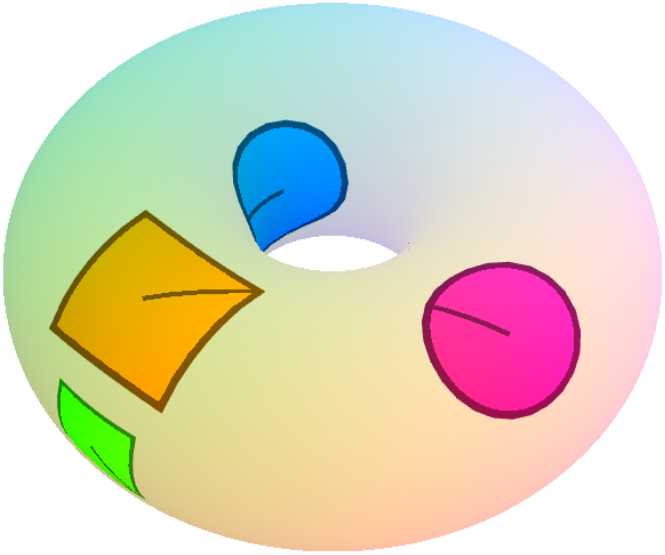
歪んだ空間で物理演算

https://phi16.github.io/Manifold/

phi16 / phi16.256@gmail.com

2017 年 12 月 10 日



概要

一般に剛体物理演算と言えば $\mathbb{R}^2$ もしくは $\mathbb{R}^3$ 上での物体の挙動をシミュレートするものである。そこでなんとなくトーラスや球体などの「歪んだ空間」の上で物理演算したくなった。のでやってみた。物体の扱いなどに注意は必要ではあるものの、物理演算の基礎部分は局所的なモデルなので問題なく多様体上に応用することができた。歪んだ空間の上での動きというものを直感的に理解することができれば嬉しいと思う。

1 導入

特に何か強いモチベーションがあるというわけではなく、唐突にトーラス上で物理演算したくなった。とは言え空間が平坦でない世界というのは結構あるもので、例えば Portal というゲーム [3] においては離れた壁を文字通りつなげることができ、これは距離構造やエネルギーなどが不思議なことになる、歪んだ空間と言える。また私たちは地球という球面の上に居るわけだが、その大陸などの動きは実質球面上の物理演算と言えなくもなくもない。このような世界で物体を動かすのも案外「不自然」ではないのかもしれない。

また、そう考えてみるとラッパ形状の空間での挙動が気になったりする。擬球と呼ばれるラッパ形状の曲面があるのだが、その上で物体が動いたときに尖っている方に移動した場合、先端のほうが「狭い」ことから「世界に詰まってしまう」と考えられる。これは物体の大きさが世界の大きさ程になってしまう、特異な例とも言える。このような空間もシミュレートすることを実現したい。もちろん空間は色々差し替えたりしたいので、一般的な形で記述できることが望ましいわけである。

目次

2　理論

2.1　リーマン幾何

まず「歪んだ空間」とは何か、ということについて考えたい。物理演算ができるレベルであって欲しいので歪みすぎてもいけない。そこで出てくるのが**多様体**である。これは何かというと「空間の各点の周りでは $\mathbb{R}^n$ と同型」な空間である。各点の周り、というのは無限にその点にズームした状態みたいなものを考えれば良い。つまり球面やトーラスはこれによって 2 次元多様体になる。この上で色んな道具をかき集めると大きさを考えたり点を動かしたりできるようになる。ざっと色んな概念について書いてみる。なお、以降微分したそうな関数は微分できるものとする。

定義 2.1 (位相空間)**.** 集合 X が**位相空間**であるとは、各点 $x \in X$ に対して x の近傍の集まりを返す写像 $\mathfrak{N} : X \to \mathfrak{P}\mathfrak{P}X$ を伴い、空間っぽい条件を満たすことを言う。

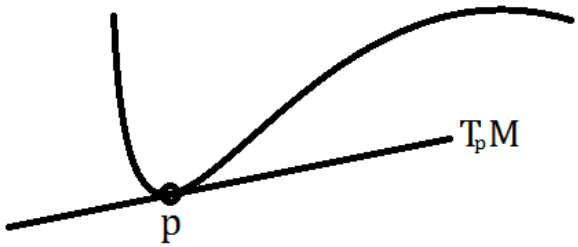
ここで $\mathfrak{P}X$ は X の冪集合。近傍というのはその点を含む領域みたいなものである。まともな詳細はこの辺 [4] に色々書いてある。あんまり気にしなくて良い。

定義 2.2 (多様体)**.** 位相空間 M が n **次元多様体**であるとは、各点 $p \in M$ についてそのある開近傍 U から $\mathbb{R}^n$ への同相写像 φ があり、このような全ての**座標付き近傍**がいい感じに両立することを言う。

「いい感じに両立」というのは異なる座標付き近傍を取っても見える空間は変わらないということである。まぁ多様体というのは端的に言えば各点の周りがまともに $\mathbb{R}^n$ な空間である。

定義 2.3 (接ベクトル空間)**.** n 次元多様体 M の各点 $p \in M$ において、その点に接する n 次元平面を考える。この平面は $\mathbb{R}^n$ と同相っぽく、点 p における「向きと大きさ」を表しているものと言える。この平面の成す線形空間 $T_p(M)$ を**接ベクトル空間**と呼ぶ。

図 1　接ベクトル空間のイメージ ($n = 1$)



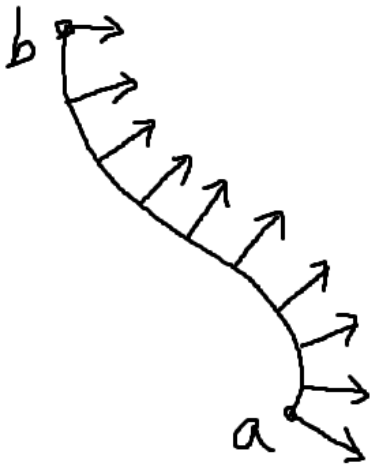
定義 2.4 (ベクトル場)**.** n 次元多様体 M 上の**ベクトル場**とは、M の各点 p に対してなめらかに $T_p(M)$ の元を与える写像のことである。M 上のベクトル場全体集合を $\mathfrak{X}(M)$ と書く。

M 上を走る点があったときに、そのある時点での速度は接ベクトル空間上の元として扱える。また曲線上の各点にその点における移動方向を与えればベクトル場の一部っぽいものができる。しかしここまでの道具では点をベクトル場に沿って移動させたときに**向きを保つ**とは限らない。

定義 2.5 (アフィン接続)**.** 多様体 M における**アフィン接続** ∇ とは、ベクトル場 X に対して「各点で定められた接ベクトル Y_p の向きに微分」することでベクトル場 $\nabla_Y X$ を定めるような方法を与えるものである。

つまりベクトル場が微分できるようになったことを言う。この時、ある曲線 c の上で $\nabla_{\dot{c}}X = 0$ であるということは c 上でベクトル場が変化していないように見えることを表している。実はこのようなベクトル場 X を曲線から作ることができ、これによってある c 上の点における接ベクトルを向きを保ったまま曲線 c に沿って「輸送」できるようになる。これを**平行移動**という。これで初めて「向きを保って移動する」ということができるようになった。

図 2　平行移動の例。曲線に乗っている人からすると向きが変わっていない。



定義 2.6 (測地線)**.** 多様体 M 上の曲線 c が**測地線**であるとは、$\nabla_{\dot{c}}\dot{c} = 0$ であることを言う。

つまり向きを変えずに多様体上を「まっすぐ」走る曲線である。これはまさしくある点に速度ベクトルが与えられたときに通る軌道である。というわけで物体の動き方というものがようやくわかってきた気がする。

今までの話は長さの話とかが入ってこなかった。このままだと図形の大きさなどがわからないのでそのあたりの情報をもらうことにする。

定義 2.7 (リーマン多様体)**.** 多様体 M が**リーマン多様体**であるとは、各点 $p \in M$ において良さみのある内積っぽい写像 (**計量**) $g : T_p(M) \times T_p(M) \to \mathbb{R}$ が伴っていることを言う。

内積があれば角度や長さが測れる。更にこの時ありえん良さみが深いアフィン接続を取れることが知られている。

定義 2.8 (レヴィ・チヴィタ接続)**.** リーマン多様体 M に対し、その計量構造といい感じに両立するようなアフィン接続が一意存在する。これを**レヴィ・チヴィタ接続**と呼ぶ。

というわけでリーマン多様体の上なら細かいことを気にせず点を動かすことができるようになるわけである。話が長い。ちなみに 2 点間の最短経路を求める方法は一般には存在しないことに注意してほしい。

2.2　距離場

話は一気に変わって、世界を方程式で書き記す方法について考える。以降 $\mathbb{R}^3$ 内で考える。関数 $f : \mathbb{R}^3 \to \mathbb{R}$ があったとき、集合 $\{p \in \mathbb{R}^3 \mid f(p) = 0\}$ を考えてみるとこれはだいたい曲面を表す。例えば $f(x, y, z) = x^2 + y^2 + z^2 - 1$ は単位球面を表し、$f(x, y, z) = x$ は $x = 0$ の成す平面を表す。曲面を表さないこともあるが、気にしないことにする。

曲面上を動く点の速度を考える時、各点での接平面が必要である。ある点 p を通る $\mathbb{R}^3$ 上の平面はある法線ベクトル n を用いて「n に直交する接ベクトル全体」と表せるため、n を f から求まれば十分である。ところでスカラー場があったとき、その等位曲面の法線方向は勾配で表せることが知られている。というわけでだいたい $n = \operatorname{grad} f$ と言えるが、折角なので f はさらに性質が良いとありがたい。そこで、距離場の概念を導入する。

定義 2.9 (距離場)**.** $f : \mathbb{R}^3 \to \mathbb{R}$ が $S \subset \mathbb{R}^3$ の距離場であるとは、

$$f(p) = \begin{cases} d(p, S) & \text{if } p \notin S \\ -d(p, \mathbb{R}^3 \setminus S) & \text{otherwise} \end{cases}$$

であることを言う。ここで $d(p, S) = \inf\{d(p, s) \mid s \in S\}$ であり、$d(p, s)$ は点 p と s の距離である。

これを考える理由としては、曲面上を点が動くことを計算機上で実現する際、その点の動きは離散的にならざるを得ないことがある。よってどうやっても「曲面上を吸い付いたように移動する」のは不可能であり、ちょっと曲面から浮いたり沈んだりしつつそれを補正しながら進んでいくことになる。よってその浮いた量・沈んだ量が必要となり、距離場はまさしくそれを与えてくれる。また距離場は曲面を可視化する際にも非常に有用であることが後に言える。なお、距離は大まかに合っていれば反復計算によってその誤差を減らして真の曲面上の点に近づけることができるため、f が正確に距離場である必要はない。

実際の距離場の例を挙げる。

- x は平面 $x = 0$ の距離場である。
- $\sqrt{x^2 + y^2 + z^2} - 1$ は単位球面の距離場である。
- $\|(\sqrt{x^2 + z^2} - r_1, y)\| - r_2$ は大半径 r_1、小半径 r_2 のトーラスの距離場である。
- $\sqrt{x^2 + z^2} - \exp(y)$ は擬球に似たラッパ形状の曲面の擬似距離場を与える。

この距離場によって生成される曲面は 2 次元多様体である。またこれは $\mathbb{R}^3$ 内の曲面であるため、各点における接ベクトル空間はその点における $\mathbb{R}^3$ の接ベクトル空間 $T_p\mathbb{R}^3 \simeq \mathbb{R}^3$ に素直に「嵌め込む」ことが出来る。この上で内積を計算することによって、曲面の接ベクトル空間に計量を入れることが出来る (標準計量の引き戻し、と表現される)。よってこれはリーマン多様体になる。また各点における法線は自然に定まっているため、面に「向き」が定められていると考えることが出来る。これはメビウスの輪のような「中身のある立体の表面の一部として表せないもの (向き付け不可能なもの) 」は表現できないということである。そのような曲面を再現したくもあったが、様々な理由から距離場を使いたかったので諦めた。というわけで、距離場を使うことによって 2 次元向き付きリーマン多様体で計量が $\mathbb{R}^3$ の標準計量の引き戻しで与えられているものを扱うことにする。

以降、考える対象/空間/世界は $\mathbb{R}^3$ の上の距離場で与えられた曲面 M とする。

2.3 図形の扱い

元々多様体というのは「各点」における挙動を調べるものであり、大きさを持った図形などを扱えるものではない。だが今回は物理演算するという目的のために図形を扱う必要がある。しかし例えば集合として図形を定義してしまうと、歪んだ空間では平行移動するだけで大きさが変わってしまう可能性がある。リーマン多様体では点によって曲率 (空間の曲がり具合) が異なるかもしれないわけだが、悲しいことに曲率の異なる点の近傍間には等長写像が存在しないのである (これが**驚異の定理** [5] というやつである)。つまり球や円筒の側面くらいでないと「正しい」物理法則はまず成り立たない。とは言え今回はシミュレーションが楽しければ良いわけで、いい感じの図形の定義を考えれば十分である。

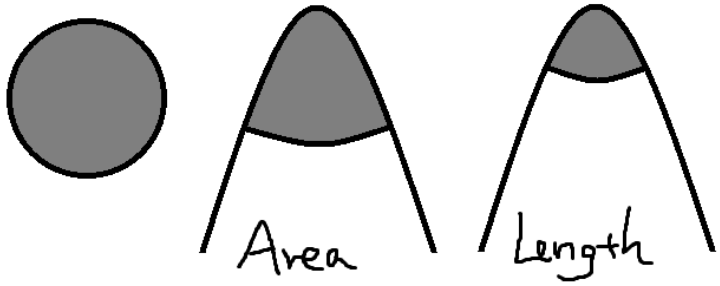
そこで、大きさをできるだけ保つべきという気持ちのもと、次の定義を採用することにした。

定義 2.10 (図形)**. 図形**とは、図形中心 $p \in \mathbb{R}^3$ と軸方向 $a \in T_p(M)$ および**形状** $s : S^1 \to \mathbb{R}^+$ から成るものである。ここで S^1 は単位円、$\mathbb{R}^+$ は正の実数全体集合である。

原点を含む凸な 2 次元図形の境界は、極座標上の方程式で $r = f(\theta)$ と表せる。逆にこれを形状として扱って図形を定義するのである。図形中心から θ 方向には、測地線を辿って長さ r のところに境界がある、ということになる。例えば $r = 1$ は半径 1 の円を表し、同様にして正 n 角形を表現することもできる。そして図形が動いたとしても形状は不変とすることで、保持する情報を 6 つの $\mathbb{R}$ に抑えることができる。後はこれが「良く振る舞う」かどうかである。

まずこの方法では図形が別の場所に移動したとしても、見かけ上の直径が不変である。これは形状が長さによって記述されているためである。しかし面積は保存しない。物体が動いている様子としては長さを保存したほうが個人的にはそれっぽい気がする (図 3) ので、これで良いことにする。また図形中心を重心と捉えれば図形が動く様子は測地線に沿って図形中心が移動し、軸方向が平行移動する様子に他ならない。平坦な平面において図形中心が重心であることは $\int_{S^1} r^2 e^{i\theta} d\theta = 0$ と表現できるが、逆にこれを満たしていれば歪んだ空間でも重心のように振る舞うと考えることにする。これは不自然な仮定ではない。気がする。

図 3　面積を保つ場合と長さを保つ場合、放物面上。 (イメージ)



実際に図形の扱いが定まってきたので、物理法則を考えることにする。

2.4　物体の運動

まず、物体を次で定義する。

定義 2.11 (物体)**.** **物体**とは、図形とその図形中心 p における速度 $v \in T_p(M) \simeq \mathbb{R}^3$、また現在角度 $\theta \in S^1$ とその角速度 $\omega \in T_\theta(S^1) \simeq \mathbb{R}$、及び質量 $m \in \mathbb{R}^+$ と慣性モーメント $I \in \mathbb{R}^+$ を持った構造である。

平坦な平面上で 2 次元図形の運動を表現する際は位置 $\mathbb{R}^2 \times S^1$ と速度 $\mathbb{R}^2 \times \mathbb{R}$ があれば十分であり、その通りの定義になっている。次に時間が dt 過ぎたときの物体の様子を調べたい。位置は M の元で速度は $T_p(M)$ の元であるため直接足し合わせることはできないが、曲面が $\mathbb{R}^3$ に埋め込まれていると考えれば位置は $\mathbb{R}^3$ の元で速度は $T_p(\mathbb{R}^3) \simeq \mathbb{R}^3$ の元と考えることが出来る。これは一般的な位置ベクトルと速度ベクトルによる表現と一致し、$p' = p + vdt$ を求られる。ただしこのままでは $p' \in M$ とは限らないため、前述の通り距離場を用いて曲面に乗るように補正することになる。これで物体を移動することが出来た。

平坦な空間では速度ベクトル $v \in T_p(M)$ は外力が無ければ不変で良かったが、物体が p が p' に微小移動したときの速度ベクトルは $T_{p'}(M)$ の元であるはずである。2 つの接ベクトル空間 $T_p(M), T_{p'}(M)$ の間には直接の関係は無いため、∇ を用いて v を p から p' に輸送する必要が出て来る。これを計算する方法として、次の定理が使える。

定理 2.1. リーマン多様体 N について、埋め込み $i : N \to M$ が存在するとする。これによって N 上のベクトル場を M 上のベクトル場とみなすことができ、また接ベクトル空間の間の直交射影 $\pi_p : T_p(M) \to T_p(N)$ を得る。それぞれからベクトル場についての写像 $\iota : \mathfrak{X}(N) \to \mathfrak{X}(M), \pi : \mathfrak{X}(M) \to \mathfrak{X}(N)$ が得られる。

このとき N, M 上でのレヴィ・チヴィタ接続を $\nabla, \tilde{\nabla}$ とおくと、ベクトル場 $X, Y \in \mathfrak{X}(N)$ について

$$\nabla_Y X = \pi\left(\tilde{\nabla}_{\iota(Y)} \iota(X)\right).$$

この辺 [6] に証明が載っているが、ここから何が言いたいかというと「ベクトルを曲線に沿って微小平行移動するには、一度 $T_p(\mathbb{R}^3)$ の上で考えてから目的の接ベクトル空間に直交射影すれば良い」ということである。ただしこのときベクトルの大きさが変わってしまうのは意図しないと考えられるため、後に補正することにした。これによって物体を正しく移動できるようになった。

角度と角速度の適用は独立して行えると仮定しているので単純に $\theta + \omega dt$ して良いとした。質量は不変であるとしているため、これによって物体を (外力がない環境で) 適切に運動させることが出来るようになった。

重力などを扱うためには「力」を適切に物体に適用させる必要がある。力の向きは接ベクトル空間の元で与えられるべきではあるが、全体空間を知っているために力の向きを $\mathbb{R}^3$ の元として扱うことにする。力の大きさの $1/m$ 倍が加速度になり、これを前と同様に $\mathbb{R}^3$ 上とみなした速度ベクトルに加算して接ベクトル空間に射影することで、力を適用させることが出来る。重力として例えば $(0,0,1)$ を与えれば物体は南極に集まるようになり、また $(-y,x,0)$ を与えれば $z=0$ を軸として加速しながら回転するようになる。

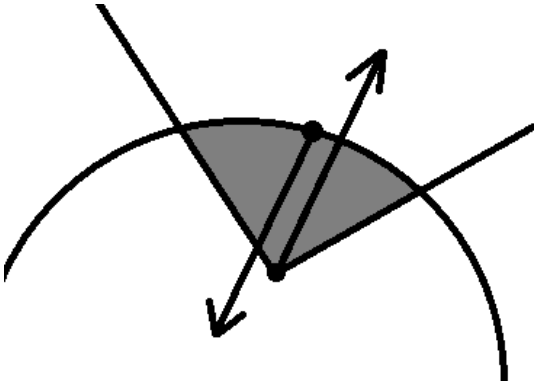
2.5 物体の衝突と拘束の解消

今までの方法で物体を空間の上ですいすい動かすことはできるが、複数の物体間には何の関連も存在しない。もちろん物理演算するからにはこれらは衝突し、お互いに速度に影響を及ぼすはずである。これには一般的な物理演算の方法をこの歪んだ空間上で再現する必要がある。

一般的な物理演算における相互干渉の考え方は「速度に条件を与えて解を調べる」というものである。衝突したときには「衝突点における相対速度と衝突した方向との内積が 0 以上」という条件を追加する。これはこれ以上埋め込まないということを表し、このような条件のことを**拘束**と呼ぶ。この時の拘束はある $J,J' \in \mathbb{R}^4$ によって $J\cdot(v,\omega)=J\cdot(v',\omega')$ と表わせ、色々頑張る [7] とだいたい解ける。この辺の話は丁寧にやってない上メインの話ではないので割愛する。

問題は「衝突点」と「相対速度」と「衝突した方向」である。元々局所的な扱いしかできない多様体では、衝突点を図形の性質から直接求めることは出来ない。そこで計算するためには実際に測地線を辿って物体が存在する領域を計算し、それらの重なりを調べることになる。重なりがわかった時には「最も短い距離でめり込みを解消できる方向」を計算することで各物体上の衝突点を得ることが出来る (図 4)。またこの点における速度は、計量が曲がっている為に $v+r\omega$ では無い。しかし図形が十分小さければそこまでの差は無いだろうと仮定し、この式を用いている。最後に衝突した方向をめり込み解消方向とし、これらの情報から拘束を生成している。

図 4　重なった領域・衝突点・めり込み解消方向



これらからわかるように多くの仮定を置いているため、元々良くない配置にあるとき (重なっているなど) にはうまく計算を行うことができない。ただし実際に実装してみたところ殆どうまく動いたため、これでいいことにする (元々が無理のあるモデルなのである)。

2.6 境界を持った空間

半球など、途中で空間に壁があるような空間を扱いたいこともある。距離場で与えられる多様体は基本境界が無く、無理に表現してしまうと特異点の上で物体が吹き飛んでしまう。そこで物体が動ける範囲を $b(p)\geq 0$ で与えることとし、これをはみ出す点に対して $\operatorname{grad} b$ 方向に拘束条件を追加する。これはまさしく壁であり、「境界を持たない多様体の一部」として境界を持った空間を表現していることになる。これによって半球や平面上の矩形・円などの上で物体を動かせることができるようになった。

3 実装

以上の考察から、実際にプログラムを組んでシミュレートするプログラムを作成した。それらの詳細について述べる。

3.1 Haskell と JS

今回、プログラムは Haskell と JavaScript の 2 つの言語を用いて作成した。これには様々な理由がある。

- 私は Haskell と JavaScript が (相反するようだが) どちらも好き
- Haskell を JavaScript にコンパイルするライブラリ haste-compiler[8] を何度か使ったことがある
- 描画などは HTML5 による Canvas/WebGL が圧倒的に便利である
- データの表現や演算を行うには Haskell のほうが圧倒的に便利である
- 何よりも次章の自動微分を行うために Haskell が必要であった
- 速度面が気になるならそこだけ JavaScript を書くということもできる
- またソフトウェアの公開に関しても JavaScript は優位にある
- というわけでもはや Haskell と JavaScript を共存させて使わない理由がない

ということである。巨大なソフトウェアになるなら流石に速度面に厳しいものがあるが、今回はそんなに大変なことでもないだろうと考えてこの構成でプログラムを作成した。最終的には 60FPS が出ない程度には重くはなったものの、私の PC で動かすには何も問題無かった為工大祭の展示には十分であった。最終的なプログラムの構造は以下である。

- 世界のデータは全て Haskell 側で管理・演算されている
- WebGL/Canvas による描画処理は JS で書かれており、Haskell 側から描画データを与えている
- 当たり判定処理は Haskell で書いたところめちゃくちゃ重くなったので、JavaScript で書き直した

なお JavaScript 側のソースコードは継ぎ足し継ぎ足しで書いたのでだいぶ汚い。というか JavaScript コードは構造化を全くしていない。メイン部分は Haskell による処理ということで許してほしい。

3.2 自動微分

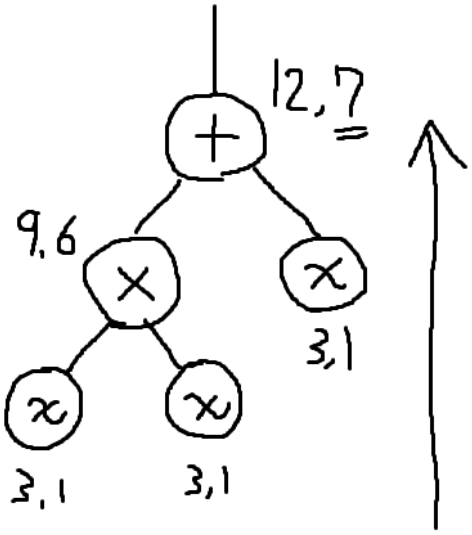
距離場 f を与えたときに物体を移動させることができるが、このときに $\mathrm{grad}\, f$ を必要とする。明らかに f から $\mathrm{grad}\, f$ は決定されるが、プログラムにおいてはこれは自明ではない。単純な線形変換などでは $\mathrm{grad}\, f$ を構成できず、微分が必要なのである。$(f(x+\varepsilon)-f(x-\varepsilon))/2\varepsilon$ などで近似的に微分を計算する方法もあるが、これは完全ではない。本質的に式から微分が決定されることから、プログラム側で $\mathrm{grad}\, f$ を生成できるはずである。それを実現するのが**自動微分**という手法である。私たちは複雑な関数の微分方法を知っているわけで、それを再現しようというものである。

重要なのは「式に情報が詰まっていること」である。例えば C 言語で `float` から `float` を返す関数を作ったとしても、それに自動微分は適用できない。もしかしたら内部で副作用を行っているかもしれないし、`if` を使って不連続な関数を構成できるかもしれない。そこで、**多相関数**を使う。これは「条件を満たしている型なら適用可能」な関数のことで、いわゆる C++ の template・Java の interface・Haskell の型クラス・Rust の trait のことを指している。「任意の」条件を満たす型なら利用できるということは、その条件を用いた式「しか」記述できないということである。これは非常に強い性質で、例えば全ての型を受理する関数はその値に一切触れないことが従う。この辺の話は [9] にあったりする。今回はある程度の数式のみを使える型とすることで、「微分のできる関数」しか記述できないようにした。といっても数学的に厳密なものではなく、例えば `abs` は使える (原点での微分係数を 0 と定義している)。自然な方法でプログラムが書ければそれでいいのである。

実装はそんなに難しくない。普段通りに関数を計算しつつ、微分値を計算する機構を取り付けた型を用意すれば良い。Haskell での簡単な例を載せる。

```
f :: Num a => a -> a
f x = x * x + x

data Expr a = Expr a a

instance Num a => Num (Expr a) where
  Expr x x' + Expr y y' = Expr (x+y) (x'+y')
  Expr x x' * Expr y y' = Expr (x*y) (x*y' + x'*y)
  fromInteger i = Expr (fromInteger i) 0

grad :: Num a => (Expr a -> Expr a) -> a -> a
grad f p = let
    Expr _ diff = f (Expr p 1)
  in diff

main = print $ grad f 3 -- Output: 7 (= 2x+1 = x*1 + 1*x + 1)
```

図 5　自動微分の内部処理 (この計算方法はボトムアップ型と呼ばれる)



`Expr a` 型における 2 つ目の `a` は微分値を保持している。`grad` で与えた 1 は x の微分値 1 のことである。合成関数の微分 $(f(x))' = x' \cdot f'(x)$ を考えれば、この 2 つの値で複雑な関数の微分も表現できることがわかる。後はこの機構を `sin`, `cos` や `min`, `max` に適用できるようにし、さらに 3 次元のスカラー場を扱えるように変更を入れる。雑に説明するとスカラー場は `f a -> a` で表現でき、微分はベクトル場なので `data Expr a = Expr a (f a)` になるというところである。これらによって $\mathrm{grad}\, f$ の導出が完成する。上のコードが理解できていれば実際の計算部分のコード[*1]を見たほうが早い。

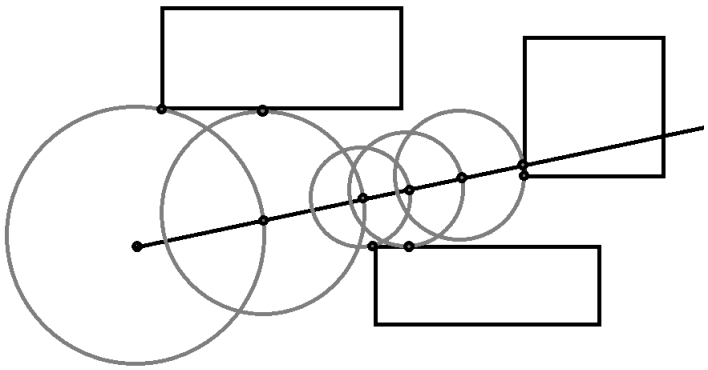
f から $\mathrm{grad}\, f$ が求まったことで、表面付近の点に対する「表面までの距離」を $f/\mathrm{grad}\, f$ で評価できるようになった (これは一次近似であり、もともと「正しい距離場」ならば f と一致する)。また表面の各点における法線ベクトルを $\mathrm{grad}\, f/\|\mathrm{grad}\, f\|$ で計算できる。よって物体を動かすには十分な情報が、全て距離場から自動生成できたことになる。これによって空間の形状を気軽に変更できるようになり、様々な空間をテストすることが容易になった。うれしい。

[*1] https://github.com/phi16/Manifold/blob/develop/Lib/AD.hs

3.3　空間のレンダリング

距離場を使うことで得られる大きな利益がレンダリングである。実は Sphere Tracing という手法が知られており [10][11][12]、これを用いると「距離場で表現された表面をある程度高速にレイトレーシング」できる。原理は「表面との距離が d なら、d 進むまでには表面と衝突しない」である。これを使って現在点から段々と前に進む (レイマーチングする) ことで最終的に表面に衝突したり、無限遠に消えることを判定できる。レンダリングするにはこれを描画領域全点で行えば良く、OpenGL のシェーダを用いて実装している (とてもはやい)。

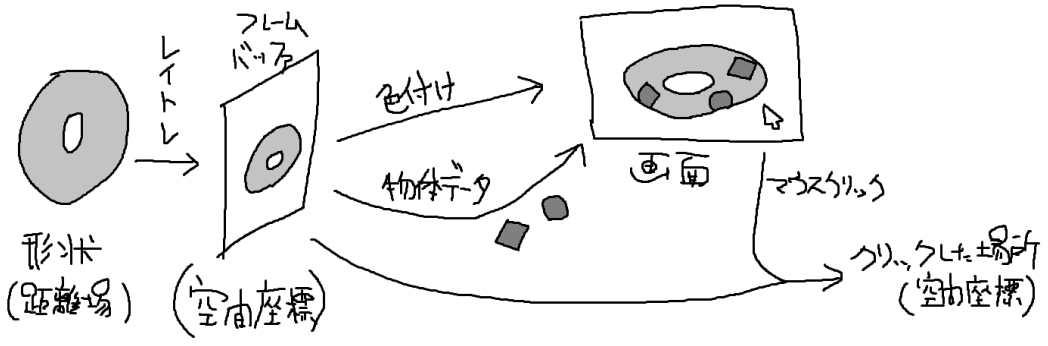
図 6　Sphere Tracing の様子。最初にレイの方向を決め、位置を移動していく。



これを行うためには距離場をシェーダ内部に記述しなければならない。しかし私達が書いた距離場は Haskell コードであり、もちろん単純にはシェーダから呼ぶことはできない。そこでまた距離場が多相関数であることを利用し、距離場からシェーダのコードを生成する機構を作成した[*2]。実装は自動微分よりも単純で、単に演算子や関数の定義が文字列の構成になっているだけである。かんたん。また勾配もシェーダコードになっているのでレンダリング時に法線を得ることもできる。今回は特に使っていない (まともなレンダリングをしようとすれば必要になるが、今回の色付けは適当である (頂点の位置を RGB と解釈しているだけ))。

また、後述の物体の描画を行うためにはレンダリング時にこの表面の情報が必要になる。一度 Sphere Tracing を行ったものをまた各物体に対して行うのは無駄なため、実際には空間を Sphere Tracing したときには画面ではなくフレームバッファというものに描画を行い、その後に表面を描画する部分と各物体を描画する部分でフレームバッファの値を再利用している。さらに、マウスで物体を引っ張りたいと思ったときにはマウスが表面のどこを触っているかを知る必要があるのだが、それはフレームバッファが知っている。ということでこれを流用してマウスと表面の当たり判定を簡単に実装できている。フレームバッファさまさまである。

図 7　フレームバッファの利用の流れ

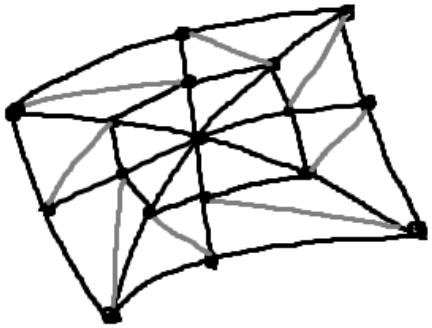


[*2] https://github.com/phi16/Manifold/blob/develop/Lib/Shader.hs

3.4　物体のポリゴン化と描画・当たり判定

物体が実際にどこにあるのかを調べるために、実際にポリゴンを生成する。これは 24 方向に 4 回測地線に沿って離散的に点を取り、実際に得た点の座標群に対してポリゴンを貼ることで行っている。辿る長さは単純に $r/4$ で計算してる為正しい大きさにはならないが、気にしないことにする。

図 8　曲面上の正方形から生成されるポリゴン (8 方向 2 分割)



これを描画するには射影変換を施す必要があるが、空間表面に合わせて描画する必要があるため、Sphere Tracing を行った際のレイのパラメータと合わせる必要がある。レイのパラメータと射影変換はアスペクト比と視野角 (FoV) でほとんど決定できるため、これらを元に計算すれば良い。

```
// ray parameter (world renderer, fragment shader)
vec3 dir = normalize(vec3(uv.x,uv.y,1./tan(fov*pi/180.)));

// projection parameter (object renderer, vertex shader)
vec3 p = transform * (coord - camera);
p.x *= 3./4.;
p.xy /= tan(fov*pi/180.);
```

レンダリングを行う位置はわかったものの、このまま描画を行うと空間の裏側にある物体も描画されてしまう。だがカリングを行ってしまうと表面の境界丁度にある物体が綺麗にレンダリングされない。そこで、空間の情報を用いて裏にあるかどうかをピクセルごとに判定することにした。先の通り私たちはその描画対象の点における「ポリゴンの奥方向の位置」と「表面の奥方向の位置」を知っているので、「ポリゴンが前にある」という条件で描画できそうである。しかし、これではポリゴンが埋まってしまうのである。これは曲面が歪んでいるため、図 9 の左側のようにポリゴンが表面の裏側にいる場合が多いからである。その対処としては単純にポリゴンの位置を 0.1 手前にずらして計算することで行った。この値は試行錯誤の結果であり、これによって実際に境界部分も概ね綺麗に描画できていることがわかる (図 10)。

図 9　単純計算と補正後

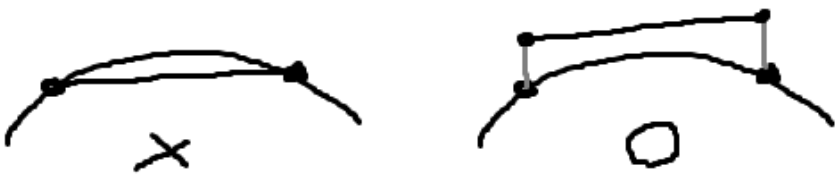
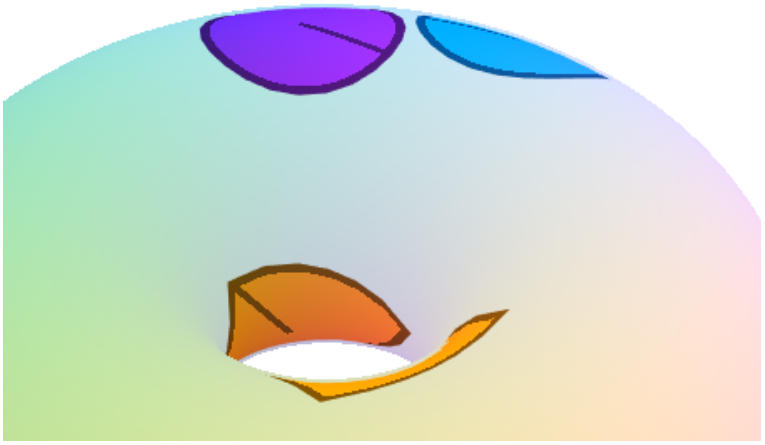


図 10　境界付近のポリゴンの様子



また、ポリゴンが生成できたことで実際に物体と物体が重なっているかどうかを「重なるポリゴン対があるか」で判定できるようになった。そして物理的な拘束条件を生成するにもまたこのポリゴンベースで求めた値が使える。これによって物体の相互干渉が実装され、目的であった「歪んだ空間で物理演算」を完全に達成することができるようになった。

4　おまけ

基本は以上だが、ちょこちょことおはなしがある。

4.1　位置と角度

物理演算では位置と角度が分離されて扱われている。気がする。でもそんな必要は無いはずである。一般化された座標概念と速度概念さえあれば力学はできる。そしてその方が本質を突いていると、私は思う。今回のプログラムは実際にその考えで以って記述されている。

位置を表す `Pos` 型は、座標 $\mathbb{R}^3$ と角度 S^1 から成る。その表現は $\mathbb{R}^3 \times \mathbb{R}$ である。また速度は $T_p\mathbb{R}^3 \times T_pS^1 \simeq \mathbb{R}^3 \times \mathbb{R}$ であり、これもまた `Pos` 型として管理している (本質からは少し外れた)。この型には各要素毎の加法や乗法などが定義されている他、空間の基底や内積までもが用意されている。もともと速度と角速度をそれぞれで内積を取って和を取るような処理が物理演算の方で行われていたが、それをさらに単純に表現できることになる。そしておそらくそれが正しい。解析力学をやればわかるようになりそう。

今回は 2 次元だからこれができた、というのは違う。実際本来は 2 次元でもできるものではない：角度 S^1 と角速度 $\mathbb{R}$ は明らかに異なる空間である。3 次元での角度は姿勢 S^3 になり、角速度は回転ベクトル $\mathbb{R}^3$ になる。ただそれだけである。もともと異なる空間である位置と速度は異なる扱いを受けるべきであり、$p + vdt$ という数式には巨大な行間が存在する。一般の空間では位置の表現に加法が備わっているとは限らないのである。

まぁなんであれ、今回では物体の移動が `coord += veloc * dt` と書け、質量が `mass` = (m, I) と書け、力による速度の更新は `veloc += force / mass` と書ける。うれしい。というかそうあるべきである。

4.2　詰まるラッパ

狭い場所に物体が入ると、詰まる。これはそれ以上進むには壁から拘束が掛かるからである (図 11)。3 次元での円錐形表面の空間に物体がある時、それもまたある程度進むと詰まるはずである。詰まる原因は**自身との衝突**である。先が細すぎると物体自体が空間全体の大きさほどになってしまい、空間を覆ってしまうのである。というわけでこれを再現したい。

図 11　2 次元のラッパ

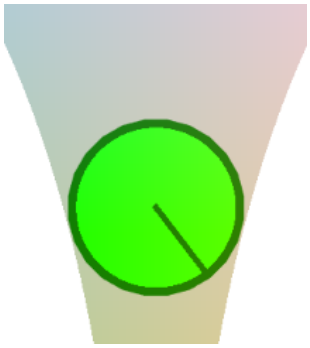
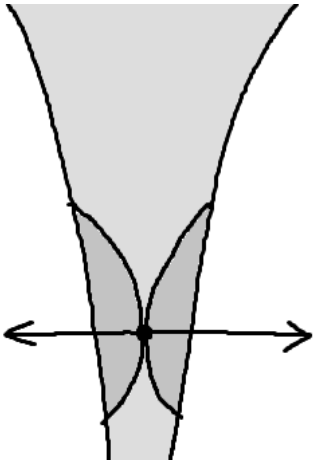


図 12　3 次元のラッパ

だが、実際に動かしてみるとガタガタした後下に沈んでいく。何故ならこの物体は (ラッパの軸方向を縦として) 横向きに衝突しているわけだが、衝突点に課せられた制約条件は完全に横向きであり、縦方向に関する速度は一切変化しないからである (図 13)。これはどうしようもない。

図 13　制約条件の方向



というわけで「自身と衝突した場合、本来の制約条件とは垂直方向に制約を加える」ことにした。自身と衝突するのはこのように世界を覆ってしまう場合のみで、そうなれば本来の制約条件は何の意味も持たない。逆に「世界を覆う直前」に戻すのが正解であると考えられるが、これは物体が覆っている向きに対する垂直方向である、ということである。また自身に関する拘束は、適用対象が同じであるために実質的に 2 倍の強さで掛かる。これはおそらく嬉しくないと考えたので、雑に 0.6 倍することにした。以上によって確かに物体は「詰まり」、想定していた挙動を実現できた。

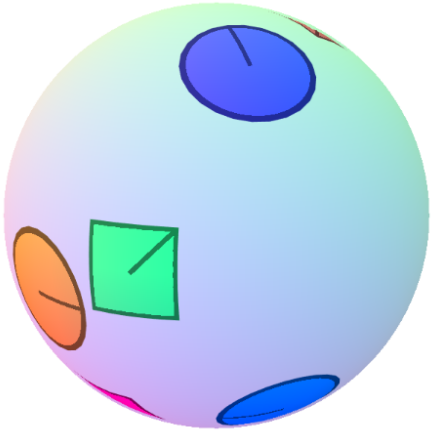
ところでラッパ形状だと物体がよく上がってくる。上がってくるというのは一度狭い部分に物体が移動しようとするものの、何故か詰まる前にふわっと上に戻ってくるのである。ある程度の速度で横向きに動いている物体は、外向きに力が入るせいで重力があるにもかかわらず落ちてこない。なんだかポテンシャルらへんの議論ができそうな気がする。詳しいことはわからないが双曲面モデルがやばそうだというのを感じる。

4.3　射影空間

この物理演算プログラムを作ろうとした最初のモチベーションは「トーラス上での物理演算」であった。実はこれは 2 次元でも再現できる。左右と上下をそれぞれつなげてしまえば良い。その発想で射影平面を扱ってみたくなったが、これは 2 次元では再現できない。適当に左右と上下をそれぞれ反転させてつなげてみたが謎の特異点がある意味のわからない空間ができてしまった[*3]。というわけで、射影平面をきちんと再現するためには 3 次元で丁寧に球から作る必要があると考え、これを作成することとなったのである。

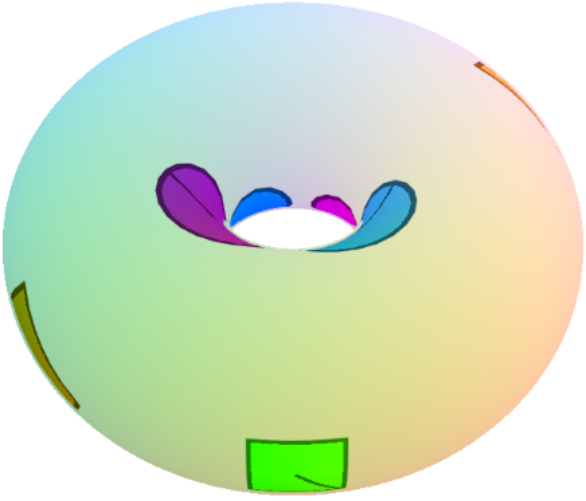
射影平面 $\mathbb{R}\mathrm{P}^2$ を適当に調べると意味のわからない図が大量に出て来るが、「複数箇所に同じ物体がいる」ことを許容すれば実現は簡単である。$S^2 \subset \mathbb{R}^3$ 上での点 p にある物体が同時に $-p$ の位置にもあると考えれば良い。あとはそれに従ってよしなに物理判定やレンダリングを行ってあげれば、これで一応完成する。見た目は球面だが、例えば右に物体を投げれば (裏を通らずに) 即座に左から出てくるように見える。裏にある物体も表から見えるはずで、空間上にある物体すべてを定点から眺められる。たのしい。

図 14　射影平面。赤の四角形が同時に 2 箇所に存在していることがわかる。



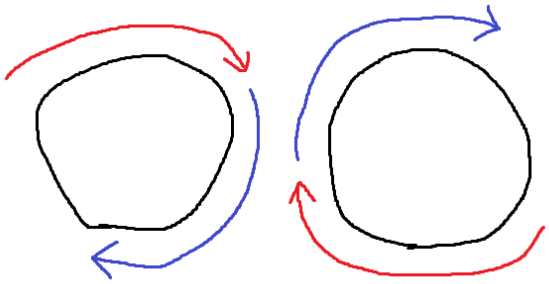
実現方法は「点 p と $-p$ を同一視」なので、原点で点対称な空間なら「射影化」できる。というわけでトーラスも「射影化」してみた。数学的にどんな意味があるかは考えてないが、とりあえずマウスでぐりぐりする分には非直感的な動きをしてくれるのでめちゃくちゃたのしい。

図 15　射影的トーラス。穴付近の挙動が面白い。



[*3] http://tori.phi16.trap.show/

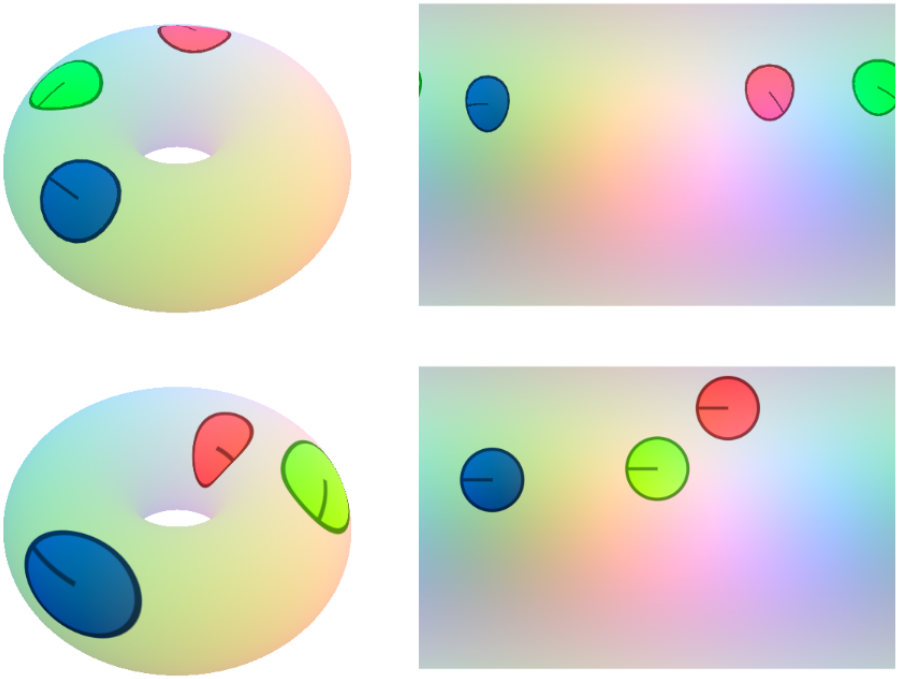
図 16　穴に物体を投げると「すり抜ける」ように見える。



4.4　トーラスの計量の比較

先程言ったように 2 次元でもトーラスは表現できる。だがその空間は 3 次元でのトーラスとは微分同相だがリーマン多様体的には異なる。2 次元で再現したトーラスの空間は平坦なはずで、実際に曲率は常に 0 である。3 次元空間上のトーラスの曲率は外側で正、内側で負であることが見てわかる。というわけで、その違いを可視化しようして出来たのがこれである。

図 17　2 つのトーラス



左上は 3 次元空間上のトーラス、右下は 2 次元で再現された平坦トーラスである。右上は 3 次元トーラスの展開図、左下は平坦トーラスを 3 次元トーラスにマッピングした図である。左上と右下では半径一定であることが見てわかり、右上と左下ではそれぞれ縦向きと横向きに歪んでいることがわかる。内径と外径の「長さ」がそれぞれ異なることで空間の歪みが起きていることもわかる。

ちなみにそれぞれの実装は全く異なり、平坦トーラス用に物理演算を別で作成した (先述の 2 次元謎空間シミュレーションと同じソースである)。この当たり判定は非常に簡単で円と円が本当に重なっているかをいつものように調べればいいだけである。で、その演算結果を用いて平坦面にレンダリングを行い、さらにそれをテクスチャとして 3 次元トーラスにマッピングするということを行っている。ちなみにこの左下の 3 次元トーラスにも所謂 3D モデルみたいなのは存在せず、Sphere Tracing で作られた図形である。

5　まとめ

最初の発想から連想してやりたいと思ったことは全て実現できたので満足している。ただこの方法では最初の通り距離場から定義される「2 次元向き付きリーマン多様体で計量が $\mathbb{R}^3$ の標準計量の引き戻しで与えられているもの」しか扱えない。そうでないと可視化に不便とは言え、いくらか条件を緩めたいという気持ちもある。特にメビウスの輪などをシミュレートするために向き付け不可能な空間も扱えるようにしたいところである。また物理屋さんからは時間軸方向を取り入れられるローレンツ計量を扱えるようにしたらどうかという話も聞いたが、作ったら面白そうではあるものの実装方法がわからない。何かまたアイディアを思いつけば作ったりするかもしれない。

また余談だが、今年は 3Q に幾何学特別講義 C[13] という講義を受講した。これはまさしくリーマン幾何学の授業で、計量や接続などについて学ぶことができた。また 4Q には幾何学特別講義 D[14] という講義があり、測地線などの概念を学んでいる。今回プログラムを作成した際には勘で作ってた部分が多くあったものの (特に直交射影)、それらに対して数学的な意味付けを行えるようになったのはこれらの講義のおかげである。圧倒的感謝。みんなも幾何学やろう。

参考文献

[1] 多様体の基礎, 松本 幸夫
https://www.amazon.co.jp/dp/4130621033

[2] リーマン幾何学, 酒井 隆
https://www.amazon.co.jp/dp/4785313137

[3] Steam : Portal (2017/12/06)
http://store.steampowered.com/app/400/Portal/

[4] 位相の特徴付け - Wikipedia (2017/12/06)
https://ja.wikipedia.org/wiki/%E4%BD%8D%E7%9B%B8%E3%81%AE%E7%89%B9%E5%BE%B4%E4%BB%98%E3%81%91

[5] 驚異の定理 - Wikipedia (2017/12/06)
https://ja.wikipedia.org/wiki/%E9%A9%9A%E7%95%B0%E3%81%AE%E5%AE%9A%E7%90%86

[6] Vector field on riemannian manifold - Mathematics Stack Exchange (2017/12/07)
https://math.stackexchange.com/questions/99395/vector-field-on-riemannian-manifold

[7] ゲーム制作者のための物理シミュレーション 剛体編, 原田 隆宏, 松生 裕史 (2017/12/07)
https://www.amazon.co.jp/dp/4844332821

[8] A GHC-based Haskell to JavaScript compiler — The Haste Compiler
https://haste-lang.org/

[9] Philip Wadler, “Theorems for free!”
https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf

[10] Iñigo Quilez, “Rendering Worlds with Two Triangles”
http://iquilezles.org/www/material/nvscene2008/rwwtt.pdf

[11] modeling with distance functions (2017/12/10)
http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

[12] Benjamin Keinert, “Enhanced Sphere Tracing”
http://erleuchtet.org/~cupe/permanent/enhanced_sphere_tracing.pdf

[13] H29 年度 — 幾何学特別講義 C - TOKYO TECH OCW
http://www.ocw.titech.ac.jp/index.php?module=General&action=T0300&JWC=201707312

[14] H29 年度 — 幾何学特別講義 D - TOKYO TECH OCW
http://www.ocw.titech.ac.jp/index.php?module=General&action=T0300&JWC=201707313