011     return unique_rows(np.array([x_table, y_table, z_table]).T)
```

Od poprzedniej funkcji różni się ona tym, że przedziały *u*,*v* nie są losowane, a rozpinane jednostajnie. Oprócz tego z wyjściowej tablicy usuwane są te wierze, które wskazują na jednakowe punkty. Wyjście z tej funkcji jest zawsze przewidywalne, natomiast z poprzedniej niekoniecznie. Do renderowania obrazu jednak lepiej wykorzystywać funkcję poprzednią.

3.3.4. Funkcja `jitter(arr, size = 0.01)`

Wykorzystanie tej funkcji w naszym kodzie nie jest zbyt chwalebnym posunięciem, ale było konieczne ze względu na pewne bardzo subtelne problemy natury numerycznej.

```
1.  def jitter(arr, size = 0.01):
2.
3.      return arr + (np.random.rand(*(arr.shape)) - 0.5)*size
```

Jak widać, funkcja ta jest niezwykle prosta. Jedyne co robi, to do każdego elementu wejściowej tablicy dodaje małą liczbę losową z przedziału `-size/2, size/w`. Wykorzystujemy ją tylko w jednym miejscu – przy generowaniu siatki punktów leżących wewnątrz kryształu, które są bazą do całkowania po jego objętości. Okazywało się, że dla deterministycznie wygenerowanych punktów występowały problemy w kryterium ich obecności wewnątrz kryształu. Świadczy to o tym, że użyte kryterium nie było do końca poprawne, albo jego implementacja nie została wykonana poprawnie. Problem jest aż tak subtelny, że autor do tej pory go nie rozwiązał. Ten jeden fakt świadczy już o tym, że prezentowany kod to dopiero wczesna wersja beta oprogramowania docelowego.

Mimo wszystko, wprowadzenie *jittera* do generowanych punktów siatki pozwala uniknąć tych problemów. Jego wprowadzenie nie wpływa na dokładność całkowania numerycznego.

3.3.5. Funkcja `meshVolume(model, N)`

Funkcja ta jest tą wspomnianą przed chwilą. Zajmuje się wygenerowaniem punktów znajdujących się wewnątrz kryształu.

```
01  def meshVolume(model, N):
02
03      n = N
04      zmax = np.max(model.mesh[:,:,2])
05      zmesh = np.linspace(0,zmax,n+1, endpoint = False)[1:]
06
07      x_max = np.max(model.mesh[:,:,0])
08      x_min = np.min(model.mesh[:,:,0])
09      y_max = np.max(model.mesh[:,:,1])
010     y_min = np.min(model.mesh[:,:,1])
011
012     xseed = np.linspace(x_min, x_max, n)
013     yseed = np.linspace(y_min, y_max, n)
014
015     xyseed = jitter(np.transpose([np.tile(xseed, len(yseed)), np.repeat(yseed, len(x
    seed))]))
016
017     globalMesh = np.array([[0,0,0]])
018
019     for zi in zmesh:
020         lines = []
021         for i in xrange(len(model.mesh)):
022             u0a = (zi - model.mesh[i,2,2])/(model.mesh[i,1,2] - model.mesh[i,2,2])
023             v0b = (zi - model.mesh[i,2,2])/(model.mesh[i,0,2] - model.mesh[i,2,2])
024             uc =  (model.mesh[i,0,2] - zi)/(model.mesh[i,0,2] - model.mesh[i,1,2])
025             vc =  (zi - model.mesh[i,1,2])/(model.mesh[i,0,2] - model.mesh[i,1,2])
026
027             testA = 0 <= u0a <= 1
028             testB = 0 <= v0b <= 1
029             testC = (0 <= uc <= 1) and (0 <= vc <= 1)
030
031             #swt = np.array([testA,testB,testC]).dot([0,2,4])
032
033             if (testA and testB): #crossing u and v axis
034                 x1 = model.mesh[i,2,0] + (model.mesh[i,1,0]-model.mesh[i,2,0])*u0a
035                 y1 = model.mesh[i,2,1] + (model.mesh[i,1,1]-model.mesh[i,2,1])*u0a
```

```
036                 x2 = model.mesh[i,2,0] + (model.mesh[i,0,0]-model.mesh[i,2,0])*v0b
037                 y2 = model.mesh[i,2,1] + (model.mesh[i,0,1]-model.mesh[i,2,1])*v0b
038                 p = np.array([[x1, y1, zi],
039                               [x2, y2, zi]])
040                 lines.append(p)
041
042             elif (testA and testC): #crossing u and line
043                 x1 = model.mesh[i,2,0] + (model.mesh[i,1,0]-model.mesh[i,2,0])*u0a
044                 y1 = model.mesh[i,2,1] + (model.mesh[i,1,1]-model.mesh[i,2,1])*u0a
045                 x2 = model.mesh[i,2,0] + (model.mesh[i,1,0]-
    model.mesh[i,2,0])*uc + (model.mesh[i,0,0]-model.mesh[i,2,0])*vc
046                 y2 = model.mesh[i,2,1] + (model.mesh[i,1,1]-
    model.mesh[i,2,1])*uc + (model.mesh[i,0,1]-model.mesh[i,2,1])*vc
047                 p = np.array([[x1, y1, zi],
048                               [x2, y2, zi]])
049                 lines.append(p)
050
051             elif (testB and testC): #crossing v and line
052                 x1 = model.mesh[i,2,0] + (model.mesh[i,1,0]-
    model.mesh[i,2,0])*uc + (model.mesh[i,0,0]-model.mesh[i,2,0])*vc
053                 y1 = model.mesh[i,2,1] + (model.mesh[i,1,1]-
    model.mesh[i,2,1])*uc + (model.mesh[i,0,1]-model.mesh[i,2,1])*vc
054                 x2 = model.mesh[i,2,0] + (model.mesh[i,0,0]-model.mesh[i,2,0])*v0b
055                 y2 = model.mesh[i,2,1] + (model.mesh[i,0,1]-model.mesh[i,2,1])*v0b
056                 p = np.array([[x1, y1, zi],
057                               [x2, y2, zi]])
058                 lines.append(p)
059
060             else:
061                 pass
062
063
064         #end for i############################################################
065         ######## lines now contains polygons
066         npLines = np.array(lines)
067
068         insideIdx = []
069
070         for j in xrange(len(xyseed)):
071             point = xyseed[j,:]
072             crossCnt = 0
073             for i in xrange(len(npLines)):
074                 tLine = (point[1] - npLines[i,1,1])/(npLines[i,0,1]-
    npLines[i,1,1])
075
076                 tp = npLines[i,1,0] + (npLines[i,0,0] -
     npLines[i,1,0])*tLine + point[0]
077
078                 if ((0 <= tp < np.inf) and ((0 < tLine <= 1) or np.isclose(tLine,0)
    or np.isclose(tLine,1))): #half-line (tp>0) and line section crossed (0<<1)
079                     crossCnt += 1
080
081             if (crossCnt % 2) == 1: #point is inside polygon
082                 insideIdx.append(j)
083
084         insidePts = xyseed[insideIdx]
085         ziPts = np.concatenate((insidePts,np.repeat(np.array([[zi]]),len(insidePts),
     axis = 0)), axis = 1)
086
087         globalMesh = np.concatenate((globalMesh, ziPts), axis = 0)
088
089     #end for zi
090     globalMesh = globalMesh[1:,:]
091
092     return globalMesh
```

Koncepcja stojąca za przedstawionym algorytmem jest następująca:

- Prostopadłościan o wymiarach takich, że zawiera wewnątrz siebie nietrywialny brzeg kryształu - przeciąć poziomymi ($z = const$) płaszczyznami.
- Na każdej z płaszczyzn wygenerować siatkę punktów które dzielą przedział x i y na równe części.
- Dla danej płaszczyzny znaleźć jej przecięcie z nietrywialnym brzegiem (będzie ono wielokątem) – wykorzystując metody z rozdziału 2.3.
- Dla każdego punktu na tej płaszczyźnie sprawdzić, czy znajduje się on wewnątrz wygenerowanego wielokąta (twierdzenie o krzywej Jordana, rozdział 2.4.). Jeżeli znajduje się, to dodać go do globalnej listy poprawnych punktów.
- Powtórzyć dla każdej płaszczyzny i każdego punktu na tej płaszczyźnie.

Argument `model` musi być obiektem klasy `class SurfaceModel(object)`. Argument `N` musi być liczbą całkowitą i mówi na ile punktów zostanie podzielony każdy z wymiarów wspomnianego wyżej prostopadłościanu. Wobec tego całkowita liczba wygenerowanych punktów będzie proporcjonalna do trzeciej potęgi `N`. Natomiast liczba punktów wygenerowanych wewnątrz samego kryształu, będzie proporcjonalna do trzeciej potęgi `N` ale także do stosunku jego objętości i objętości prostopadłościanu.

W liniach `04-013` wyliczamy rozmiary najmniejszego prostopadłościanu i rozpinamy na nim punkty. W następnej linii tworzymy tablicę współrzędnych xy i aplikujemy na niej funkcję małego jittera. Zmienną `globalMesh` inicjalizujemy zerami. Będzie ona służyć zebraniu wszystkich punktów siatki. Główna pętla funkcji rozpoczyna się w linii `019`. Iterujemy w niej po współrzędnych z płaszczyzn którymi przecinamy nietrywialny brzeg kryształu. Wewnątrz deklarujemy zmienną `lines` jako listę i rozpoczynamy kolejną pętlę w której iterujemy po trójkątach siatki. Od linii `022` do linii `025` rozwiązujemy zagadnienie opisane w rozdziale 2.3. – czyli przecięcie płaszczyzna trójkąt (po których to właśnie iterujemy). Zmienne `testX` posłużą nam do rozstrzygnięcia czy płaszczyzna w ogóle przecina się z aktualnie rozważanym trójkątem i jeśli tak, to jakie są współrzędne tych przecięć. Instrukcja sterująca `if..elif..else` rozpoczynająca się w linii `033` służy temu rozstrzygnięciu oraz wyliczeniu punktów przecięcia. Jeżeli tylko przecięcie następuje, to współrzędne dwóch punktów przecięcia są dopisywane do listy `lines`. W ten sposób reprezentujemy krzywą łamaną – poprzez listę par punktów, które składają się na jej odcinki. Podejście to może nie być najbezpieczniejsze i prawdopodobnie może rodzić niepożądane konsekwencje, natomiast w obecnej wersji oprogramowania, która nie jest wersją finalną, arbitralnie podjęto decyzję o takiej właśnie reprezentacji.

W linii `070` dokonaliśmy już iteracji po wszystkich trójkątach i rozpoczynamy kolejną pętlę – po indeksie punktu z siatki xy rozpiętej na wysokości zi. Wewnątrz niej iterujemy po kolejnych fragmentach krzywej łamanej i sprawdzamy, czy aktualnie rozważany punkt z siatki xy leży wewnątrz niej – na podstawie *Twierdzenia o punkcie wewnątrz krzywej* z rozdziału 2.4., gdzie za wektor $\begin{bmatrix} a \\ b \end{bmatrix}$ przyjęliśmy $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Kluczowe sprawdzenie, ile razy testowa półprosta przecina się z krzywą odbywa się w linii `081`. Wtedy też dodajemy do listy `insideIdx`

indeks punktu który spełnia warunki przecięcia. W następnej linii wybieramy z listy punktów siatki *xy* te punkty, które rzeczywiście leżą wewnątrz krzywej łamanej, a więc wewnątrz brzegu kryształu. Kolejne linie to już tylko kroki konieczne do odpowiedniego sformatowania danych i dodania ich do tablicy punktów tworzących siatkę. Tablica ta jest zwracana.

3.3.6. Funkcja `zeroPoly(model, zero = 0)`

```
01  def zeroPoly(model, zero = 0):
02      zi = zero
03      lines = []
04      for i in xrange(len(model.mesh)):
05          u0a = (zi - model.mesh[i,2,2])/(model.mesh[i,1,2] - model.mesh[i,2,2])
06          v0b = (zi - model.mesh[i,2,2])/(model.mesh[i,0,2] - model.mesh[i,2,2])
07          uc =  (model.mesh[i,0,2] - zi)/(model.mesh[i,0,2] - model.mesh[i,1,2])
08          vc =  (zi - model.mesh[i,1,2])/(model.mesh[i,0,2] - model.mesh[i,1,2])
09
010         testA = 0 <= u0a <= 1
011         testB = 0 <= v0b <= 1
012         testC = (0 <= uc <= 1) and (0 <= vc <= 1)
013
014         #swt = np.array([testA,testB,testC]).dot([0,2,4])
015
016         if (testA and testB): #crossing u and v axis
017             x1 = model.mesh[i,2,0] + (model.mesh[i,1,0]-model.mesh[i,2,0])*u0a
018             y1 = model.mesh[i,2,1] + (model.mesh[i,1,1]-model.mesh[i,2,1])*u0a
019             x2 = model.mesh[i,2,0] + (model.mesh[i,0,0]-model.mesh[i,2,0])*v0b
020             y2 = model.mesh[i,2,1] + (model.mesh[i,0,1]-model.mesh[i,2,1])*v0b
021             p = np.array([[x1, y1, zi],
022                           [x2, y2, zi]])
023             lines.append(p)
024
025         elif (testA and testC): #crossing u and line
026             x1 = model.mesh[i,2,0] + (model.mesh[i,1,0]-model.mesh[i,2,0])*u0a
027             y1 = model.mesh[i,2,1] + (model.mesh[i,1,1]-model.mesh[i,2,1])*u0a
028             x2 = model.mesh[i,2,0] + (model.mesh[i,1,0]-
    model.mesh[i,2,0])*uc + (model.mesh[i,0,0]-model.mesh[i,2,0])*vc
029             y2 = model.mesh[i,2,1] + (model.mesh[i,1,1]-
    model.mesh[i,2,1])*uc + (model.mesh[i,0,1]-model.mesh[i,2,1])*vc
030             p = np.array([[x1, y1, zi],
031                           [x2, y2, zi]])
032             lines.append(p)
033
034         elif (testB and testC): #crossing v and line
035             x1 = model.mesh[i,2,0] + (model.mesh[i,1,0]-
    model.mesh[i,2,0])*uc + (model.mesh[i,0,0]-model.mesh[i,2,0])*vc
036             y1 = model.mesh[i,2,1] + (model.mesh[i,1,1]-
    model.mesh[i,2,1])*uc + (model.mesh[i,0,1]-model.mesh[i,2,1])*vc
037             x2 = model.mesh[i,2,0] + (model.mesh[i,0,0]-model.mesh[i,2,0])*v0b
038             y2 = model.mesh[i,2,1] + (model.mesh[i,0,1]-model.mesh[i,2,1])*v0b
039             p = np.array([[x1, y1, zi],
040                           [x2, y2, zi]])
041             lines.append(p)
042
043         else:
044             pass
045
046     npLines = np.array(lines)
047
048     return npLines
```

Do funkcji przekazujemy obiekt klasy `class SurfaceModel(object)` jako `model` oraz liczbę rzeczywistą jako `zero`. Dla podanego obiektu reprezentującego numeryczny model nietrywialnego brzegu kryształu funkcja zwraca listę par punktów które tworzą odcinki krzywej łamanej będącej rozwiązaniem problemu przecięcia się płaszczyzny poziomej umieszczonej na wysokości `zero`. Metoda i jej implementacja w tym przypadku jest dokładnie taka sama jak ta zaprezentowana w poprzednim akapicie, w związku z czym nie widzimy potrzeby jej opisywania.

3.3.7. Funkcja `axisEqual3D(ax)`

```
01 def axisEqual3D(ax):
02     extents = np.array([getattr(ax, 'get_{}lim'.format(dim))() for dim in 'xyz'])
03     sz = extents[:,1] - extents[:,0]
04     centers = np.mean(extents, axis=1) #[0,0,0]
05     maxsize = max(abs(sz))
06     r = maxsize/2
07     for ctr, dim in zip(centers, 'xyz'):
08         getattr(ax, 'set_{}lim'.format(dim))(ctr - r, ctr + r)
```

Funkcja ta służy do poprawienia funkcjonalności oferowanej przez biblioteki pakietu *matplotlib*. Okazuje się, że pakiet ten ma problemy z ustaleniem proporcji osi gdy tworzy wykresy trójwymiarowe. W naszym przypadku, zachowanie proporcji osi ma kluczowe znaczenie, dlatego została wprowadzona ta funkcja. Opisywanie mechanizmów użytych w jej ciele nie ma sensu, przynajmniej nie w kontekście tej pracy. Wystarczy jedynie wiedzieć, że funkcja przyjmuje jako argument obiekt klasy *axes* używany przez *matplotlib* do reprezentacji wykresu i modyfikuje go tak, by proporcje osi zostały zachowane.

3.3.8. Klasa `class Scintillator(object)`

Z punktu widzenia użytkownika, klasa ta będzie jedną z najczęściej używanych, gdyż służy do abstrakcyjnej reprezentacji kryształu scyntylacyjnego.

```
01 class Scintillator(object):
02     def __init__(self, param_def=None, shape_def=None, ranges=None, precision_cnt =
   40, alpha = 1, R=1):
03         if not ((param_def is None) or (shape_def is None) or (ranges is None) or (p
   recision_cnt is None)):
04             self.bound = DifferentialManifold(param_def, shape_def, ranges)
05             self.surface = SurfaceModel(precision_cnt)
06             self.surface.addSheet(self.bound)
07             self.surface.bakeMesh()
08 
09         self.alpha = alpha
010        self.R = R
011
012        self.globalMesh = None
013        self._gMeshSpatPrec = 0
014        self._gMeshDirPrec = 0
015
016        #self.horzMap = []
017        self.beta = None
018
019    def loadRaw(self, raw):
020        S = SurfaceModel(int(np.sqrt(len(raw))))
021        S.fromRawMesh(raw)
022        self.surface = S
```

```
023
024
025     def lightYield(self, spatial_prec=15, direct_prec=100, maxref=20):
026         if self.globalMesh is None:
027             self.globalMesh = meshVolume(self.surface, spatial_prec)
028             self._gMeshSpatPrec = spatial_prec
029             self._gMeshDirPrec = direct_prec
030         elif ((self._gMeshSpatPrec != spatial_prec) or (self._gMeshDirPrec != direct
    _prec)):
031             self.globalMesh = meshVolume(self.surface, spatial_prec)
032             self._gMeshSpatPrec = spatial_prec
033             self._gMeshDirPrec = direct_prec
034
035         phasePointsCnt = len(self.globalMesh) * len(seedDirections(self._gMeshDirPre
    c))
036         lengths = np.zeros(phasePointsCnt)
037         reflections = np.zeros(phasePointsCnt)
038
039
040
041
042         lastEntry = 0;
043         j = 0
044         for initPos in self.globalMesh:
045             initDirs = seedDirections(self._gMeshDirPrec)
046             j += 1
047             for initDir in initDirs:
048                 beam = Ray(initPos, initDir)
049                 beam.placeEnviorment(self.surface)
050                 beam.trace(maxref)
051                 if beam.reflectCnt != maxref:
052                     #self.horzMap.append(beam.path[beam.reflectCnt+1])
053                     lengths[lastEntry] = beam.pathLength()
054                     reflections[lastEntry] = beam.reflectCnt
055                     lastEntry += 1
056
057         self.beta = np.mean(np.exp((-
    1)*self.alpha*lengths[:lastEntry+1])*(self.R**reflections[:lastEntry+1]))
058
059         return self.beta
060
061
062     def screenMap(self, spatial_prec=15, direct_prec=100, maxref=20, pixels = 20, po
    intSource = None, dirSeed = True, name = 'Screen intensity map'):
063         if self.globalMesh is None:
064             self.globalMesh = meshVolume(self.surface, spatial_prec)
065             self._gMeshSpatPrec = spatial_prec
066             self._gMeshDirPrec = direct_prec
067         elif ((self._gMeshSpatPrec != spatial_prec) or (self._gMeshDirPrec != direct
    _prec)):
068             self.globalMesh = meshVolume(self.surface, spatial_prec)
069             self._gMeshSpatPrec = spatial_prec
070             self._gMeshDirPrec = direct_prec
071
072         if (pointSource is None):
073             mesh = self.globalMesh
074         else:
075             mesh = pointSource
076
077         phasePointsCnt = len(mesh) * len(seedDirections(self._gMeshDirPrec))
078         xy_hist = np.zeros((phasePointsCnt,3))
079         weights = np.zeros(phasePointsCnt)
080
081         zeroBound = zeroPoly(self.surface)
082         rmin = np.min(zeroBound[:,:,0:2])
083         rmax = np.max(zeroBound[:,:,0:2])
```

```
084         ranges = [[rmin, rmax],
085                   [rmin, rmax]]
086 
087         lastEntry = 0;
088 
089         for initPos in mesh:
090             initDirs = seedDirections(self._gMeshDirPrec) if dirSeed else meshDirect
    ions(self._gMeshDirPrec)
091 
092             for initDir in initDirs:
093                 beam = Ray(initPos, initDir)
094                 beam.placeEnviorment(self.surface)
095                 beam.trace(maxref)
096                 if beam.reflectCnt != maxref:
097                     xy_hist[lastEntry,0] = beam.path[beam.reflectCnt+1, 0]
098                     xy_hist[lastEntry,1] = beam.path[beam.reflectCnt+1, 1]
099                     xy_hist[lastEntry,2] = beam.path[beam.reflectCnt+1, 2]
0100                        weights[lastEntry] = np.exp((-
    1)*self.alpha*beam.pathLength())*(self.R**(beam.reflectCnt))
0101                        lastEntry += 1
0102             self.H, self.xedges, self.yedges = np.histogram2d(xy_hist[:(lastEntry
    +1),0], xy_hist[:(lastEntry+1),1], weights = weights[:(lastEntry+1)],
0103                                                                bins = pixels, range
    = ranges)
0104             #self.H = self.H.T
0105 
0106             fig = plt.figure()
0107             fig.canvas.set_window_title(name)
0108             ax = fig.add_subplot(111, title = name)
0109             ax.set_aspect('equal')
0110             plt.imshow(self.H, interpolation = 'bilinear', origin = 'low',
0111                        extent = [self.xedges[0], self.xedges[-
    1], self.yedges[0], self.yedges[-1]])
0112             plt.colorbar()
0113             return fig, ax
0114 
```

Konstruktor obiektu, metoda:

`__init__(self, param_def=None, shape_def=None, ranges=None, precision_cnt = 40, alpha = 1, R=1)`

Przyjmuje szereg argumentów:

- `param_def` – jest to *string* przekazywany później do klasy `class DifferentialManifold(object)`. Odpowiada on argumentowi konstruktora tej klasy o takiej samej nazwie, czyli dokładniej – są to wymienione nazwy argumentów parametryzacji oddzielone białym znakiem.
- `shape_def` – jest to *string* przekazywany później do klasy `class DifferentialManifold(object)`. Odpowiada on argumentowi konstruktora tej klasy o takiej samej nazwie, czyli dokładniej – jest symbolicznym zapisem parametryzacji nietrywialnego brzegu.
- `ranges` - jest to *krotka krotek*, przekazywana później do klasy `class DifferentialManifold(object)`. Odpowiada on argumentowi konstruktora tej klasy o takiej samej nazwie, czyli dokładniej – reprezentuje ona zakresy zmienności argumentów.

- `precision_cnt` – ten argument z kolei jest przekazywany do konstruktora klasy `class SurfaceModel(object)`, a mówi o tym na ile punktów zostanie podzielony przedział zmienności każdego z argumentów parametryzacji.
- `alpha` – współczynnik pochłaniania z prawa Lamberta-Beera
- `R` – współczynnik odbicia

Na początku sprawdzamy, czy wszystkie kluczowe argumenty zostały podane. Jeśli nie, to zakładamy, że model kryształu ma być zaimportowany z pliku poprzez metodę `loadRaw(self, raw)`. W konstruktorze inicjalizujemy także pewne zmienne.

Metoda `loadRaw(self, raw)` służy zaimportowaniu modelu kryształu z zewnątrz. W argumencie `raw` podawana jest tablica zawierająca współrzędne wierzchołków nietrywialnego brzegu kryształu, na przykład taka, jaka jest generowana przez funkcje `primitives(prim, params)`. Mechanizm ładowania i zapisywania nie jest jeszcze w pełni dopracowany i nie zalecamy korzystania z tej funkcji. Jest ona obecna głownie ze względu na kompatybilność.

W metodzie `lightYield(self, spatial_prec=15, direct_prec=100, maxref=20)` jako argumenty podajemy:

- `spatial_prec` – jest przestrzenną rozdzielczością do wygenerowania siatki punktów wewnątrz brzegu kryształu. Wartość ta jest później przekazywana do funkcji `meshVolume(model,N)` jako `N`. Liczba wygenerowanych punktów będzie proporcjonalna do jej trzeciej potęgi oraz do stosunku objętości kryształu i jego najmniejszego prostopadłościanu.
- `direct_prec` – w przybliżeniu liczba wygenerowanych kierunków początkowych dla przeprowadzenia śledzenia promieni.
- `maxref` – maksymalna liczba odbić do jakiej przeprowadzamy śledzenie promieni.

Metoda ta zwraca wartość obserwowanej wydajności scyntylacji dla kryształu danego zadaną parametryzacją. Zakładamy, ze wydajność absolutna materiału kryształu jest równa jedności. W ciele metody, zanim wygenerujemy siatkę, sprawdzamy, czy czasem wcześniej siatka przestrzenna (na potrzeby całkowania) nie została już wygenerowana i zarejestrowana w tym obiekcie. Jako, że jej generowanie jest dość kosztowne obliczeniowo, a dla danego kryształu może zostać wykonane tylko raz, zabieg ten zwiększa szybkość obliczeń.

W linii `035` zapisujemy w zmiennej całkowitą liczbę kombinacji punktów i kierunków. Następnie inicjalizujemy pewne zmienne. W pętli iterujemy po punktach z siatki przestrzennej, generujemy losowe kierunki i zaczynamy po nich iterować. Dla każdego punktu i kierunku śledzimy wiązkę wewnątrz kryształu. Jeżeli liczba odbić jest dość mała, by była istotna, zapisujemy długość śledzonej wiązki i jej liczbę odbić. Po rozważeniu każdej możliwej kombinacji punktów siatki przestrzennej i kierunków wychodzimy z pętli, obliczamy parametr osłabienia dla każdej z tych kombinacji i liczymy średnią – czyli całkujemy. Parametr osłabienia jest zwracany.

Metoda:

```
screenMap(self, spatial_prec=15, direct_prec=100, maxref=20, pixels = 20, pointSource = Non
e, dirSeed = True, name = 'Screen intensity map')
```

jest istotna z punktu widzenia samego tytułu pracy. Służy ona wizualizacji procesów które zachodzą wewnątrz kryształu. Z jej pomocą możemy wygenerować obraz przedstawiający mapę intensywności w przekroju okienka fotopowielacza. Przyjmuje argumenty takie same jak metoda opisywana przed chwilą, a oprócz tego:

- `pixels` – wynikowa rozdzielczość renederowanego obrazu
- `pointSource` – tablica współrzędnych punktów w których umieszczamy oświetlenie punktowe (mogą być rozpatrywane jako punkty pojedynczych zdarzeń luminescencyjnych w objętości kryształu). Jeżeli nie podany, to jako oświetlenia punktowe rozważana jest siatka przestrzenna używana do całkowania.
- `dirSeed` – metoda jest polimorficzna ze względu na ten argument. Jeżeli przekazana zostanie prawda logiczna `True`, to liczba śledzony promieni będzie równa wcześniej podanej na użytek całkowania, natomiast jeśli będzie to liczba całkowita, to użyte zostanie właśnie tyle (w przybliżeniu) promieni
- `name` – nazwa okna w którym zostanie wyświetlony render

Metoda ta jest kwintesencją procesu *ray tracingu*, generuje ona realny render wnętrza kryształu. Gdyby oszlifowany kryształ przyłożyć do swojej gałki ocznej tak, jak mocowany on jest na okienku fotopowielacza, i w jakiś sposób spowodować wewnątrz zdarzenie luminescencyjne, to obraz jaki byśmy wtedy ujrzeli jest wynikiem działania tej metody.

W jej ciele najpierw testujemy obecność siatki, tak jak poprzednio. Inicjalizujemy zmienne `xy_hist` oraz `weights`. Informacje o obrazie będziemy przechowywać w tej pierwszej. Druga będzie przechowywać wartości osłabionych wiązek po dotarciu do okienka fotopowielacza. Wyliczamy wielokąt będący przecięciem płaszczyzny okienka z nietrywialnym brzegiem i znajdujemy wartości maksymalne i minimalne dla każdej z osi. Będą one wyznaczały granice renderowanego obrazu. W pętlach dokonujemy śledzenia promieni dla każdego promienia i dla każdej pozycji oświetlenia punktowego. Posiadając zbiór współrzędnych, na których promienie przecięły się z płaszczyzną okienka, generujemy histogram dwuwymiarowy, czyli macierz obrazu. Kolejne instrukcje obsługują pakiet *matplotlib* i wyświetlają na ekranie wynikowy obraz. Metoda zwraca uchwyty do okna i obrazu.

3.3.9. Funkcja `plotSurf(model, axes = False, z0 = True, color = 'g', alpha = 1)`

```
01  def plotSurf(model, axes = False, z0 = True, color = 'g', alpha = 1):
02
03      fig = plt.figure()
04      fig.canvas.set_window_title('Surface mesh model')
05      ax = fig.add_subplot(111, projection = '3d')
06
07      for tri in model.mesh:
08          lut = tri[:,2] >= 0
09          ax.plot(tri[lut,0], tri[lut,1], tri[lut,2], color = color, alpha = alpha)
010         if lut[0] and lut[-1]:
```

```
011             ax.plot(tri[[0,-1],0], tri[[0,-1],1], tri[[0,-
    1],2], color = color, alpha = alpha)
012     axisEqual3D(ax)
013     zeroBound = zeroPoly(model)
014     if z0:
015         for line in zeroBound:
016             ax.plot(line[:,0], line[:,1], line[:,2], color = 'm')
017 
018     x_ranges = np.array([np.max(zeroBound[:,:,0])*1.2, np.min(zeroBound[:,:,0])*1.2]
    )
019     y_ranges = np.array([np.max(zeroBound[:,:,1])*1.2, np.min(zeroBound[:,:,1])*1.2]
    )
020     X,Y, = np.meshgrid(x_ranges, y_ranges)
021     ax.plot_surface(X,Y,0, color = 'c', alpha = 0.2)
022 
023     if not axes:
024         ax.set_axis_off()
025 
026     plt.show()
027     return fig, ax
```

Funkcja ta służy wyświetleniu trójwymiarowego wykresu powierzchni kryształu. Jako argumenty przyjmuje:

- `model` – obiekt klasy `class SurfaceModel(object)`
- `axes` – wartość logiczna. Jeżeli `True` to na wykresie zostaną narysowane osie.
- `z0` – jeżeli `True` to narysowany zostanie także wielokąt który jest przecięciem nietrywialnego brzegu z płaszczyzną *xy*.
- `color` – kolor wykresu, kodowany według konwencji *matplotlib*
- `alpha` – przezroczystość, jeżeli 0 to całkiem przezroczysty wykres, jeżeli 1 to całkowicie nieprzezroczysty.

Na początku tworzone i konfigurowane są okna. Następnie w pętli iterujemy po trójkącie z siatki. Sprawdzamy, czy punkt będący wierzchołkiem trójkąta jest powyżej płaszczyzny *xy*, i jeżeli tak, to jest rysowana linia między nim a jego sąsiadem. Wykorzystujemy funkcję `axisEqual3D(ax)` aby wykres trójwymiarowy zachowywał proporcje osi. Jeżeli spełniony jest warunek, to rysowana jest linia przecięcia się kryształu z płaszczyzną *z=0*. Na koniec, w razie potrzeby wyłączamy widok osi. Funkcja zwraca uchwyt do okna i wykresu.

3.3.10. Funkcja

`sampleTrace(model, pos, dirs = None, ax = None, fig = None, maxRecursion = 15)`

Funkcja ta służy wyrysowaniu trójwymiarowego wykresu trajektorii dla przykładowego promienia.

```
01 def sampleTrace(model, pos, dirs = None, ax = None, fig = None, maxRecursion = 15):
02 
03     if (ax is None) or (fig is None):
04         figp, axp = plotSurf(model, axes = False, alpha = 0.3)
05     else:
06         figp = fig
07         axp = ax
08     axisEqual3D(axp)
09     figp.canvas.set_window_title('Trajectories')
010 
```

```
011     if dirs is None:
012         init_dirs = seedDirections(9)
013     elif type(dirs) is int:
014         init_dirs = seedDirections(dirs)
015     else:
016         init_dirs = dirs
017 
018     for direction in init_dirs:
019         beam = Ray(pos, direction)
020         beam.placeEnviorment(model)
021         beam.trace(maxRecursion)
022         axp.plot(beam.path[:beam.reflectCnt+2,0], beam.path[:beam.reflectCnt+2,1], b
    eam.path[:beam.reflectCnt+2,2], color = 'b', alpha = 0.6)
023         axp.scatter(beam.path[1:beam.reflectCnt+2,0], beam.path[1:beam.reflectCnt+2,
    1], beam.path[1:beam.reflectCnt+2,2], color = 'b', alpha = 0.6)
024 
025     plt.show()
026 
027     return figp, axp
```

Argumenty tej funkcji to:

- `model` – obiekt klasy `class SurfaceModel(object)`
- `pos` – tablica współrzędnych punktu z którego wyprowadzane będą śledzone promienie
- `dirs` – tablica kierunków, dla których ma zostać przeprowadzone śledzenie lub liczba kierunków, które mają zostać wylosowane
- `ax, fig` – uchwyt na wykres i okno, jeżeli obydwa podane, to na podanym wykresie zostanie dorysowana trajektoria, jeżeli nie, to zostanie ona narysowana na nowym wykresie i w nowym oknie
- `maxRecursion` - maksymalna liczba odbić przy śledzeniu promieni

Na początku sprawdzamy, czy podane zostało okno, w którym ma być narysowana trajektoria. Jeżeli nie, to jest ono tworzone i rysowana jest na nim powierzchnia. Następnie upewniamy się, że osie są poprawnie skalowane, wykonując na uchwycie do wykresu funkcję `axisEqual3D(axp)`. Następnie, jeżeli kierunki do śledzenia nie zostały podane, to są losowane w liczbie dziewięciu, jeżeli natomiast została podana liczba, to tyle właśnie jest losowane. Jeżeli została podana tablica kierunków, to właśnie te są używane. Dla każdego takiego kierunku przeprowadzane jest śledzenie, a następnie wynikowa trajektoria jest rysowana. Funkcja zwraca uchwyty do okna i wykresu.

4. Symulacje i obliczenia

W tym rozdziale wykorzystamy opisany powyżej kod, by wykonać obliczenia i symulacje dla kilku kształtów kryształów scyntylacyjnych. Rozważymy kryształy o kształtach półsfery i (bardziej egzotyczny) połówkę torusa. Dla każdego z nich wyrysujemy mapę intensywności, rozważymy zależność wydajności od ich parametrów (np. dla półsfery – jej promień) oraz zaprezentujemy przykładowe trajektorie. Aby tego dokonać napiszemy naprawdę krótkie skrypty w języku Python, korzystające z wyżej opisanych bibliotek.

4.1. Półsfera

Półsfera jest najprostszym kształtem różniczkowym jaki możemy rozważyć. Parametryzacja nietrywialnego brzegu jest dobrze znana, przedstawia się następująco:

$$\vec{S}(u,v) \coloneqq R[\sin(u)\cos(v), \sin(u)\sin(v), \cos(u)]$$

I aby była to półsfera, należy dobrać zakres zmienności parametrów:

$$u \in [0, \pi/2]$$
$$v \in [0, 2\pi]$$

Parametr R jest oczywiście jej promieniem.

4.1.1. Przebieg trajektorii

Wykonajmy najpierw wizualizacje przykładowych trajektorii. W tym celu tworzymy krótki skrypt:

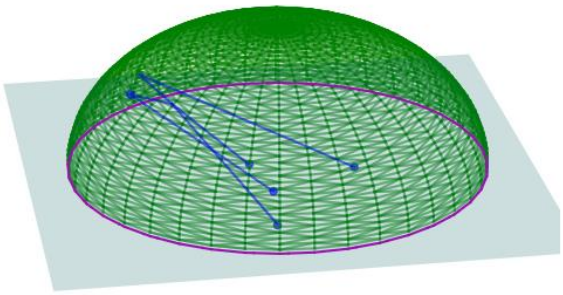
```
1.  import numpy as np
2.  from fwdraytracing.geo import *
3.  from fwdraytracing.utils import *
4.
5.
6.  crystal = Scintillator('u v',
7.                         '[sin(u)*cos(v),sin(u)*sin(v),cos(u)]',
8.                          ((0,np.pi/2 +0.01),(0,2*np.pi)),
9.                          precision_cnt=41,
10.                         alpha = 1, R=1)
11.
12. dirs = np.array([[0,0,1]])
13. dirshor = np.array([np.cos(np.linspace(0, 2*np.pi,9, endpoint=False)), np.sin(np.lin
    space(0, 2*np.pi,9,endpoint=False)), np.zeros(9)]).T
14.
15. fig, ax = sampleTrace(crystal.surface, pos = np.array([0,0.5,0.5]), dirs = 4, maxRec
    ursion = 99)
16.
17. fig2, ax2 = sampleTrace(crystal.surface, pos = np.array([0.6,0.6,0.01]), dirs = dirs
    , maxRecursion = 99)
18. fig3, ax3 = sampleTrace(crystal.surface, pos = np.array([0.7,0.7,0.01]), dirs = dirs
    , maxRecursion = 99)
19.
20. fig4, ax4 = sampleTrace(crystal.surface, pos = np.array([0,0,0.5]), dirs = dirshor,
    maxRecursion = 99)
```

Na początku tworzony jest obiekt `crystal`, przekazując do konstruktora odpowiednie argumenty, w tym parametryzację. Komentarza wymaga przekazanie zakresów zmienności argumentów, gdyż nie są one zgodne z tymi przedstawionymi wyżej. W rozdziale 2. zaznaczyliśmy, że to po stronie użytkownika leży obowiązek zapewnienia, by nietrywialny brzeg przecinał się z płaszczyzną *z=0*. Ze względu na numeryczne nieścisłości, najbezpieczniej jest przekazać wartość nieco większą niż ta, która jedynie dotyka żądanej. Z tego też powodu, do końca zakresu odpowiadającego za wertykalne rozpinanie sfery dodaliśmy względnie małą liczbę.

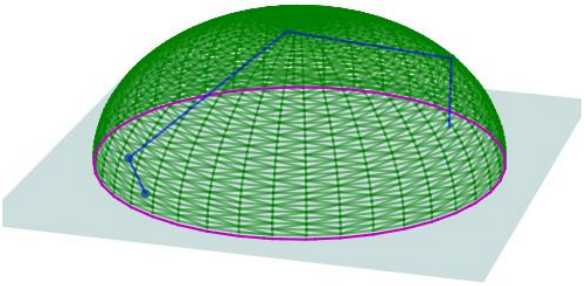
Skrypt generuje cztery wykresy, kolejno:

- przebieg czterech trajektorii o początku w punkcie `[0,0.5,0.5]` i losowych kierunkach
- przebieg trajektorii obrazującej efekt lekkiego ślizgania się po powierzchni
- przebieg trajektorii reprezentującej efekt silnego ślizgania się po powierzchni
- przebieg trajektorii które początkowo biegły całkowicie poziomo

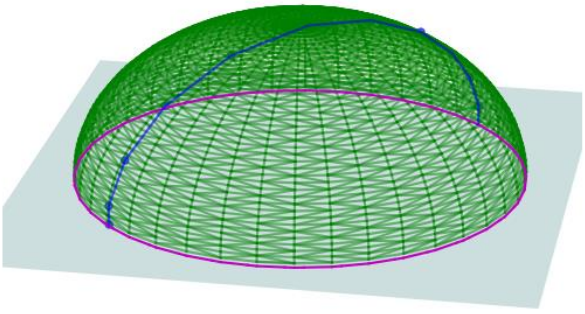
Rzuty z tych wykresów zostały przedstawione kolejno na rysunkach: 4.1, 4.2, 4.3, 4.4. Na rysunku 4.1. widzimy, że szarym kolorem zaznaczona jest płaszczyzna okienka fotopowielacza, kolorem różowym –



Rys. 4.1. Obraz przebiegu trajektorii z losowo wygenerowanych kierunków początkowych - dla półsfery o promieniu 1.

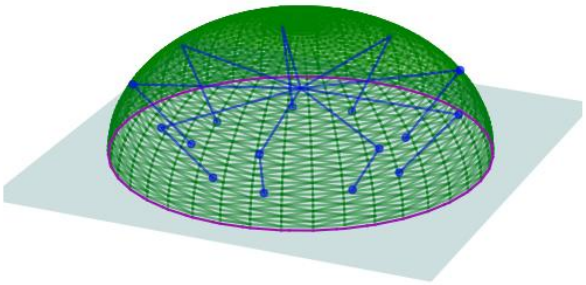


Rys. 4.2. Obraz przebiegu trajektorii z punktu *[.6, .6, .01]* i kierunku *[0,0,1]* - dla półsfery o promieniu 1.



Rys. 4.3. Obraz przebiegu trajektorii z punktu *[.7, .7, .01]* i kierunku *[0,0,1]* - dla półsfery o promieniu 1.

linia przecięcia brzegu z płaszczyzną okienka, kolorem zielonym – numeryczny model nietrywialnego brzegu, zaś kolorem niebieskim – trajektorie. Punkty przecięcia się trajektorii z całym brzegiem dodatkowo zaznaczane są niebieskimi kropkami.



Rys. 4.4. Obraz przebiegu trajektorii z kierunkami początkowymi leżącymi w płaszczyźnie poziomej - dla półsfery o promieniu 1.

Rys. 4.2. przedstawia efekt lekkiego ślizgania się po brzegu kryształu. Może on być rozważany jako efekt zarówno korzystny, jak i negatywny. W dużej mierze jednak przeważają jego zalety. Dokładniejszy obraz sytuacji przedstawiony jest na kolejnym rysunku. Widzimy, że promień uciekający od okienka jest z każdym odbiciem naprowadzany w kierunku okienka fotopowielacza. Właściwość ta powoduje, że każdy promień na pewno dotrze do detektora i zostanie zarejestrowany. W skrajnych przypadkach sytuacja taka wymaga jednak dość dużej liczby odbić. Przypominamy, że obserwowana wydajność scyntylacji jest tym większa im krótsza jest średnia droga promienia do okienka i im mniejsza jest średnia liczba odbić.

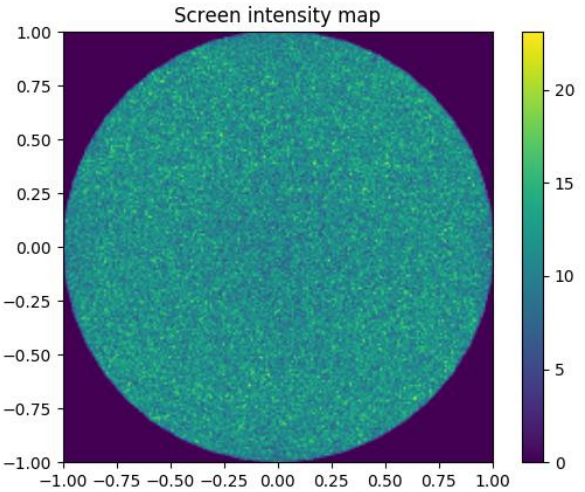
Rysunek 4.4. wizualizuje największe korzyści płynące z wyboru sfery jako kształtu dla kryształu scyntylacyjnego. Przedstawia on wynik śledzenia promieni, które rozpoczynają swój bieg posiadając zerową składową wertykalną kierunku. W przypadku kryształu prostopadłościennego, wiązki takie nigdy nie dotarły by do okienka fotopowielacza. Dla kształtów o odpowiednim znaku krzywizny, takim jak półsfera, promienie takie są natychmiast kierowane w kierunku płaszczyzny $z=0$, co zdecydowanie zwiększa obserwowaną wydajność scyntylacji. Kształty takie mają tę własność, że z każdym odbiciem zawsze *zmniejszają* składową wertykalną kierunku propagacji, wobec czego kierują wiązkę na okienko.

4.1.2. Mapa intensywności

Aby utworzyć mapę intensywności, wykorzystamy następujący skrypt:

```
01 import numpy as np
02 from fwdraytracing.geo import *
03 from fwdraytracing.utils import *
04
05
06 crystal = Scintillator('u v',
07                         '[sin(u)*cos(v),sin(u)*sin(v),cos(u)]',
08                          ((0,np.pi/2 +0.01),(0,2*np.pi)),
09                          precision_cnt=41,
010                         alpha = 1, R=0.95)
011
012 map1, ax1 = crystal.screenMap(spatial_prec=20, direct_prec=300, maxref=50, pixels=20
    0)
```

W skrypcie tym wykorzystujemy tylko dwie instrukcje, pomijając oczywiście nagłówek importujący potrzebne moduły. W linii `06` tworzymy obiekt klasy `Scintillator`, przekazując do konstruktora niezbędne informację: parametryzację kształtu, zakres zmienności parametrów, liczbę podziałów siatki parametrów, wartość współczynnika pochłaniania i wartość współczynnika odbicia. W ostatniej linii wywołujemy metodę odpowiednią do wygenerowania obrazu. Przekazujemy do niej: rozdzielczość przestrzennego całkowania (~20^3 punktów całkowania), rozdzielczość całkowania po kierunkach (~300 wygenerowanych kierunków), maksymalna rozważana liczba odbić, oraz rozdzielczość wynikowego obrazu (200x200 px). Przeprowadzone wcześniej testy pokazywały, że taki dobór parametrów całkowania skutkuje w precyzji obliczeń wydajności scyntylacji na poziomie trzech cyfr znaczących, więc niepewność zawiera się w granicach kilku procent. Efektem działania skryptu jest rzeczywisty render obrazu, jaki pojawiłby się na płaszczyźnie okienka fotopowielacza, gdyby móc obserwować pomiar w ten sposób. Obraz ten przedstawiony został na rysunku 4.5. Możemy na nim zaobserwować, że w krysztale który błyszczy w wielu miejscach jednocześnie, rozkład intensywności na okienku fotopowielacza jest raczej równomierny. Przede wszystkim, nie widzimy aby jakikolwiek kierunek był wyróżniony pod względem intensywności, co zgadza się z intuicją dotyczącą sfery – która jest kształtem osiowo-symetrycznym.

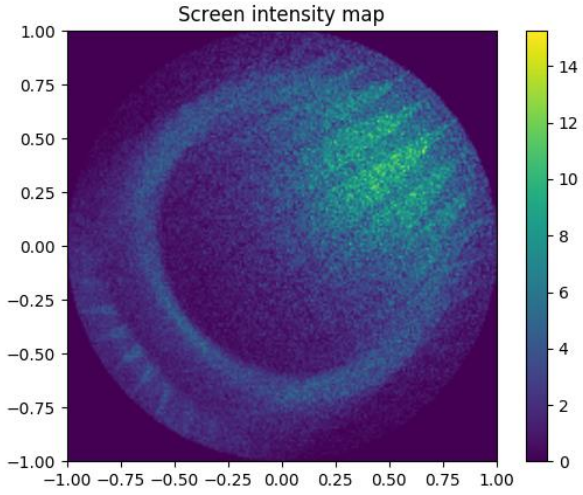


Rys. 4.5. Render obrazu jaki powstałby na płaszczyźnie okienka fotopowielacza, dla kryształu o kształcie półsfery o promieniu 1. Na osiach x,y przedstawiono wymiary liniowe, kolorem zaznaczono intensywność danego pixela (w jednostkach arbitralnych).

Sytuacja ma się zupełnie inaczej, gdy zaznaczyć, by kryształ był oświetlany błyskiem powstającym w określonym punkcie. Wykorzystując wcześniejszy skrypt i nieco go modyfikując, możemy wyrenderować obraz powstały w wyniku oświetlenia punktowego w krysztale.

```
01 import numpy as np
02 from fwdraytracing.geo import *
03 from fwdraytracing.utils import *
04
05
06 crystal = Scintillator('u v',
07                         '[sin(u)*cos(v),sin(u)*sin(v),cos(u)]',
08                          ((0,np.pi/2 +0.01),(0,2*np.pi)),
09                          precision_cnt=41,
010                         alpha = 1, R=0.95)
011
012 map1, ax1 = crystal.screenMap(direct_prec=300000, maxref=50, pixels=200, pointSource
    =np.array([[.3,.4,.5]]))
```

Jak widać, do poprzedniego wywołania metody rysującej render dodaliśmy opcjonalny argument mówiący o położeniu oświetlenia (wzbudzenia) punktowego, jeżeli taki typ oświetlenia chcemy wykorzystać. Argument ten, `pointSource=np.array([[.3,.4,.5]])` przekazuje do metody informację, że oświetlenie punktowe ma leżeć w położeniu `[.3,.4,.5]`. Jeżeli chcielibyśmy dodać kolejne oświetlenie punktowe, jego pozycję należałoby wpisać po przecinku (np. `pointSource=np.array([[.3,.4,.5], [.4,.5,.6]])`). Można też zauważyć, że zwiększyliśmy liczbę kierunków. W ten sposób lepiej odwzorowujemy źródło punktowe i zapewniamy odpowiednią jasność pixeli. Gdyby pozostawić ten parametr niezmieniony, to w poprzednim przypadku na jeden piksel przypadało około ~20^3 razy więcej promieni, gdyż były one wyprowadzane z każdego z tych punktów.



Rys. 4.6. Render obrazu jaki powstałby na płaszczyźnie okienka fotopowielacza, dla kryształu o kształcie półsfery o promieniu 1, oświetlonego z punktu o współrzędnych `[.3,.4,.5]`. Na osiach x,y przedstawiono wymiary liniowe, kolorem zaznaczono intensywność danego pixela (w jednostkach arbitralnych).

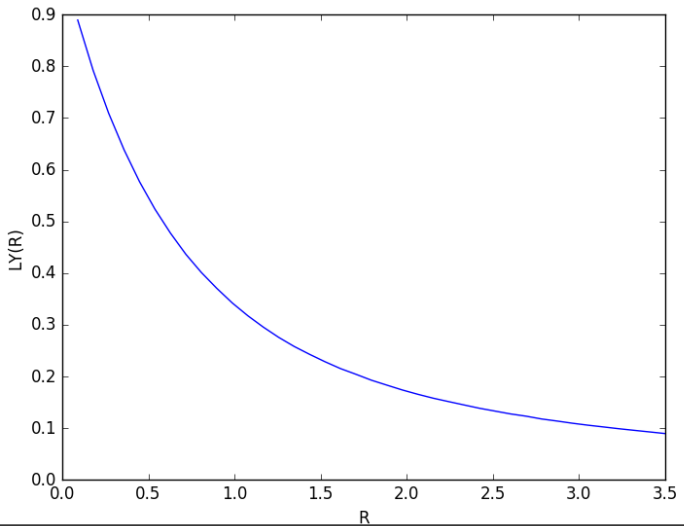
Efekt działania tego skryptu obserwujemy na rysunku 4.6. Widzimy, że tym razem rozkład natężenia z pewnością nie jest jednorodny. Wyraźnie wyróżniony jest punkt z którego oświetlaliśmy kryształ, aczkolwiek wyróżnienie to także nie jest trywialne. Widoczne miraże są efektem tego, w jak regularny sposób promienie odbijają się od wnętrza kształtu jakim jest sfera. O ile do obliczeń wydajności scyntylacji takie rozważania wnoszą niewiele, to rozkład oświetlenia punktowego jest niezwykle kluczowy w zagadnieniach dotyczących urządzeń obrazowania medycznego, zwanych gamma kamerami [13]. W gamma kamerach obraz powstaje w wyniku badania rozkładu intensywności światła na jednej z płaszczyzn końcowych dużego kryształu. Widzimy, że w przypadku kształtu, jakim jest półsfera, rozstrzygnięcie na podstawie rozkładu intensywności, gdzie krysztale pojawił się błysk, nie jest wcale trywialnym zadaniem.

4.1.3. Obserwowana wydajność scyntylacji

Zwieńczeniem procesu symulacji, w naszym przypadku, będzie wyliczenie wielkości łatwo mierzalnej, czyli obserwowanej wydajności scyntylacji. Oczywiście nie podamy tej wielkości w postaci dokładnej, gdyż nie znamy wydajności absolutnych oraz parametrów fotopowielacza, natomiast możemy z całą pewności wyznaczyć *zależność* wydajności od rozmiarów kryształu, i zależność ta będzie proporcjonalna do mierzalnej. Będziemy więc rozważać półsferę o różnych promieniach i sprawdzimy, jaką wydajność dla każdej obserwujemy. W tym celu musimy stworzyć skrypt, który wygeneruje parametryzację każdego rozważanego kształtu i wyliczy wydajność scyntylacji:

```
01 import numpy as np
02 import matplotlib.pyplot as plt
03 from fwdraytracing.geo import *
04 from fwdraytracing.utils import *
05 from time import clock
06
07
08 rads = np.linspace(0,3.5, 40)[1:]
09 base = '[sin(u)*cos(v),sin(u)*sin(v),cos(u)]'
010 argstr = []
011 for radii in rads:
012     splited = base[1:-1].split(',')
013     argstr.append( '[' + ','.join([str(radii) + '*' + s for s in splited]) + ']' )
014
015 ress = np.zeros(len(rads))
016 times = np.zeros(len(rads))
017
018 for i in xrange(len(ress)):
019     arg = argstr[i]
020     t0 = clock()
021     crystal = Scintillator('u v', arg,((0,(np.pi+0.2)/2 ),(0,2*np.pi)))
022     LY = crystal.lightYield(spatial_prec = 15, direct_prec = 500)
023     t1 = clock()
024     ress[i] = LY
025     times[i] = t1-t0
026
027
028
029
030 with open('results_sphere.data','w') as f:
031     for i in xrange(len(ress)):
032         f.write(str(rads[i]) + '\t')
033         f.write(str(ress[i]) + '\t')
034         f.write(str(times[i]) + '\n')
035
036 data = np.loadtxt('results.data')
037
038 fig = plt.figure()
039 ax = fig.add_subplot(111)
040 ax.plot(data[:,0], data[:,1])
041 ax.set_xlabel("R")
042 ax.set_ylabel("LY(R)")
```

Skrypt ten po pierwsze generuje listę parametryzacji, każda dla różnych wartości promienia. Promień przebiega wartości od 0 do 3.5. Następnie, dla każdej parametryzacji, tworzony jest obiekt scyntylatora i obliczana jest obserwowana wydajność scyntylacji (z dokładnością do wydajności absolutnej kryształu). Oprócz tego, obliczany jest czas poświęcony na obliczenia dla każdej parametryzacji. Jako, że czas poświęcony na obliczenia jest dość duży, prezentujemy także, w jaki sposób wygenerowane dane zapisać do pliku. Następnie dane odczytywane są z



Rys. 4.7. Wykres obserwowanej wydajności scyntylacji dla kryształu o kształcie półsfery. Na osi pionowej odłożono obserwowaną wydajność scyntylacji (z dokładnością do wydajności absolutnej), a na osi poziomej – promień.

pliku, a potem rysowane na wykresie. Wykres ten prezentujemy na Rys. 4.7. Widzimy na nim, że charakter przebiegu jest wciąż bliski temu prezentowanemu przez Wojtowicza i współpracowników w modelu 2R [3].

4.2. Półtorus

Połtorus jest dość egzotycznym kształtem, jaki możemy sobie wyobrazić jako kryształ scyntylacyjny. Mimo to chcemy zaprezentować w ten sposób możliwości stworzonego oprogramowania. Parametryzacja rozważanego przez nas kształtu wygląda następująco:

$$\vec{S}(u,v) = [(R + r\cos(u))\cos(v), (R + r\cos(u))\sin(v), r\sin(u)]$$

$$u \in (0,\pi)$$
$$v \in (0,2\pi)$$

W ten sposób, parametr *u* tworzy koło małe torusa, a parametr *v* obraca koło małe wokół osi *z*. Promień wielki *R* pozostanie w naszym przypadku stały i przyjmie wartość *R=2.5*. Do kreślenia wykresu wydajności scyntylacji przyjmiemy, że zmieniać będzie się promień mały *r* i to w dodatku według następującej zależności:

$$r = \sqrt{\frac{2r_s^3}{3\pi R}}$$

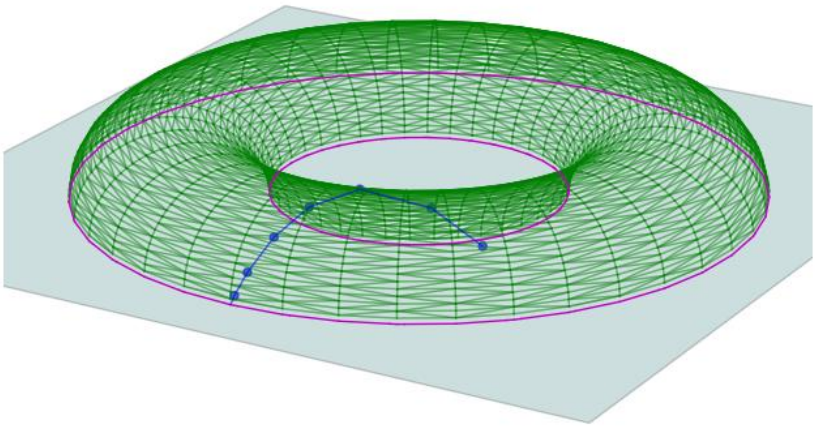
gdzie przez r_s oznaczyliśmy promień poprzednio rozważanej przez nas sfery. Przez taki zabieg zapewnimy, że oba rozważane przez nas kształty będą miały tę samą objętość. Warunek ten można uznać jako warunek normalizacyjny na cele porównywania wydajności kryształów.

4.2.1. Przebieg trajektorii

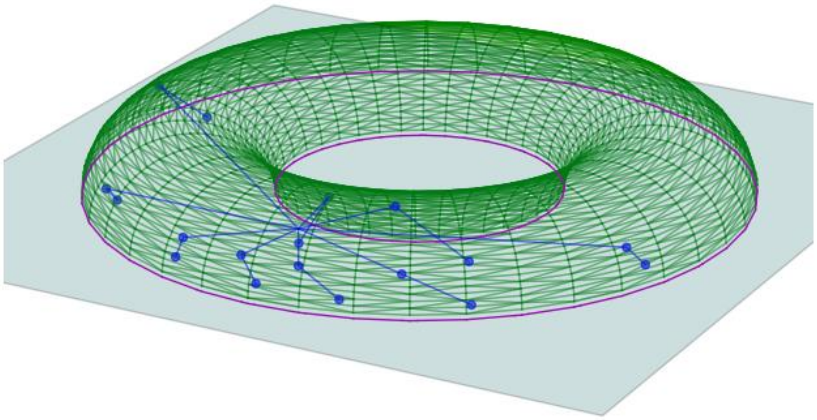
Przeanalizujmy zachowanie się przykładowych trajektorii wewnątrz rozważanego kształtu kryształu. Zmodyfikujemy nieco skrypt który generował trajektorie dla sfery:

```
1.  import numpy as np
2.  from fwdraytracing.geo import *
3.  from fwdraytracing.utils import *
4.
5.
6.  crystal = Scintillator('u v',
7.                         '[(2.5 + cos(u))*cos(v),(2.5 + cos(u))*sin(v),sin(u)]',
8.                          ((-0.01,np.pi+0.01),(0,2*np.pi)),
9.                          precision_cnt=41,
10.                         alpha = 1, R=1)
11.
12. dirs = np.array([[-0.35,0,0.9]])
13. dirshor = np.array([np.cos(np.linspace(0, 2*np.pi,9, endpoint=False)), np.sin(np.lin
    space(0, 2*np.pi,9,endpoint=False)), np.zeros(9)]).T
14.
15. fig1, ax1 = sampleTrace(crystal.surface, pos = np.array([0,3.49,0.01]), dirs = dirs,
     maxRecursion = 99)
16.
17. fig2, ax2 = sampleTrace(crystal.surface, pos = np.array([0,2.5,0.5]), dirs = dirshor
    , maxRecursion = 99)
```

Skrypt ten generuje dwa interaktywne wykresy trójwymiarowe. Przedyskutujemy na nich te same sytuacje które dyskutowaliśmy dla sfery. Pierwszy wykres prezentuje sytuację ślizgania się promienia świetlnego, zamieszczamy go na rysunku 4.8. Widzimy na nim, że sytuacja ślizgania się promieni po powierzchni kryształu wciąż jest obecna, choć ma nieco inny charakter niż na sferze. Wciąż jednak promienie skierowane początkowo ku górze, są ostatecznie prowadzone w stronę okienka fotopowielacza. Ze względu na dość jednorodną krzywiznę torusa, nie będą istnieć takie promienie, które odbijać się będą nieskończenie wiele razy, zanim dotrą do okienka fotopowielacza. Z kolei na rys. 4.9. prezentujemy sytuację w której promienie nie mają składowej pionowej początkowego kierunku - biegną równolegle do okienka fotopowielacza. W tym momencie krzywizna torusa także spełnia pozytywną rolę – każdy wyprowadzony promień już po pierwszym odbiciu trafia w okienko fotopowielacza.



Rys. 4.8. Obraz przebiegu trajektorii która *ślizga* się po powierzchni torusa.



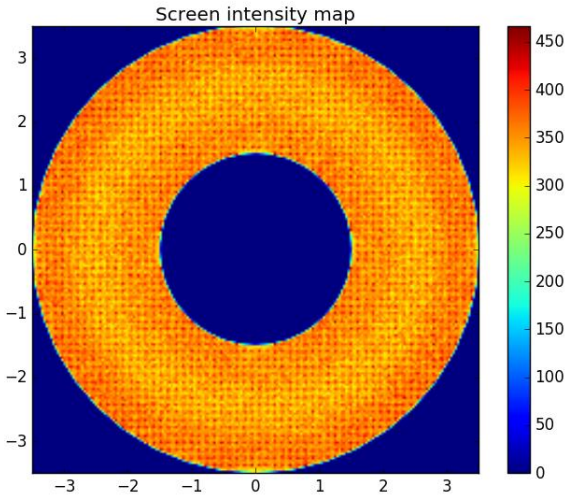
Rys. 4.9. Obraz przebiegu trajektorii o początkowych kierunkach skierowanych równolegle do płaszczyzny okienka fotopowielacza.

4.2.2. Mapa intensywności

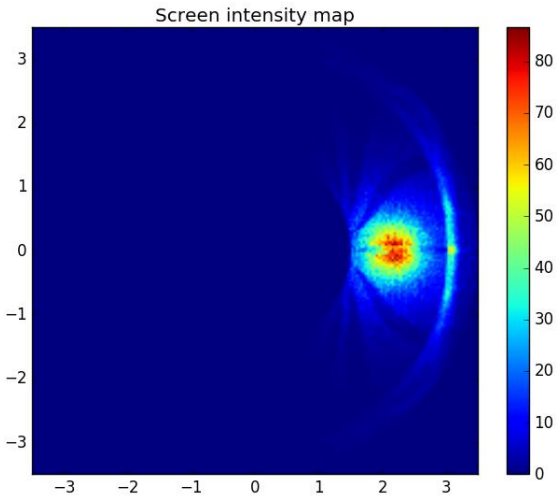
Aby utworzyć mapę intensywności, wykorzystujemy skrypt strukturalnie taki sam, jak analogiczny przedstawiony dla sfery. Zmieniamy jedynie parametryzację brzegu kryształu:

```
01 import numpy as np
02 from fwdraytracing.geo import *
03 from fwdraytracing.utils import *
04
05
06 crystal = Scintillator('u v',
07                         '[(2.5 + cos(u))*cos(v),(2.5 + cos(u))*sin(v),sin(u)]',
08                          ((-0.01,np.pi+0.01),(0,2*np.pi)),
09                          precision_cnt=41,
010                         alpha = 1, R=1)
011
012 map1, ax1 = crystal.screenMap(spatial_prec=60, direct_prec=300, maxref=200)
```

Efektem jego działania jest mapa rozkładu intensywności promieniowania docierającego do okienka fotopowielacza. Prezentujemy ją na rys. 4.10. Widzimy, że i tym razem promieniowanie rozkłada się na okienko w dość równomierny sposób. Można zauważyć, że obraz natężenia promieniowania jest dość *ziarnisty*, mimo, że tym razem użyliśmy 60-ciu podziałów odcinka do zasiania punktów całkowania. Może to świadczyć o tym, że kształt torusa mniej sprzyja



Rys. 4.10. Render obrazu jaki powstałby na płaszczyźnie okienka fotopowielacza, dla kryształu o kształcie półtorusa, rozświetlonego błyskami równomiernie rozłożonymi w objętości kryształu.



Rys. 4.11. Render obrazu jaki powstałby na płaszczyźnie okienka fotopowielacza, dla kryształu o kształcie półtorusa, rozświetlonego błyskiem z pozycji *[0, 2.2, 0.5]*.

rozpraszaniu się światła. Oprócz tego można zauważyć, że intensywność światła przy krańcach obszaru styku torusa z okienkiem jest nieco większa. Różnica ta jest niewielka, aczkolwiek obecna. Ciekawym okazuje się być render wnętrza torusa oświetlonego oświetleniem punktowym. Oświetlając kryształ z punktu o współrzędnych *[0,2.2,0.5]* otrzymujemy rozkład intensywności przedstawiony na rysunku 4.11. Widzimy na nim, że w porównaniu do sfery, wynikowy rozkład

promieniowania jest o wiele bardziej punktowy. Ciekawym zjawiskiem mogą też być widoczne, pojawiające się na przemian - ciemne i jasne prążki. Okazuje się więc, że dla rozważanego kształtu, istnieją obszary, w pewnym sensie zabronione, lub trudno dostępne dla promieniowania wyprowadzonego z oświetlenia punktowego. Widoczna jest także łuna przy zewnętrznej krawędzi obszaru styku torusa z okienkiem. Łuna ta jest zbliżona jasnością do samego punktu. Obserwacja ta tylko potwierdza poprzednią – torus faworyzuje obszary na swoich krawędziach. Jakkolwiek dziwnym byłby wybór torusa jako kryształu bazowego do gamma kamery, musimy przyznać, że jest on wyborem o wiele lepszym niż sfera, która rozprasza nawet promieniowanie punktowe.

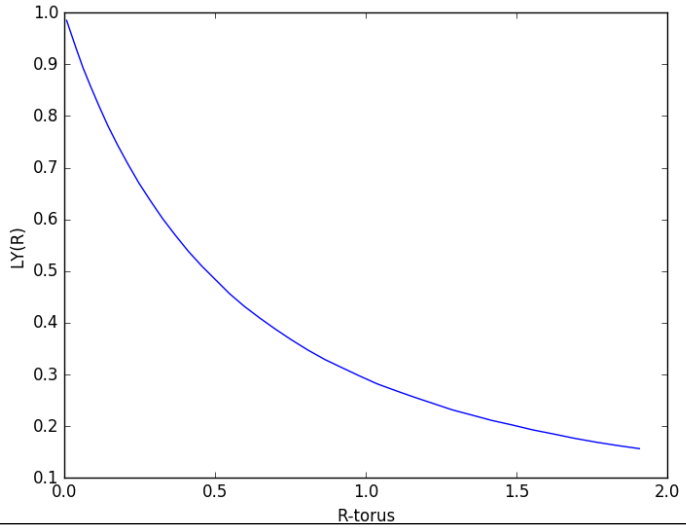
4.2.3. Obserwowana wydajność scyntylacji.

Do wykreślenie obserwowanej wydajności scyntylacji wykorzystamy następujący skrypt:

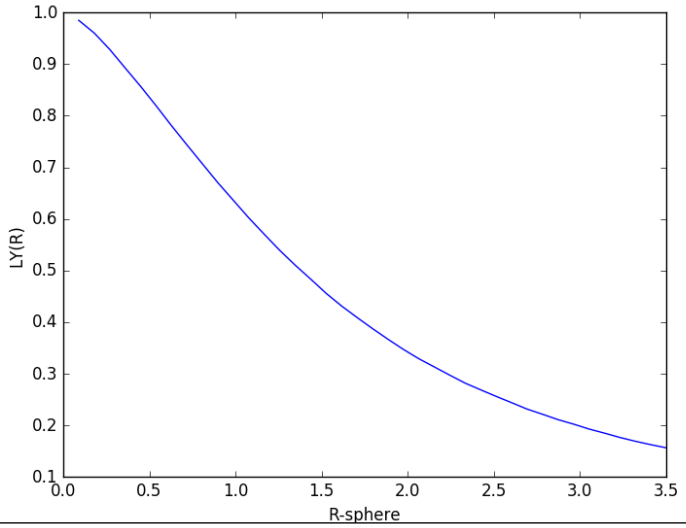
```
import numpy as np
import matplotlib.pyplot as plt
from fwdraytracing.geo import *
from fwdraytracing.utils import *
from time import clock

Rt = 2.5
rads = np.linspace(0,3.5, 40)[1:]
rads_tor = np.sqrt((rads**3)*2.0/(3*np.pi*Rt))
base = '[(2.5 + %f*cos(u))*cos(v),(2.5 +%f*cos(u))*sin(v),%f*sin(u)]'
argstr = []
for radii in rads_tor:
    argstr.append( base % (radii, radii, radii))

ress = np.zeros(len(rads_tor))
times = np.zeros(len(rads_tor))

for i in xrange(len(ress)):
    arg = argstr[i]
    t0 = clock()
    crystal = Scintillator('u v', arg,((-0.01,np.pi+0.01),(0,2*np.pi)))
    if rads_tor[i] < 0.30:
        LY = crystal.lightYield(spatial_prec = 40, direct_prec = 400)
    else:
        LY = crystal.lightYield(spatial_prec = 20, direct_prec = 400)
    t1 = clock()
    ress[i] = LY
    times[i] = t1-t0
    print (i, times[i])



with open('results_torus.data','w') as f:
    for i in xrange(len(ress)):
        f.write(str(rads[i]) + '\t')
        f.write(str(rads_tor[i]) + '\t')
        f.write(str(ress[i]) + '\t')
        f.write(str(times[i]) + '\n')

data = np.loadtxt('results_torus.data')

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(data[:,1], data[:,2])
ax.set_xlabel("R-torus")
```

```
47. ax.set_ylabel("LY(R)")
48. plt.show()
49.
50. fig2 = plt.figure()
51. ax2 = fig2.add_subplot(111)
52. ax2.plot(data[:,0], data[:,2])
53. ax2.set_xlabel("R-sphere")
54. ax2.set_ylabel("LY(R)")
55. plt.show()
56.
```

Skrypt ten jest podobny do analogicznego przedstawionego dla sfery, z wyjątkiem linii od 022 do 025. W związku z tym, że w pewnym momencie skalowania małego promienia torusa, punkty zasiane do całkowania mogą przestać dobrze wypełniać zajmowaną przez kryształ przestrzeń, dla mniejszych promieni generujemy więcej punktów całkowania. Oprócz tego, na samym początku skryptu dokonujemy skalowania zgodnie z tym przedstawionym na początku tego podrozdziału. Efektem działania skryptu są dwa wykresy prezentowane na rysunkach 4.12. i 4.13. Pierwszy z nich prezentuje obserwowaną wydajność scyntylacji w funkcji promienia koła małego torusa. Jest to oczywiste przedstawienie przebiegu zmienności tej wielkości i widzimy, że i tym razem kształt krzywej opisywanej przez Wojtowicza i współpracowników w modelu 2R [3] jest zachowany. Aby jednak porównać wyniki dla torusa i sfery, musimy ustalić jakiś wspólny mianownik dla tych dwóch kształtów. Oczywistym wyborem wydaje się być zażądanie, aby porównywane kształty miały tę sama objętość. Wybierając takie kryterium, można otrzymać zależność między promieniem sfery a promieniem małym torusa. W świetle tej zależności wyrysowany został



Rys. 4.12. Wykres obserwowanej wydajności scyntylacji dla kryształu o kształcie półtorusa. Na osi pionowej odłożono znormalizowaną obserwowaną wydajność scyntylacji, a na osi poziomej promień koła małego torusa.



Rys. 4.13. Wykres obserwowanej wydajności scyntylacji dla kryształu o kształcie półtorusa. Na osi pionowej odłożono znormalizowaną obserwowaną wydajność scyntylacji, a na osi poziomej promień sfery która miałaby taką samą objętość jak rozważany torus.

wykres przedstawiony na rys. 4.14. Na osi poziomej odłożony został promień takiej sfery, dla której rozważany torus ma taką samą objętość jak ona. Tak przedstawiona zależność nie pasuje już do modelu 2R [3].

4.3. Porównanie wydajności kryształów

Zwieńczeniem przedstawionej pracy będzie porównanie wydajności scyntylacji dla obydwu rozważanych kształtów oraz dla kształtu wzorcowego – używanego najczęściej w obrazowaniu medycznym, czyli sześcianu. Wydajność scyntylacji kryształu prostopadłościennego dokładnie opisuje model stworzony w pracy licencjackiej autora, częściowo zreferowany także w artykule [7]. Dla kryształu scyntylacyjnego o doskonale odbijających ścianach, obserwowana wydajność scyntylacji przedstawia się wzorem:

$$LY(H) = \frac{1}{4H}\int_0^H dz_0 \left(\int_0^{\pi} d\vartheta \left(e^{-\alpha \frac{2H - z_0}{|\sin(\vartheta)|}} \right) + \int_{\pi}^{2\pi} d\vartheta \left(e^{-\alpha \frac{z_0}{|\sin(\vartheta)|}} \right) \right)$$

Kryształ taki ma taką samą objętość jak półsfera, gdy tylko jego wymiar liniowy H związany jest z promieniem r_s półsfery następująca zależnością:

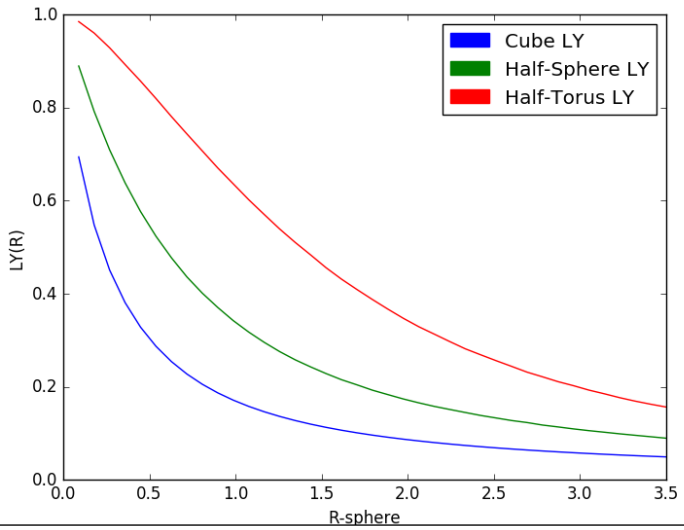
$$H = r_s \sqrt{\frac{2\pi}{3}}$$

Do porównania wydajności tych trzech kształtów wykorzystamy następujący skrypt:

```
01 import numpy as np
02 from fwdraytracing.geo import *
03 from fwdraytracing.utils import *
04 import matplotlib.pyplot as plt
05 import matplotlib.patches as mpatches
06 from scipy.integrate import dblquad
07
08 def podcalk(H,z0,theta):
09     if 0 <= theta < np.pi:
010         o = np.exp(-(2*H - z0)/np.abs(np.sin(theta)))*np.abs(np.cos(theta))
011     elif np.pi <= theta < 2*np.pi:
012         o = np.exp(-(z0)/np.abs(np.sin(theta)))*np.abs(np.cos(theta))
013     return o
014
015 def LY(H):
016     f = lambda z0,th: podcalk(H, z0,th)
017
018     return dblquad(f, a = 0, b = 2*np.pi, gfun = lambda th: 0, hfun = lambda th: H)[
    0]/(4*H)
019
020
021 data_sphere = np.loadtxt("results_sphere.data")
022 data_torus = np.loadtxt("results_torus.data")
023
024 rads = data_sphere[:,0]
025 rads_tor = data_torus[:,1]
026 heights = np.sqrt(2*np.pi/3)*rads
027
028 Ly3D = np.array([LY(H) for H in heights])
029 Lysphere = data_sphere[:,1]
```

```
030 Lytorus = data_torus[:,2]
031
032 fig = plt.figure()
033 ax = fig.add_subplot(111)
034 ax.plot(rads, Ly3D, color='b')
035 ax.plot(rads, Lysphere, color='g')
036 ax.plot(rads, Lytorus, color='r')
037
038 ax.set_xlabel("R-sphere")
039 ax.set_ylabel("LY(R)")
040
041 blue_patch = mpatches.Patch(color='blue', label='Cube LY')
042 green_patch = mpatches.Patch(color='green', label='Half-Sphere LY')
043 red_patch = mpatches.Patch(color='red', label='Half-Torus LY')
044 plt.legend(handles = [blue_patch, green_patch, red_patch])
045
046 plt.show()
```

Skrypt ten wprowadza podany wyżej wzór na wydajność scyntylacji kryształu w kształcie kostki, ładuje z pliku poprzednie obliczenia i rysuje dane na wykresie. Wykres ten prezentujemy na rys. 4.14. Zależności, gdy przedstawić je na wspólnym obrazie, prezentują się nad wyraz ciekawie.



Rys. 4.14. Wykres porównujący wydajności wcześniej dyskutowanych kształtów z wydajnością kryształu sześciennego. Na osi poziomej odłożono promień półsfery dla której reszta ma taką samą objętość jak wspomniana półsfera.

Na poziomej osi odłożono promień pół-sfery dla której pozostałe kształty mają tę samą objętość jak ona. Dla danego punktu na osi poziomej każdy z kształtów ma tę samą objętość.

To, co od razu rzuca się w oczy, to fakt, że najpowszechniej używany kształt kryształu – prostopadłościan, ma najmniejszą obserwowaną wydajność ze wszystkich rozważanych. Jest to *najważniejszy wniosek* płynący ze wszystkich rozważań prowadzonych w tej pracy. Następnym w kolejności kształtem jest półsfera. Kształt ten ma obserwowaną wydajność zdecydowanie większą niż prostopadłościan o tej samej objętości. Zaskoczeniem natomiast okazuje się być torus. To, że półsfera mogłaby mieć większą wydajność niż prostopadłościan mogło wydawać się oczywiste, natomiast to, że półtorus sprawdzi się lepiej nić pół-sfera, było nierozstrzygnięte. Wykres 4.14. rozstrzyga tę kwestię zdecydowanie na korzyść torusa. Dyskusyjne pozostaje jedynie to, czy szlifowanie kryształu w tak egzotyczny kształt jest opłacalne i wykonalne, jednak odpowiedź na to pytanie wykracza poza kompetencje autora.

5. Podsumowanie

Przedstawiona praca okazała się być bardzo mocno zorientowana na koncepcję i implementacje pomysłu użycia *ray-tracingu* do obliczeń obserwowanej wydajności scyntylacji kryształów scyntylacyjnych. Znaczną większość jej objętości zajmuje przedstawienie używanych algorytmów i opis ich implementacji w języku C oraz Python. Stawia ją to jako pracę o zorientowaniu inżynierskim, jednak bez wykorzystania zaprojektowanych i zaimplementowanych metod miałaby ona małą wartość. Pełnym sukcesem okazuje się być to, że implementacja algorytmów *działa* dość niezawodnie. Kwestią dalszych udoskonaleń pozostaje dopracowanie kodu pod względem szybkości działania. Istnieją bowiem takie fragmenty kodu, które już na pierwszy rzut oka powodują zmniejszenie wydajności działania programu. Celem pracy było zaprojektowanie, implementacja i *wykorzystanie* algorytmów do obliczeń związanych z wydajnością scyntylacji w kryształach. W rozdziale 4. Przedstawiliśmy wiele ciekawych faktów na temat tego, jak zachowuje się światło wewnątrz kryształów o różnych geometriach. Pewien niedosyt pozostawiać może rozważenie tylko dwóch kształtów kryształu scyntylacyjnego. Jednak zabieg ten jest umyślny i w oczach autora dzięki niemu będzie można rozstrzygnąć, czy stworzony kod okaże się użyteczny w przyszłości dla innych osób. Przedstawiono szablony, które w łatwy sposób pozwalają wykonanie dokładnie takich samych obliczeń dla prawie dowolnego kształtu różniczkowego. Autor planuje w niedalekiej przyszłości umieścić prezentowany kod w publicznym repozytorium w serwisie *git*, tak by był on dostępny dla każdego zainteresowanego tematem zależności obserwowanej wydajności scyntylacji od kształtu kryształu. Mimo, że autor nie wiąże swojej dalszej przyszłości z pracą naukową, projekt ten będzie otoczony jego ciągłą opieką i zdecydowanie chętnie udzieli odpowiedzi na wszelkie pytania jego dotyczące.

Ostatecznie cel pracy został osiągnięty. Zaprojektowano i zaimplementowano odpowiednie metody, oraz wykorzystano je do symulacji zachowania się światła w kryształach scyntylacyjnych. Udało się także poczynić jedną, niezmiernie istotną obserwację – najbardziej popularny kształt kryształu scyntylacyjnego – prostopadłościan - okazuje się być najmniej optymalny pod względem obserwowanej wydajności scyntylacji. Każdy zainteresowany tematem scyntylatorów powinien rozważyć zmianę podejścia do szlifowania ich kształtu. O ile kształt torusa jest bardziej ciekawostką, niż realną opcją, o tyle szlifowanie kryształu na kształt półsfery wydaje się być całkiem realnym pomysłem!

6. Biblografia

[1] A. Lempicki, A.J. Wojtowicz, E. Berman, "Fundamental Limits of Scintillator Performance", *Nuclear Instruments and Methods in Physics Research* **A333** (1993), 304-311

[2] M. Nikl, A. Yoshikawa, "Recent R&D Trends in Inorganic Single-Crystal Scintillator Materials for Radiation Detection", *Advanced Optical Materials* **3** (2015), 463-481

[3] A.J. Wojtowicz, W. Drozdowski, M. Ptaszyk, Z. Gałązka, J.L. Lefaucheur, "Scintillation Light Yield of Ce-Doped LuAP and LuYAP Pixel Crystals", in: "Proceedings of the 8th International Conference on Inorganic Scintillators and their Use in Scientific and Industrial Applications", eds. A. Gektin, B. Grinyov, National Academy of Sciences of Ukraine, Kharkov 2006, pp. 473-476

[4] C. Dujardin, C. Pedrini, W. Blanc, J.C. Gacon, J.C. van't Spijker, O.W.V. Frijns, C.W.E. van Eijk, P. Dorenbos, R. Chen, A. Fremout, F. Tallouf, S. Tavernier, P. Bruyndonckx, A.G. Petrosyan, "Optical and Scintillation Properties of Large $LuAlO_3$:Ce^{3+} Crystals", *Journal of Physics: Condensed Matter* **10** (1998), 3061-3073

[5] M. Balcerzyk, M. Moszyński, Z. Gałązka, M. Kapusta, A. Syntfeld, J.L. Lefaucheur, "Perspectives for High Resolution and High Light Output LuAP:Ce Crystals", *IEEE Transactions on Nuclear Science* **52** (2005), 1823-1829

[6] *Praca dyplomowa:* "Teoretyczny opis wydajności scyntylacji w zależności od rozmiarów kryształu w układzie detekcyjnym scyntylator + fotopowielacz", I. Kantorski, 2015

[7] I. Kantorski, W. Drozdowski, J. Jurkowski, „Observed light yield of scintillation pixels: Extending the two-ray model", *Optical Materials* **59** (2016), 91-95

[8] E. Hartmann, "Geometry and Algorithms for Computer Aided Design", *skrypt*, Darmstadt University of Technology, 2003

[9] https://wiki.python.org/moin/GlobalInterpreterLock, dostęp z 01.03.2017 r.

[10] https://docs.python.org/2/c-api/index.html, dostęp z 01.03.2017 r.

[11] https://docs.scipy.org/doc/numpy/reference/c-api.html, dostęp z 01.03.2017 r.

[12] G. van Rossum, Python language reference, version 2.7, http://www.python.org, 2016

[13] K. Maher, D. Hünniger, "Basic Physics of Nuclear Medicine", https://en.wikibooks.org/wiki/Basic_Physics_of_Nuclear_Medicine, dostęp z 15.08.2017 r.

[14] E. Roberts, "Ray Tracing", *skrypt*, Stanford University, 1998