卒業論文
準光速世界で見える風景の擬似撮影

大阪工業大学
情報科学部　情報システム学科
学籍番号　B05-139
葭矢 景淑

2013 年 3 月 4 日

目 次

1 序論

1.1 背景

相対性理論では、光速に比べて無視できない速度で運動すると、様々な効果が起こることがわかっている。例えば、運動方向に空間が縮んで見えるローレンツ収縮や、質量の増加などがある。有名なのは時間の進みが遅くなることだろう。光に近い速度で飛ぶロケットで宇宙旅行から帰ってくると、地球では何十年も時間が進んでいるため、未来に行くことができるという話もある。また、浦島太郎という童話では、亀を助けた浦島太郎が竜宮城から帰ってくると、知っている人は誰もいなくなるほど時間が経過していたのは、竜宮城や亀が光速に近い速度で運動していたのではないかという話もある。

1.2 本研究の目的

この研究では、相対性理論の中でも視覚に作用する効果に注目し準光速世界をシミュレーションし、画像として出力することで、準光速世界の擬似撮影をするのが目的である。

準光速運動をした時の可視化シミュレーションをする上で、どのような効果が必要か考えてみる。現段階でわかっている効果は、色の変化が起こる「光のドップラー効果」と「横ドップラー効果」、視野の変化及び明るさ変化が起こる「光行差」、景色そのものの変化が起こる「ローレンツ収縮」及び見え方の変化による「テレル回転」である。$3D$ で簡単な街を作成し、その中で景色の形状変化と色の変化、そして視野の変化及び明るさの変化を組み合わせ、準光速世界のシミュレーションを行う。最終的な $3D$ のシミュレーションには OpenGL を使用した。また、詳しい効果やシミュレーションに必要な式を説明する前に、これらの効果を簡単に説明する。

1.3 光とは

本研究では、観測者にはどのように見えるかを考える。そのため、可視化シミュレーションに必要な効果を説明する前に、光について簡単に説明しておく。

一般に光とは、人が特定の波長の電磁波を人の目が捉えることで明るさとして認識できる可視光線のことである。この可視光線の範囲内で電磁波が変化すると、見える光の色も変化する。他にも、太陽光や電気による光は、電磁波の形で伝播する放射エネルギーのこととする場合もある [1]。しかし、本研究では、人の視覚にどのように捉えられるかまでを研究テーマに取り入れるため、電磁波の内、人が明るさとして認識できる特定の波長である可視光のことを光とする。

では、この特定の範囲外の波長の電磁波は何なのか。可視光線の他にも、特定の範囲内の波長を持つ電磁波には様々な呼び方がついている。図 1 にあるように、可視光線より波長が長い電磁波を赤外線 (IR) と呼び、さらに波長が長い電磁波はマイクロ波と呼ばれている。逆に、可視光線より波長が短い電磁波を紫外線 (UV) と呼び、さらに波長が短くなっていく

と X 線や γ 線になる。これらの電磁波は人の目には見えない。図 1 を見ると分かるように、人の目に見える可視光線の範囲は、電磁波の中でも極めて狭い範囲である。同じ電磁波でも可視光線のみを、光として人が認識できる原因は、はっきりとは解っていない。恐らく、太陽光の下では、この可視光線の波長を持つ電磁波が多かったため、長年の進化の中で光として認識できるようになったのだろう。また、近年の技術では、赤外線を可視化する赤外線カメラなどがある。赤外線や X 線も目には見えなくとも、光の仲間であるという良い例だ。

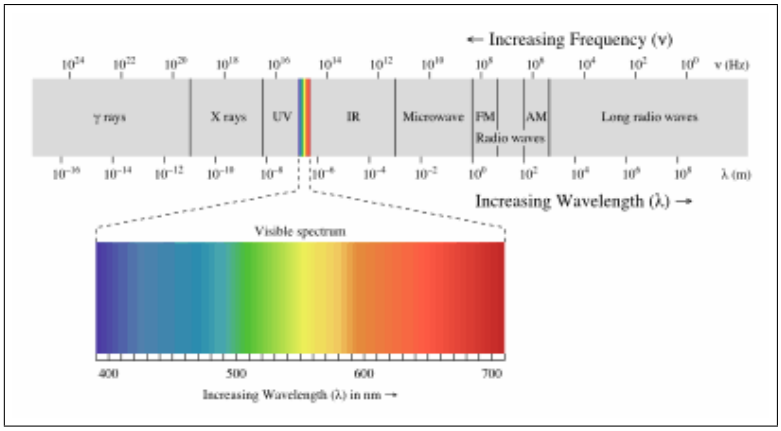


図 1: 電磁波の波長及び周波数とその種類 ([1] より)

1.4　光のドップラー効果と横ドップラー効果

音のドップラー効果は、救急車で例を挙げると、前方から近づいてくる時と、すれ違い後方へ遠ざかっていく時では、救急車の音が変化して聞こえる効果である。救急車と観測者の距離が相対的に近づくように運動すると、観測される波長が短くなり、音が高くなる。逆に、相対的に遠ざかるように運動すると波長が長くなり、音が低くなる。

光は光子であり、波の特性も持っている。そのため、光にもドップラー効果が起こり、光のドップラー効果では、波長が変化すると観測される色が変化する。光源と観測者が相対的に近づくと、波長が短くなり青方偏移が起こり、逆に遠ざかると波長が長くなり赤方偏移が起こる。図 2 は、光の波長と観測される色のサンプルである。波長が $400nm$ ぐらいでは紫色として観測され、波長が $700nm$ ぐらいでは赤色として観測される。この波長が $400nm$ よりさらに短くなっていくと、紫外線や X 線や γ 線になり人間の目には観測できなくなる。逆に波長が $700nm$ よりずっと長くなってしまうと、赤外線やマイクロ波になり、こちらも人間の目には観測できなくなる。

今回のシミュレーションでは、波長が $700nm$ で純粋な赤、$546nm$ で純粋な緑、$435nm$ で純粋な青として RGB で表現した。波長を RGB で表現する際に、この 3 種の波長を赤緑青とすることが多いようなので、本研究でもこの波長を採用した。人間がどの波長でどの色

として捉えるかは曖昧なため、明確にはどの波長がどの色かは決まっていない。同じように、人間が光として捉えられる波長の範囲を $380nm \sim 750nm$ として、紫を $405nm$、藍色を $420nm$、オレンジを $605nm$、シアンを $490nm$、黄色を $580nm$ とした。そして、上記で決めた色と色の間の波長差によって一定の割合で RGB の値を変化させた。これにより、$1nm$ 毎に色を変化させることができる。ただし、これは波長と色の関係を比較し決めた値であるため、より正確な値があればそちらを採用した方が良いだろう。なお、図 2 はこの値を元に作成した波長と観測される色のサンプルである。

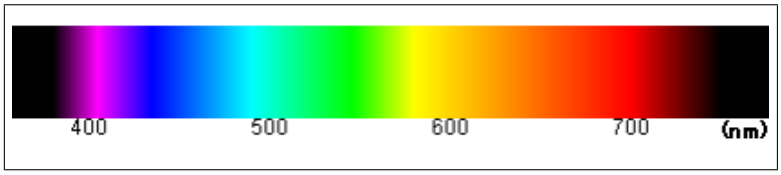


図 2: 光の波長と色の変化

よく救急車で音のドップラー効果を説明されているが、それはお互いの直線上で運動をしている場合である。しかし、実際のドップラー効果では、お互いの直線上で等速運動している場合はほとんどなく、音源と観測者の相対的な運動方向と、音の伝播方向には角度を持っている。図 3 の救急車は全て前方に運動し、観測者は静止している。この場合のドップラー効果は、観測者との直線上で近づいてくるため観測者への伝播方向と運動方向の角度 0 の場合か、遠ざかっていく π の救急車が一番ドップラー効果の影響が現れる。そこから、それぞれ $\frac{1}{2}\pi$ に近づくほどドップラー効果の影響が少なくなっていく。この角度によってドップラー効果の影響に差がでることを横ドップラー効果という。

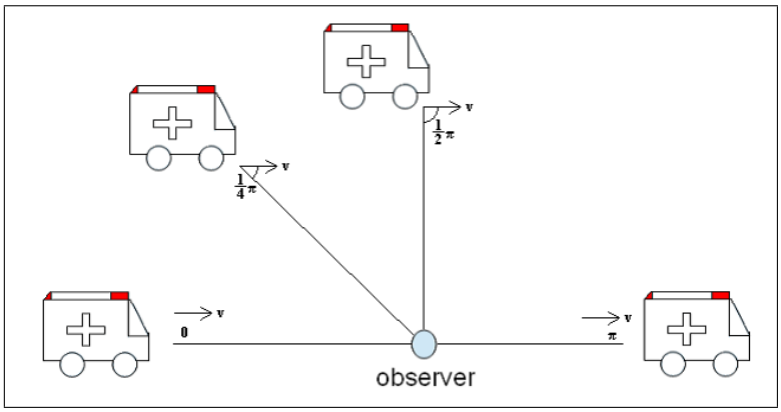


図 3: 相対的な運動方向に角度を持つドップラー効果

1.5 光行差

例えば、風のない状況で真上から雨が降っている時に、観測者が前方に向けて走ると、まるで雨が前方斜め上空から降ってくるように見えることになる。このような角度差が生じる現象を光行差という。特殊相対性理論における光行差は、観測者が高速運動をしている時に、運動方向に物体が集まってくるように見えることである。つまり、周りの物体が静止している空間を観測者が前方に向かって高速運動すると、視野が前方に集まってくる。逆に、遠ざかるように運動すると、観測者には実際の位置より後方、つまり運動方向へズレて見える。雨の速度と違い光の速度はとても大きいため、実生活で感じることは少ない。しかし、地球は太陽の周りを約 $30km/s$ で運動している影響で、星の見える位置にズレが生じることがある。これにより、光行差の現象が実際に観測されている。天体望遠鏡で星を観察する時には、ほんの少しのズレが生じると星が観測できなくなるためだ。

また、準光速の世界のシミュレーションを行う上で、この光行差を取り入れると前方に光が集まることになる。そのため、前方に光が集まったかどうかで明るさも変化する。

1.6 物体の形状変化

物体の形状変化と書いたが、大きくわけて 2 種類ある。特殊相対性理論によるローレンツ収縮と、見え方を考慮する上で物体が変形して見える物体の形状変化である。

まず、ローレンツ収縮は、光速に比べて無視できない速度で、細長い棒と観測者が相対的に運動していると、観測者には棒が運動方向に短くなって見えるという現象だ。ただし、これは細長い棒の話であり、実際の物体には奥行きがあるため、ただ短くなることはない。

しかし、これに見え方を考慮すると、奥行きのある物体では、見えていないはずの奥行きの面が見えてくることがあり、それは、まるで物体が回転しているかのように見えることがある。これをテレル回転と言う。図 4 は、このテレル回転の参考画像として、文献 [6, p.3] の一部を引用したものだ。図 4 左が静止している元々の画像だ。数字の 3 を観測者側にした賽子が一列に並んでいる。そして、図 4 右が、この賽子が数字の 3 方向、つまり観測者方面へ高速運動した時の画像である。一番右の賽子を見ると、手前側の 3 が見えなくなり、見えなかった奥側の 4 が見えている。また、奥側の賽子から近くの賽子になるにつれて、徐々に変化が出ており、まるで回転しているように見える。

これは、光速は有限であるため、高速運動すると物体が変形して見えるためだ。例えば、光が到達するのに 1 分かかる距離である 1 光分の長さの電車があったとして、その先頭付近にいる人が電車の最後部を観測していたとする。この電車の先頭と最後部が同時に動きだし、電車が観測者の後方へ運動を始めたとする。観測者には、電車の先頭部はすぐ近くのため、電車が動き出すのとほぼ同時に先頭部が動き出すのを観測する。しかし、最後部が動き出したという光は、観測者には 1 分後にならなければ届かない。そのため、観測者には、電車の先頭部は動き出しているのに、最後部は動き出していないように見え、電車が伸びていくように見える。電車が動き出しても、最後部の光が観測者に届くのに 1 分かかるため、観測者が見える景色と実際の電車ではズレが生じることになる。

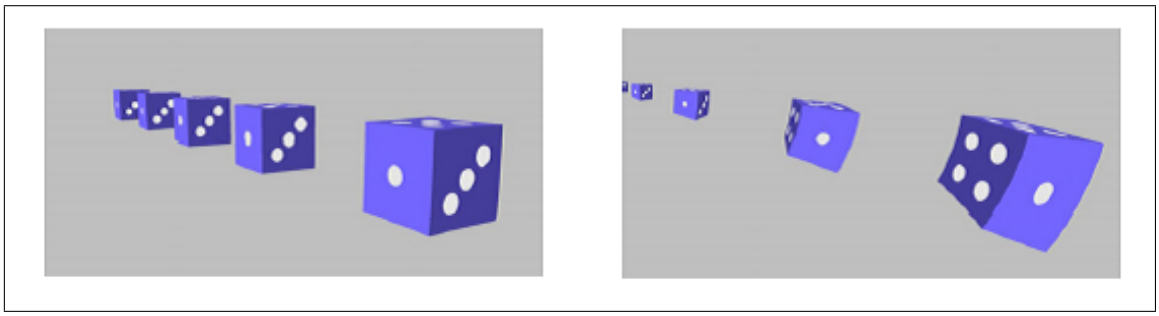


図 4: 高速運動すると回転して見える賽子 ([6, p.3] より)

このように、見え方も考慮しなければ実際の景色はシミュレーションできないため、準光速の世界を可視化する上で必須である。

1.7　本論文の構成

2 章では、波長と RGB と明るさについて説明し、それぞれの変換方法や計算方法を示す。3 章では、横ドップラー効果について説明し、この効果の簡単なシミュレーション結果を示す。4 章では、光行差について説明し、こちらでも簡単なシミュレーション結果を示す。5 章では、テレル回転を含む、物体の見え方についての説明する。また、シミュレーションをまずは 2 次元で行い、その後、3 次元へ発展させる。それ以降の章では、最終目標である準光速世界の擬似撮影を行うための方法や結果及び考察をしていく。

2　波長と RGB と明るさ

本研究では、横ドップラー効果や光行差で波長や明るさについて取り扱うため、この章で波長と RGB と明るさについて簡単に考察を行うとともに、本研究で行った計算方法をここで説明する。

2.1　RGB と HSV モデル

RGB とは、光の三原色と呼ばれる色の表現法の一種である。赤の R、緑の G、青の B を混ぜて幅広い色を再現するが、絵の具のように混ぜるほど黒色に近づくのではなく、光では RGB 全てを重ねると白色になる。赤と緑を重ねると黄色になり、赤と青を重ねると紫になり、緑と青を重ねるとシアンになる。RGB では、それぞれの色の強さを 0 から 255 までの 256 段階で表し、1677 万色以上の表現が可能だ。RGB の全てが 255 であれば白色になり、全て 0 だと黒色になるのだ。また、RGB の欠点として、細かい色の指定はできるが、光の明るさを直接指定し調整する事はできない。光行差では、明るさの変化も取り扱うため、直接明るさのみを変更できるように HSV モデルを採用した。

HSV モデルとは、色相の H、彩度の S、明度の V の 3 つの成分からなる色空間のことである。色相は、赤を 0 度とし、そこから時計周りに 120 度で緑、240 度で青と、360 度で赤や青や黄色といった色の種類を表す。彩度は、色の鮮やかさのことで、100% が全く濁っていない状態であり、0% が最も濁っている色を表す。明度は、明るさのことで 100% で最も明るく、0% で最も暗い色を表す。

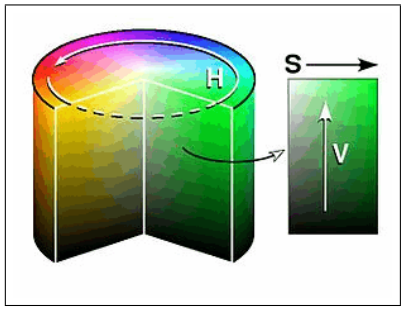


図 5: 円柱の HSV 色空間モデル ([2] より)

2.2 RGB から HSV への変換

ここではRGBからHSVへの変換方法を説明する。彩度と明度の説明を0%から100%としていたが、実際には彩度と明度は0から255までの256段階で使用する。RGBからHSVへの変換手順を以下に示す[2]。

RGBの3つの値のうち、最大値を$M = max(R, G, B)$とし、最小値を$m = min(R, G, B)$とすると明度Vは次のようになる。

$$V = M \tag{2.1}$$

ここで$V = 0$であれば、黒になるため、SとHは共に0となる。また、彩度Sは次式で求められる。

$$S = 255 \times \frac{M - m}{M} \tag{2.2}$$

残りは色相Hだが、HはRGBの最大値がどの色だったかで計算式が異なる。

$$H = 60 \times \frac{G - B}{M - m} \tag{2.3}$$

$$H = 60 \times \left\{2 + \frac{B - R}{M - m}\right\} \tag{2.4}$$

$$H = 60 \times \left\{4 + \frac{R - G}{M - m}\right\} \tag{2.5}$$

Rがmaxだった場合は式(2.3)となり、Gがmaxだった場合は式(2.4)となり、Bがmaxだった場合は式(2.5)となる。この明度Vを調整し、再びRGBへ変換することでRGBの明るさを変更することができる。

なお、この変換時に、Hは0から359までの値なので、負の値だった場合や360以上の場合は、範囲内に数値を収まるよう360を足すなどしておく必要がある。作成したプログラムでは、Hをint型とし以下のように値を調整した。

```
while(H < 0)
   H += 360;

H = H % 360;
```

2.3　HSV から RGB への変換

次に、HSV から RGB への変換手順を以下に示す。まず、彩度 S が 0 だった場合は、式 (2.2) で $M-m$ が同じ値だったということなので、RGB は全て同じ $R=G=B=V$ である。その他の場合は以下の通りである。

$$i = \lfloor \frac{H}{60} \rfloor \tag{2.6}$$

$$F = \frac{H}{60} - i \tag{2.7}$$

$$\begin{aligned} M &= V \times \left\{ 1 - \frac{S}{255} \right\} && (2.8) \\ N &= V \times \left\{ 1 - \frac{S}{255} \times F \right\} && (2.9) \\ K &= V \times \left\{ 1 - \frac{S}{255} \times (1-F) \right\} && (2.10) \end{aligned}$$

式 (2.6) は H を 60 で割った小数点以下を切り捨てた数値のことだ。この式 (2.6) の i の値で、R、G、B それぞれに何が当てはまるかが以下のように決まる。

$$\begin{cases} R=V, G=K, B=M & (i=0) \\ R=N, G=V, B=M & (i=1) \\ R=M, G=V, B=K & (i=2) \\ R=M, G=N, B=V & (i=3) \\ R=K, G=M, B=V & (i=4) \\ R=V, G=M, B=N & (i=5) \end{cases} \tag{2.11}$$

これで RGB と HSV を変換することができる。なお、プログラムでは RGB を使って色を登録するので、必ず最後は RGB へ変換する必要がある。

2.4　RGB から波長への変換

今度は RGB から波長への変換だが、これは本研究では行わないものとする。その理由は、波長から RGB への変換は可能であるが、RGB から波長への変換は困難だからだ。赤と緑の光が重なると黄色の光になるが、例え黄色の光が見えていたとしても、その光がどのような光の合成でできているかは解らない。赤と緑が重なっている場合や、黄色の波長の光が直接見えている可能性もある上に、多くの光が重なった結果で見えている可能性もある。

RGB から波長への変換ができないため、本研究では、どの座標からどんな波長が出ているかを調べ、表示の際に RGB へ変換する方式を取るものとする。

2.5 波長から RGB への変換

横ドップラー効果では波長の変化が起きるため、直接 RGB の変化を計算することはできない。そのため、波長を計算後に RGB へ変換する必要があるため、波長から RGB へ変換する方法を決めておく必要がある。

どの波長がどの色に対応しているかは、人間の認識によって決められているため、一部の波長以外は正確に決められていない。例を挙げると、赤が $700nm$、緑が $546nm$、青が $435nm$ と国際照明委員会 (CIE) で定められているが、紫などは $380nm \sim 420nm$ とだいたいの範囲が定められているだけである。本研究では、赤緑青以外の波長を黄色 $570nm$、シアン $500nm$、紫色 $405nm$ とし、人が光として認識できる範囲を $380nm \sim 750nm$ とした。

波長から RGB への変換は、波長の変化による RGB の変化に注目した。波長が $700nm$ であれば $R = 255$ で他は 0 だが、そこから波長が $570nm$ まで一定の割合で緑が上昇し、$570nm$ には RG 共に 255 になるようにした。波長から RGB の内の R を出すサンプルプログラムを以下に示す。可視可能範囲は $LIMIT\_PURPLE = 380$ から $LIMIT\_RED = 750$ で、$RED = 700$、$YELLOW = 570$、$GREEN = 546$、$BLUE = 436$、$PURPLE = 405$ である。なお、Wavelength は現在の波長である。

```
if(LIMIT_RED > Wavelength && Wavelength > RED){
reS = LIMIT_RED - RED;
R = 255.0 * ((LIMIT_RED - Wavelength)/reS);
}else{ if(RED >= Wavelength && Wavelength >= YELLOW){
R = 255;
}else{ if(YELLOW > Wavelength && Wavelength > GREEN){
reS = YELLOW - GREEN;
R = 255.0 - (255.0 * ((YELLOW - Wavelength)/reS));
}else{ if(BLUE > Wavelength && Wavelength >= PURPLE){
reS = BLUE - PURPLE;
R = 255.0 * ((BLUE - Wavelength)/reS);
}else{ if(PURPLE > Wavelength && Wavelength > LIMIT_PURPLE){
reS = PURPLE - LIMIT_PURPLE;
R = 255.0 - (255.0 * ((PURPLE - Wavelength)/reS));
}else{
R = 0;
}}}}}
```

他の GB も同じように、どこからどこの範囲で値が上昇又は減少しているかを調べ、一定の割合で増減するようにした。

3 横ドップラー効果

3.1 横ドップラー効果の式の導出

まずは、光源と観測者の直線上を相対的に運動している単純なドップラー効果について説明する。観測者が静止している座標系で考える。光源が v で運動し、固有時間 0、x_0 の位置で最初のパルスを発した。その後、時刻 τ_0、$x_0 + vt$ の位置で第二のパルスを発したとする。この v は、観測者に近づく場合は正であり、遠ざかる場合は負の値である。観測者は、最初のパルスを $t_1 = x_0/c$ で受信し、第二のパルスを $t_2 = t + (x_0 - vt)/c$ の時に受信する。ここで注意すべきなのは、観測者と光源が相対的に運動しているため、特殊相対性理論では時間のズレを考慮に入れる必要がある。そのため、ここでは n 番目のパルスを発した時の時刻を $t = (n-1)\tau_0\gamma$ とする。以上のことから、観測者での最初と第二のパルスを受信した時刻差、つまり観測者側での周期は次式になる。

$$\tau = t_2 - t_1 = \tau_0\gamma(1-\beta) = \tau_0 \left\{ \frac{1-\beta}{1+\beta} \right\}^{\frac{1}{2}} \tag{3.1}$$

ここで、β と γ は、それぞれ $\beta = \frac{v}{c}$、$\gamma = (1-\beta^2)^{-\frac{1}{2}}$ である。特殊相対性理論では、光源と観測者の相対速度のみに依存するため、光源が静止している座標系で考えても同じことが言える。

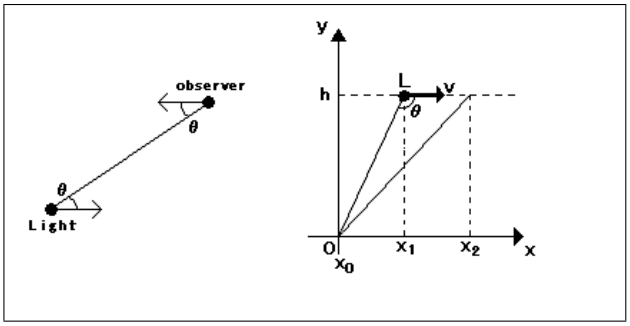


図 6: 光源と観測者の間に角度を持つ運動

この単純なドップラー効果は、光源と観測者がお互いの直線上を等速運動した場合であり、実際は図 6 左のように、観測者と光源の運動には角度を持っている。光源及び観測者が相対的に運動している方向を角度 0 とし、その運動方向から光源及び観測者までの角度を θ とする。光源の座標系での光の周期を τ_0 とし、光源 L は図 6 右のように時刻 $t_0 = 0$ で $x = x_0$ を通過し、観測者から高さ h を観測者の x 軸と平行に速度 v で等速運動しているとする。x_1 の位置で n 番目のパルスを発し、x_2 の位置で $n+1$ 番目のパルスを発した。よって、光源の周期は $x_2 - x_1$ で表すことができる。

$$x_2 - x_1 = v\tau_0\gamma \tag{3.2}$$

また、観測者の n 番目に受信したパルスと、$n+1$ 番目に受信したパルスの時刻 t_1 と t_2 は次式になる。

$$t_1 = \frac{x_1}{v} + \frac{r_1}{c} \tag{3.3}$$

$$t_2 = \frac{x_2}{v} + \frac{r_2}{c} \tag{3.4}$$

以上により、$t_2 - t_1$ で観測者側の周期 τ が求められる。

$$\begin{aligned} t_2 - t_1 &= \left\{\frac{x_2}{v} + \frac{r_2}{c}\right\} - \left\{\frac{x_1}{v} + \frac{r_1}{c}\right\} \\ &= \frac{(x_2 - x_1)}{v} + \frac{(r_2 - r_1)}{c} \\ &= \frac{c(v\tau_0\gamma)}{cv} + \frac{v\tau_0\gamma}{c}\frac{(r_2 - r_1)}{(x_2 - x_1)} \end{aligned} \tag{3.5}$$

ここで x_2 を x_1 に近似させる。

$$\begin{aligned} \lim_{x_2 \to x_1} \frac{(r_2 - r_1)}{(x_2 - x_1)} &= \frac{dr}{dx} \\ &= \frac{x}{r} \\ &= \cos(\pi - \theta) \\ &= -\cos\theta \end{aligned} \tag{3.6}$$

この式 (3.5) と (3.6) を使い、観測者側の周期 τ は次式になる。

$$\begin{aligned} \tau &= \tau_0\gamma\left\{1 + \frac{v}{c}\frac{(r_2 - r_1)}{(x_2 - x_1)}\right\} \\ &= \tau_0\gamma(1 - \beta\cos\theta) \\ &= \tau_0\frac{1 - \beta\cos\theta}{(1 - \beta^2)^{\frac{1}{2}}} \end{aligned} \tag{3.7}$$

また、波長 λ や周波数 f で表すと次式になる。

$$\lambda = \lambda_0\frac{1 - \beta\cos\theta}{(1 - \beta^2)^{\frac{1}{2}}} \tag{3.8}$$

$$f = f_0\frac{(1 - \beta^2)^{\frac{1}{2}}}{1 - \beta\cos\theta} \tag{3.9}$$

ここで、光源と観測者がお互いの直線上を近づく場合である $\theta = 0$ を式 (3.7) に代入すると、式 (3.1) に一致する。これらの式に光源と観測者の相対的な速度 β、角度 θ に加え、式 (3.8) には波長、式 (3.9) には周波数を与えることで、実際に観測される光の波長及び周波数が得られる。光を伝える媒質エーテルという存在が否定されているため、相対速度のみに依存する。また、相対的に運動がある場合、座標系の違いによる時刻の遅れなどの効果が現れている。

3.2　横ドップラー効果の式のグラフ化

式 (3.8) から角度が変化することで、観測される波長がどのように変化するかグラフ化した。元々の波長は全て 100nm の紫外線である。図 7 及び図 8 の横軸は角度であり、縦軸は波長で元の波長の何 % になったかを表している。図 7 では、$\beta = 0.2$ と $\beta = 0.4$ の変化が見えないため、図 8 は、$\beta = 0.2$ と $\beta = 0.5$ と $\beta = 0.8$ のみグラフ化したものだ。図 8 の $\beta = 0.2$ のように β が低ければ、あまり変化がない。それでも、正面は約 80% ほどに波長が減少しているため、ドップラー効果による色の変化は。十分に起きている。そして、観測者の真横では波長の変化はほとんど起きておらず、元の波長に近い状態だ。しかし、図 7 の $\beta = 0.99$ のように β が 1 に近づくと変化率が激しくなり、真横の $\theta = 90°$ の時も全く違う波長になっている。変化率が激しいということは、周り一面が可視光線の波長で満たされていたとしても、視野の一部しか可視光線が残らないということだ。そのため、赤外線や紫外線だったものが色として観測できる可能性はあるが、ほとんどの視野は光として捉えられない世界になる。

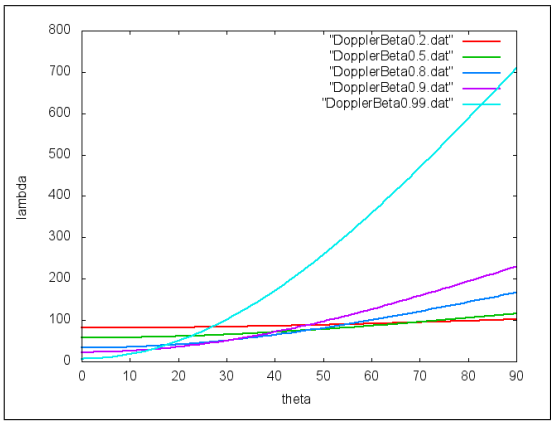


図 7: 高速ほど元々の波長が前方で観測される

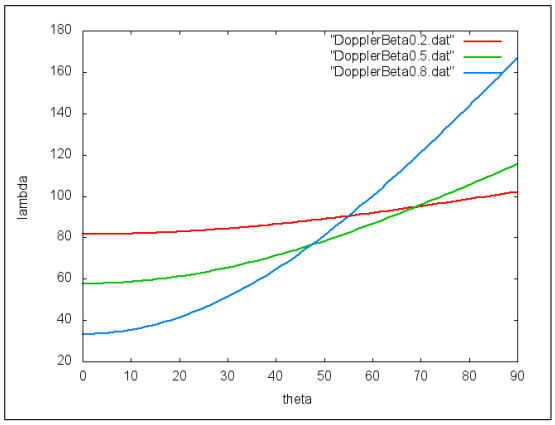


図 8: 図 7 の一部分を拡大したもの

3.3　横ドップラー効果の可視化

横ドップラー効果の式 (3.8) を用いてシミュレーションを行った。自分の周り全てが特定の波長の光一色の光源で覆われている世界で高速運動した時のシミュレーションである。図 9 で設定した波長は、画像通りで 580nm である。実際のシミュレーション画面に赤い丸数字と説明を追記したので、その番号順に画面の説明を行う。

図 9 の一番上でシミュレーションの設定を行う。　と記入されている場所が、シミュレーションを行う光源の波長だ。ここに波長を入力するか、　にある色のボタンをクリックして波長を設定する。そして、　の光速に対してどれだけの速度で運動するかを入力すると、その時に観測される景色が表示される。光源と観測者が相対的に静止していれば 0 であり、光

速で運動する場合は 1.0 であるが、光速には追いつけないという前提でシミュレーションを行っているため、速度の絶対値は 1.0 未満を入力する。この時、光源と観測者が近づくよう運動する時は負、遠ざかるように運動する時は正の値を入力する。

図 9 の　の上の虹帯は、光の波長と色のサンプルであり、下の虹帯はドップラー効果によって波長が変化した分だけズラしたものである。中央付近に入っている黒線は、設定した波長と変化した波長の色を示した線である。これは光源と観測者の直線上を運動した純粋なドップラー効果を表したもので、黄色の光源に向かって光速の 0.3 倍で運動すると藍色に変化して見えた、ということだ。

さて、図 9 の　　が横ドップラー効果を取り入れたシミュレーションになっている。　が光速の 0.3 倍で運動した時の運動方向の景色で、　が後方の景色であり、それぞれの円の中心が真正面で、円の外側が観測者の上下左右である。　の中心では、光源に向かって運動しているので、　と同じ藍色だが、角度が増すにつれて徐々にドップラー効果に差が出ている。観測者の真横である円の外側では、ドップラー効果がほぼないため元々の光源色のままである。　では逆に光源から遠ざかっているため、円の中心では人間には観測できない光である赤外線になっている。図 9 の　は　と　の景色を 1 つの円にまとめたものであり、円の中心が前方、円の外側が後方の色である。ある程度速度を上げると、図 9 の　のように同心円状の虹色に見えるスターボウという現象が起こった。

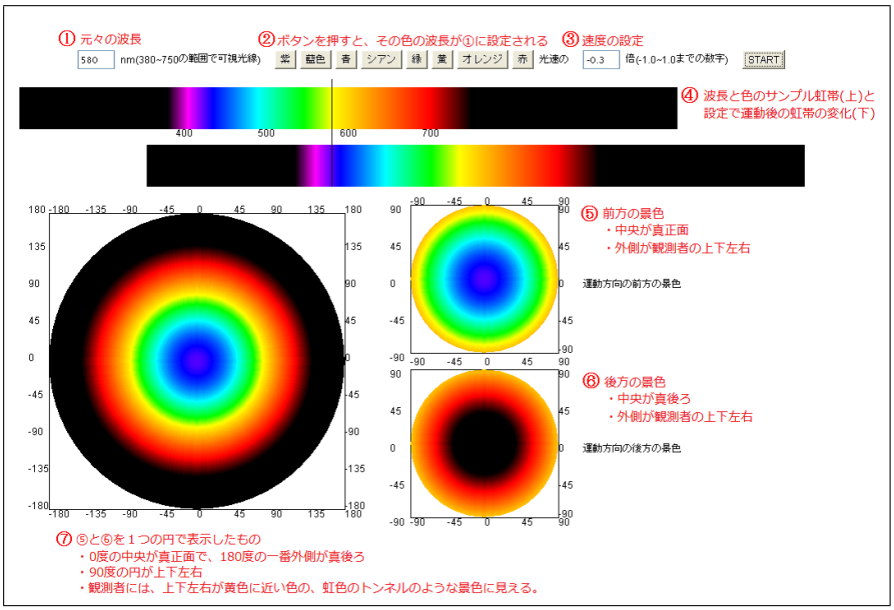


図 9: 横ドップラー効果のシミュレーション画面に赤文字で説明を加えたもの

4 光行差

4.1 光行差の式の導出

光行差とは、運動する物体から光源を見たとき、実際の角度方向とはズレが生じる現象のことである。光源から観測者へ光が届いた時、光源までの距離を x 座標と y 座標で考えると図 10 のように、x 座標の差は $v\cos\theta$、y 座標の差は $v\sin\theta$ になる。本来の光源までの角度が、運動することによりどのように変化するかを説明する。慣性系 S を x 軸方向に、速度 V で等速運動する観測者の慣性系 S' がある。

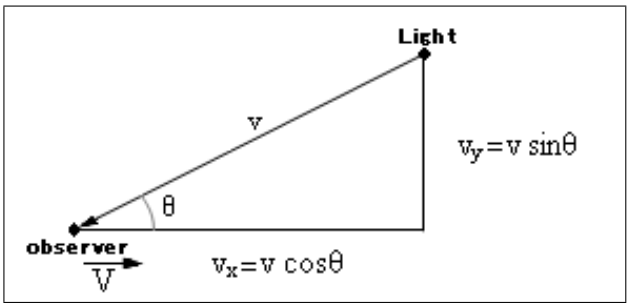


図 10: 光行差を考慮しない観測者から見る光源までの座標差

x 系と x' 系の式を、次式 (4.1) を用いて、ローレンツ逆変換を行う。

$$\begin{aligned} t &= \gamma(t' + \frac{V}{c^2}x') \\ x &= \gamma(x' + Vt') \end{aligned} \tag{4.1}$$

また、図 10 により次式も成り立ち、x' 系でも同じことが成り立つ。

$$\begin{aligned} v_x &= v\cos\theta \\ v_y &= v\sin\theta \end{aligned} \tag{4.2}$$

$$\begin{aligned} v'_x &= v'\cos\theta \\ v'_y &= v'\sin\theta \end{aligned} \tag{4.3}$$

これらの式から、観測者と光源の実際の角度と、光行差により見える角度の関係式を求める。

$$v_x = \frac{x}{t} = \frac{v'_x + V}{1 + \dfrac{Vv'_x}{c^2}} \tag{4.4}$$

$$v_y = \frac{y}{t} = \frac{v'_y}{\gamma(1 + \frac{Vv'_x}{c^2})} \tag{4.5}$$

この式 (4.4) と (4.5) に、式 (4.2) と (4.3) を用いることで、光行差の影響で見える星の角度 θ は次式にで求められる。また、光速不変の原理により、$v = c$、$v' = c$ である。

$$\begin{aligned} \tan\theta &= \frac{v\sin\theta}{v\cos\theta} \\ &= \frac{\sin\theta'\sqrt{1-\beta^2}}{\cos\theta' + \beta} \end{aligned} \tag{4.6}$$

式 (4.6) に、実際の観測者と光源の角度 θ' と相対的な速度 β を与えることにより、光行差の影響で観測者が見ることのできる光源の角度 θ が求められる。また、次式の通常のローレンツ変換を用いて、式 (4.6) までと同じ手順で、光行差のない実際の光源との角度を求める式にすることも可能である。

$$\begin{aligned} t' &= \gamma(t - \frac{V}{c^2}x) \\ x' &= \gamma(x - Vt) \end{aligned} \tag{4.7}$$

$$\tan\theta' = \frac{\sin\theta\sqrt{1-\beta^2}}{\cos\theta + \beta} \tag{4.8}$$

この式 (4.8) に、観測されている光源の角度 θ と相対的な速度 β を与えることで、実際に光源がある角度 θ' が得られる。

4.2　光行差の式のグラフ化

式 (4.6) を使い、元々見えていた光源の角度が、速度を上げることによってどのように変化するかグラフ化した。図 11 は、$\beta = 0$ の静止状態では $\theta = 30°$ と $\theta = 60°$ と $\theta = 90°$ に見えていた物体が、速度を上げるとそれぞれの角度がどれだけ変化したかを表したグラフである。$\theta = 90°$ は、観測者が運動方向を向いているとすると、真横や真上、真下となる角度である。それぞれの角度に見えていた光源は、速度を上げると共に角度が減少していき、速度を表す β が 1 の極限で 0 となる。つまり、視野は光速に限りなく近づくことで前方の一点に集まることが解る。

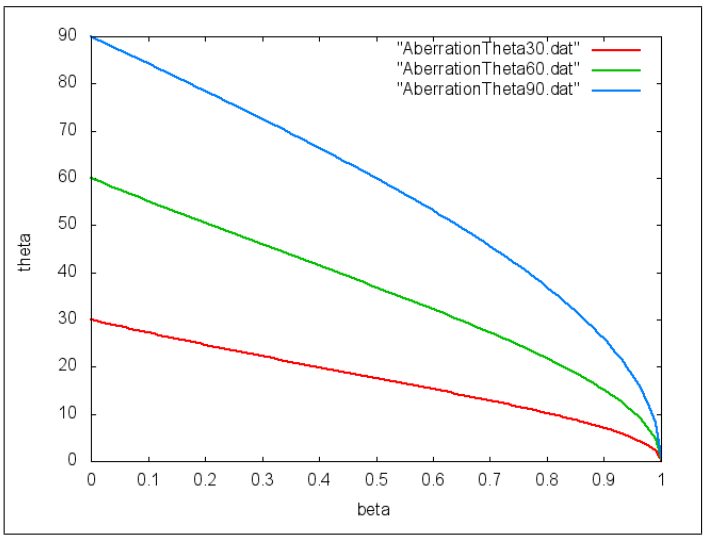


図 11: 光行差式 (4.6) をグラフ化した　光速へ近づくと視野は前方一点に集まる

4.3　光行差の可視化

光行差の式を用いて、再度図 9 の横ドップラー効果の時と同じ設定でシミュレーションを行った。図 12 がそのシミュレーション結果である。このシミュレーションは、図 9 で使った横ドップラー効果のシミュレーションに式 (4.6) を加え、光行差と明るさ変化のシミュレーションを追加したものだ。

の設定及びサンプルに関しては、横ドップラー効果の時と同じなので説明を省略する。光行差で視野が集まるということは、それだけ光が集まるということなので、その分明るくなる。画面の明るさを変化させることはできないため、RGB を白色に近づけることで明るさを表現した。

この明るさの調節は HSV モデルに変換し、光行差で視野が集まった角度分だけ明度 V を

上昇させた。

$$V = V\frac{\theta'^2\pi}{\theta^2\pi} \tag{4.9}$$

観測者が静止している場合を除き $\theta' > \theta$ である。そして、RGB に戻し、シミュレーション画面として出力した。

横ドップラー効果のシミュレーション結果を比較すると、前方に視野が集まった分明るくなっている。光行差で運動方向に視野が集まるため、　では光を観測できない範囲がより広くなっている。全体を表示した　では、図 9 と比べて光が中心に集まり、明るくなっているため、全体の色がぼやけて表示されている。

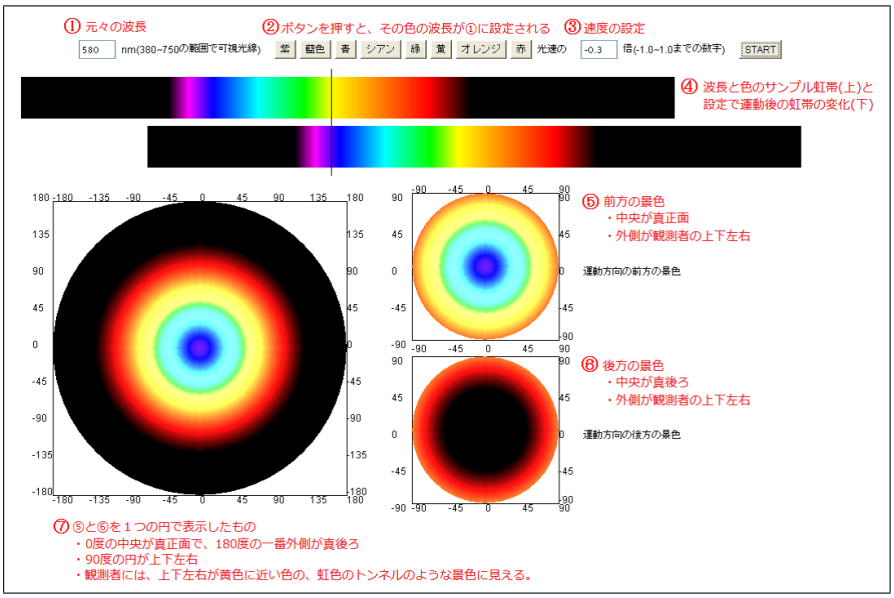


図 12: 光行差のシミュレーション画像に赤文字で説明を加えたもの

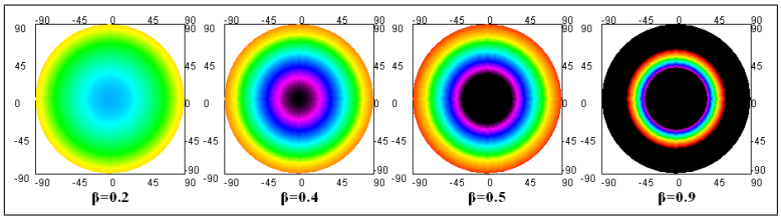


図 13: ドップラー効果のみを考えた前方の景色

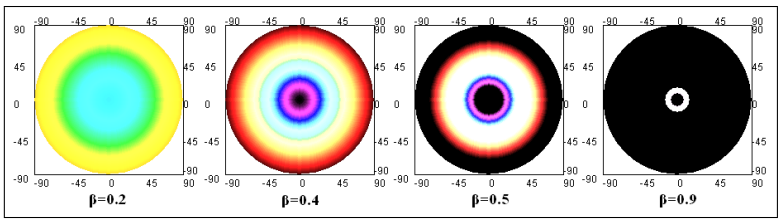


図 14: 図 13 に光行差及び明るさ変化を取り入れた前方の景色

図 13 と図 14 共に設定は、元々の色は黄色で、左から β が 0.2、0.4、0.5、0.9 の時、それぞれがどのように前方の景色が変化していくかを示した図である。横ドップラー効果では、速度が上がると紫外線と赤外線の範囲が徐々に広がっていった。それに比べ光行差では、紫外線の範囲は広くなっているが、その分視野も前方に集まっているため、同時に紫外線の範囲も狭くなっている。そのため、光行差のシミュレーションでは、紫外線の範囲は $\beta = 0.5$ ぐらいが最大の広さになり、そこから速度が上昇するにつれて、徐々に紫外線の範囲も $\beta = 0.9$ のように狭くなっていった。しかし、視野が前方に集まっているため、赤外線の範囲では光行差の方が広くなっている。

また、光行差の $\beta = 0.5$ では中心の紫と、外側の赤色は明るさ変化の影響が少ない。これは、光行差で角度の変化で光が集中していても、中心や外側では集まった光が紫外線や赤外線であれば、明るさの変化は起こらないためだ。

横ドップラー効果でスターボウが起こっていても、光行差では光が集まり明るさが変化するため、速度が上がると真っ白になってしまう。PC の画面で明るさを白色に近づけることで表現しているためで、簡単に説明すると、赤一色の光でも、光が非常に強ければ明るさを表現するために白一色で表示されるからだ。実際には、眩しいもののスターボウが観測できると思われるが、今後のシミュレーションで真っ白の世界になる可能性があるため、対策も考えなければならない。

5　見え方による物体の形状変化

5.1　テレル回転

夜空を見上げるとたくさんの星が輝いているが、その星の光は同時に発せられたものではない。光を同時に観測できても、数年前の光もあれば数百年前の光のものもあるからだ。準光速の世界のシミュレーションを正確に行うには、物体からの光が観測者に同時に届いたかどうかも正確に計算しなければならない。この時、星で例えたように、同時に観測できても、その光が発せられたのが同時とは限らない。ここからの話は光の速度が非常にゆっくりであったものとして考えると、イメージしやすいだろう。奥行きが見えている状態で静止している物体があるとする。観測者も静止しているならば、この物体は何も変わらない状態で観測できている。だが、物体の奥側と手前から発せられた光は同時に観測者には届いていない。つまり、図 15 左のように観測者が物体の奥側と手前側の光を同時に観測しているが、観測している奥側の光は、手前側の光が発せられたものより過去に発せられた光なのである。

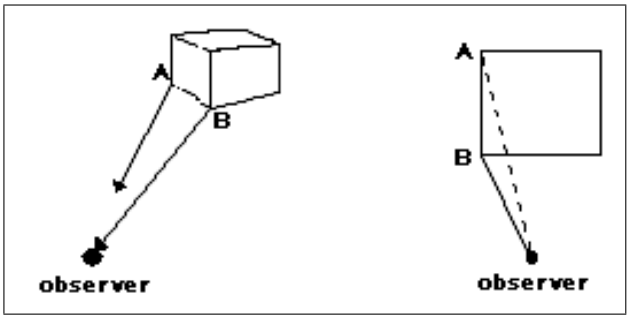


図 15: 奥行きがある物体の光

さて、奥行きがあると同時に発せられた光でも、観測者に届く時間にはズレが生じることを考えた上で、観測者が静止し物体が準光速で等速運動していた場合を考えてみる。物体が静止している時は、図 15 右のように、観測者と奥側の A の間に物体があるため、奥側 A の光が観測者に届くことはない。では、物体が準光速で運動している時、どのようなことが起こるのか。奥側の A から観測者に向けて発せられた光の速度は、物体の運動方向と垂直方向に分解することができる。そして、分解した物体の運動方向の速度を c_v とする。この c_v より物体の運動速度 v が上回っている時、A から観測者へ向かって発せられた光は物体に衝突し、観測者へ届かなくなることはない。このため、図 16 の (a) から (b) のように、徐々に発せられた光と物体の距離が広がりつつ、光は観測者へ向かう。そして、(c) からさらに次の瞬間の (d) では、ついに光と観測者の間に物体がなくなり、光が観測者へ届くことになる。

元々見えていた手前の面はローレンツ収縮で短くなっていくが、本来見えないはずの奥側の面が見えていくることにより、まるで物体が回転しているかのように見える。これをテレル回転という。James Terrel が高速で直線運動する物体が向きを変えて歪んで映ることを発見し、1959 年に論文として発表し [7]、回転しているように見えることからこう呼ばれて

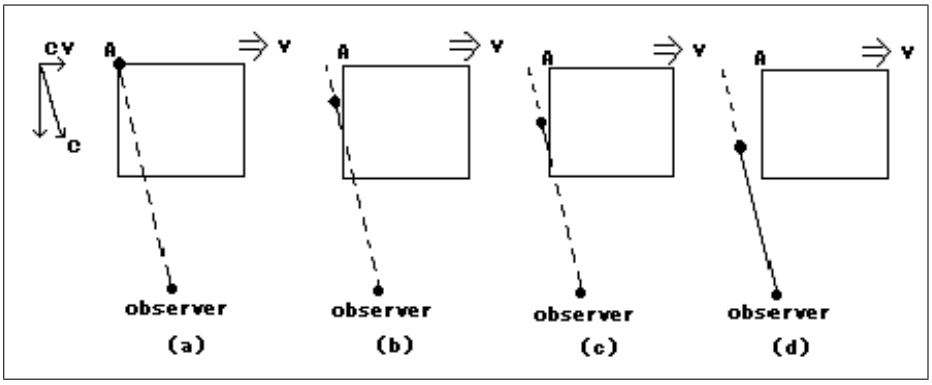


図 16: 物体の奥側の光が観測者側に逃げる

いる。

また、この回転して見えることと、回転しているように錯覚することの違いを図 17 で簡単に説明する。図 17 左がテレル回転をしていない静止した状態の物体である。そして、図 17 中央が静止した物体とローレンツ収縮をしただけの物体を重ね合わせたもので、図 17 右が静止した物体とローレンツ収縮しつつテレル回転した物体を予想して描き重ね合わせたものである。元の物体を黒色、ローレンツ収縮した物体を青色、テレル回転したものを赤色で描いた。重要なのは、テレル回転では、物体が運動してきた場所からしか光は発せられることはないことである。つまり、奥側の角が実際に回転してしまうと、物体がなかった場所から光が飛んできたことになってしまう。そのため、図 17 右で元の物体と比較すると、テレル回転で奥側の角が潰れたようになっている。また、手前の面だけを見るとローレンツ収縮をしているだけで、回転していない。つまり、テレル回転とは、ローレンツ収縮で短くなる面と、見えていた面が見えなくなったり、見えていなかった面が見えることによって回転しているように錯覚することである。

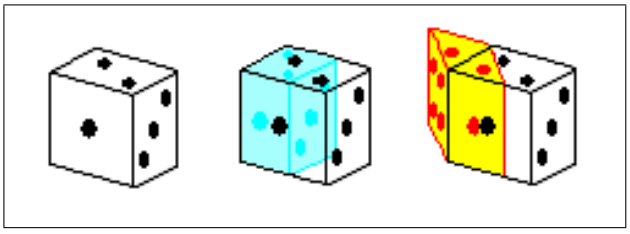


図 17: 元の物体とローレンツ収縮とテレル回転の違い

5.2　テレル回転による物体の変形

まるで回転しているように見えるテレル回転の変形は、どのような要素によってその変化が激しくなるのか考えてみる。そのため、これから考えることはテレル回転が起こる条件を満たしているものとする。

5.2.1　テレル回転と速度

1つ目の要素は、物体と観測者の相対速度である。今回は、観測者が静止し、物体が高速運動している座標系で考える。図 18 は、物体の速度のみ違うだけで、他の条件は同じとする。図の点 B の位置から発せられた光が、観測者に届いた時と同時に届く物体の奥側の A からの光は、図のように A_1 及び A_2 の時に発せられた光であるとする。この条件で、図の位置で共に O に向けて発せられた A の光について考えてみる。ここで、図 18 左の物体の速度 v_1 は、テレル回転が起こりうるぎりぎりの速度である $c_v \leq v_1$ とする。また、図 18 右はの物体の速度 v_2 は、テレル回転が十分起こりうる速度である $c_v < v_2$ とする。$v_1 < v_2$ のため、A_1 から光が発せられてから、点 B から光が発せられるまでの時間にも差が出る。そのため、v_1 の速度では、A_1 と B の光が同時に O に届くが、v_2 の方は速度が速いため、図 18 右の A_1 から光が発せられてから B の光が発せられるまでの時間も短い。よって、図 18 右では A_1 と B の光は同時には届かず、B が O に届いても A_1 の光はまだ O には届いていない。つまり、速度以外の条件が同じ時は B の光と同時に O へ届く A 側の光は、速度が速ければ速いほど、位置的により遠くの場所から発せられた光が B と同時に届くことになる。これにより、図 18 の左と右を比べたように、通常見えないはずの奥行きの面は、速度が速いほど大きく見えるようになる。

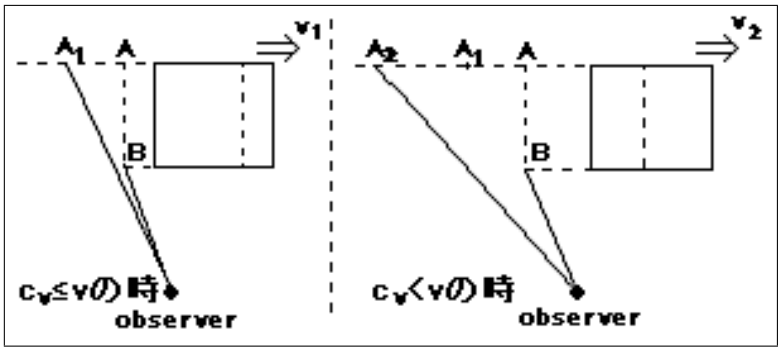


図 18: 速度 v が c_v を辛うじて上回った場合 (左) とある程度上回っている場合 (右)

5.2.2　テレル回転と奥行き

2 つ目は、奥行きの長さである。奥行きが長ければ長いほど、テレル回転した時に見える奥行きの面も大きくなるのかということである。もし、奥行きの面に模様があるならば、奥行きの長さと無関係に面が見えるならばこの模様も崩れて見える可能性がある。図 19 のように、一番奥の A と奥行きの中間の C が同時に観測者に届くかを考えてみる。同時に届くということは、A は A_1、C は A_2 の点から光が発せられることになる。さらに、A_1 と A_2 の光が重ならなければ、観測者に同時に届くことはない。しかし、このためには、c_v と v が $c_v = v$ の関係が成り立たなければならない。だが、このテレル回転には、$c_v < v$ が条件である。よって、A と C の光が重なり、観測者へ同時に届くことはない。C の光は A_3 辺りから発せられたものが A_1 の光と同時に届くの考えられる。物体の側面に模様があれば、物体の手前と奥側では比率が変化する可能性がまだ捨てきれないが、十分に側面が見えるほどテレル回転が起きていれば模様も認識できるだろうということがわかる。

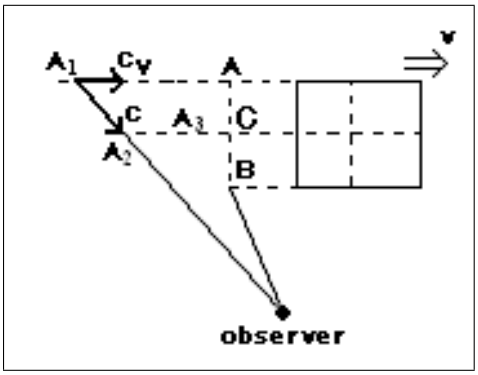


図 19: 奥行きの長さが物体と無関係であれば A_1 と A_2 の光は同時に届く

5.2.3　テレル回転と運動方向の位置

3 つ目は、物体と観測者の相対的な位置である。テレル回転が現れるのに十分な速度で物体と観測者が相対的に運動している場合で、物体と観測者の位置とテレル回転について考えてみる。

(a) では、距離が離れているため、奥行きの光の速度を分解した時に、運動方向の速度が光速 c に非常に近い状態になる。そのため、(a) では、物体は非常に光速に近い速度で運動していなければ、A_1 の光は観測者側へ逃げることはなく、テレル回転も起きない。また、光速は超えられないことを前提に考えているため、物体の速度も有限である。そのため、運動方向の光の速度を大きく上回ることはできない。よって、(a) では、テレル回転が起きるには非常に光速に近い速度が必要であり、テレル回転も大きく現れることはない。次に (a) より近づいてきた (b) でも、ある程度光速に近づかなければテレル回転は起きない。しかし、(a) よりも運動方向への A_2 からの光の速度が遅いため、(a) よりも低速であってもテレル回転は現れる。また、光速に非常に近づくことができれば、それなりにテレル回転も現れるだろう。最後の (c) では、観測者の位置に近いため、A_3 からの観測者への光の速度を分解すると、物体の運動方向への速度は光速 c より明らかに低速であることがわかる。そのため、物体は光速 c よりある程度遅くてもテレル回転は起こり、光速 c に非常に近づくことができれば、大きくテレル回転の効果が現れる。

光が観測者へ向かう速度は光速の c なので、物体が近づくにつれて速度分解すると運動方向への速度 c_v が遅くなっていくのがわかる。つまり、観測者へ近づくほど小さな速度でテレル回転が起こり、同じ速度でも観測者へ近づくほどテレル回転も大きく現れることがわかる。また、光の速度分解をすると、(a) から (c) へ近づくほど、物体の運動方向と光が観測者へ向かう間の角度が θ_1 から θ_3 へ大きくなっていくのがわかる

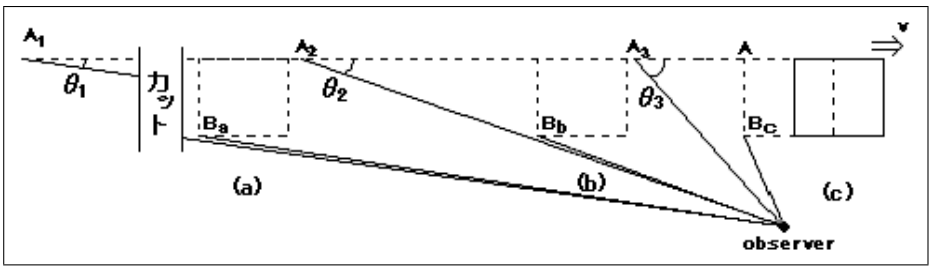


図 20: 物体と観測者が近づくほどテレル回転が起きやすい

5.2.4　テレル回転と観測者と物体の距離

次に、物体の運動方向上で観測者に近づいてきた場合ではなく、観測者が遠ざかって観測した場合について考える。そのため、速度、物体の大きさ、基準となる手前側の B 点から光が発せられたとする座標も同じで考える。物体の運動方向上から垂直に、観測者が遠ざかった場所で観測していくと、テレル回転はどのように変化するのだろうか。

これまでのことから、物体から観測者が離れると B 点から光が届く時間が伸びるため、当然 A 側の光は、観測者が物体からより離れた場所にいれば、過去に発せられた光になっていく。観測者が離れた場所にいると B 点から発せられた光と比べて、より過去の光を観測するということは、物体から離れるとテレル回転の効果が大きく現れるということである。

5.2.5　テレル回転を簡潔化した式

特殊相対性理論のローレンツ収縮の説明を行う時、稀にテレル回転も簡単ではあるが一緒に説明している時がある [3, p.15]。そこでは、直方体の物体が実際の物体の運動方向より θ 回転して見えるということを説明し、式 (5.1) のように表している。奥行きの長さが a の奥側の光が、物体の手前側に届くまでの時間は $\dfrac{a}{c}$ である。その間も物体は速度 v で運動しているため、$\dfrac{v}{c}a$ だけ動いている。また、物体の横幅 b はローレンツ収縮により、$\beta = \dfrac{v}{c}$ として $b\sqrt{1-\beta^2}$ に見える。このため、この直方体は角度 θ で回転してるのと同じだというのだ。この θ は、次式を満たす角度である。

$$\tan\theta = \frac{\beta}{\sqrt{1-\beta^2}} \tag{5.1}$$

しかし、これはテレル回転の回転率を簡単に説明しているだけであるため、式 (5.1) は実際のテレル回転の式ではない。

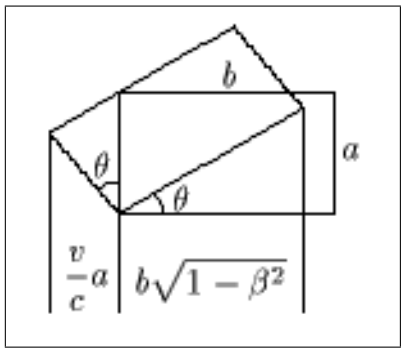


図 21: この図のように [3, p.15] による簡略化された式 (5.1) は正確ではない

5.3　見え方による物体の変形

ここまで物体の回転について考察してきたが、テレル回転が起こる原因はどの光が同時に届くかを考慮すると現れる現象であることが解った。では、ローレンツ収縮とテレル回転以外にはどのような物体の変形が起こるのか。それは、観測者と物体が相対的に近づくと、物体がローレンツ収縮以上に伸びて見えて、逆に遠ざかるとローレンツ収縮以上に縮んで見えるということが起きる。これもどこの光が同時に届くかを考えると解ることだが、光が届くのに 20 分かかる長さの 20 光分の列車を使って上手に説明されていたので要約しつつ引用する [5, p.100-105]。

> 同時に動いた車両が、本当に同時に動いて見えるかといわれれば、特殊な場合を除いて NO である。先頭車両付近のホームにいる観測者は、先頭車両が動き出した 20 分後に最後部車両が動き出すのを観測することになる。
>
> 観測者には、先頭車両が動き出してから 20 分間は最後部車両は動かず、先頭車両が進んだ分だけ伸びて観測される。逆に、最後部車両付近に観測者がいた場合は、先頭車両は 20 分間動いていないように観測されるため、最後部車両が進んだ分だけ縮んで観測される。これは、見え方の問題であって、ローレンツ収縮とは全く違う話である。

観測者が先頭車両付近にいて、この列車が光速に近い速度で運動していたとする。列車が動き出してから 20 分間は、最後部車両が動いていないように見えているため、最後部車両が動き出したという光が観測者に届く直前には、約 20 分間をほぼ光速で列車が運動しているのだから、まるで列車が約 2 倍に伸びたように想像するかもしれない。しかし、観測者から遠ざかる物体は縮んで見える。観測者の横をほぼ光速で運動する最後部車両が通過した瞬間に先頭車両を観測するとどのように見えるか。先頭車両は、観測者から 20 光分離れた場所にあるため、この光が観測者に届くには 20 分かかる。そのため、この瞬間に観測者が見る先頭車両の光はもっと過去の光になる。この列車はほぼ光速で運動しているため、観測者が先頭車両を観測した瞬間に届く光は、20 光分の半分の 10 光分離れた場所の光が観測者には見えることになる。観測者と物体の相対速度にもよるが、近づく場合と遠ざかる場合で、それぞれ物体の長さは 2 倍から $\frac{1}{2}$ 倍も変形して見えることになる。

このような物体が変形して見える現象は、光速が有限であるために生じる見かけの変化である。この見かけの変化の中で、見えなかった奥側が見えるほど物体の見かけが変化したもの、それがテレル回転である。

5.4　2 次元の物体変形の式の導出

テレル回転を含む物体の変形をシミュレーションするために、これまでのことを踏まえて、まずは簡単な二次元で、どのように物体の変形が起こるかを表す式を導出する。これまでのことから、物体の変形は、ある光と同時に届くのは、その光より過去や未来に発せられた光であるために起こる現象であることがわかった。このため、物体の A 点が変形して見えても、その A 点の光は当然 A 点が運動してきた直線上から発せられたものである。いつ発せられた光か、その時間差が計算できれば、この物体の変形をシミュレーションすることができる。式の導出の前に理解すべき一番重要なことは、この A 点から届く光がいつ発せられたものであれ、求める光が発せられた座標は、運動方向上にズレた点であるということだ。

図 22 のように、物体の観測者側の z 軸上に C 点があり、奥側に A 点がある。そして、この物体は z 軸上を z 軸正方向へ速度 v で等速運動し、観測者は z 軸上の原点から L 離れた場所から物体を観測しているものとする。式の導出には、図 22 の物体の手前側 C 点からの光と同時に届く奥側 A 点の光の発生場所を求めるには、C 点と観測者の座標差がまず必要になる。物体の運動方向を z 軸方向とした時の、観測者と C 点の z 座標差である z_C と、C 点の運動している直線上までの距離 L で表すことができる。他には、物体の奥行き a、運動方向にローレンツ収縮した物体の横幅 b' と、C 点から光が観測者に届くまでの時間 T、さらに物体の速度 v、光速 c が必要なことがわかっている。

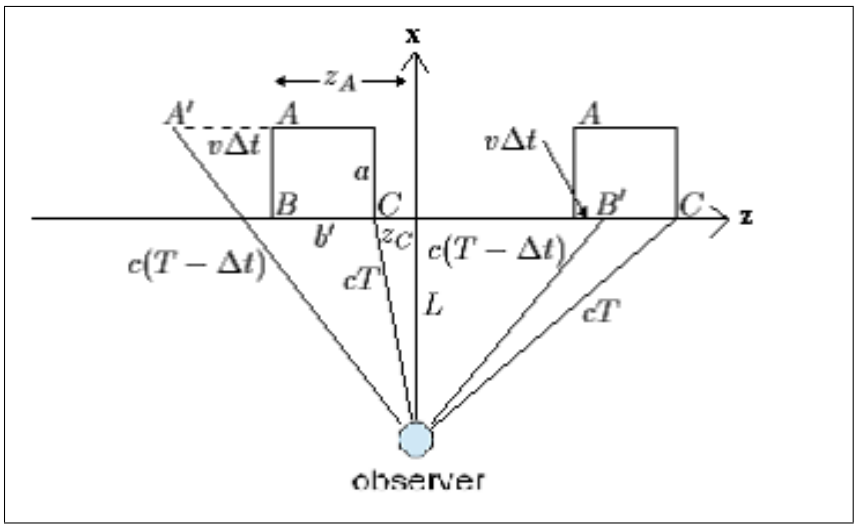


図 22: 基準点より過去の光 A' と未来の光 B'

点 C の z 座標は z_C で、点 A の z 座標は $z_A = z_C + b'$ である。そして、求める A' の z 座標は、$z'_A = z_C + b' + v\Delta t$ である。C 点からの光が観測者に届くまでの時間を T とし、A' 点から光が発せられてから、C 点で光が発せられるまでの時間差を Δt とする。この A' の時間差 Δt は、基準である C 点から光が発せられるより過去の場合の Δt の値は負である。しかし、B' のように基準の点から光が発せられた後に求める点の光が発せられるような未

来の光である場合は、時間差 Δt の値が正になる。また、光なので速度は c である。

まずは、図 22 より以下の式がわかる。

$$\begin{aligned} c(T-\Delta t) &= \sqrt{(L+a)^2+z_A'^2} \\ cT &= \sqrt{L^2+z_C^2} \\ z_A' &= z_C+b'+v\Delta t \end{aligned} \tag{5.2}$$

これらの式から、z_A' の値を求める。

$$\begin{aligned} cT-c\Delta t &= \sqrt{(L+a)^2+z_A'^2} \\ \sqrt{L^2+z_C^2}-c\Delta t &= \sqrt{(L+a)^2+(z_C+b'+v\Delta t)^2} \\ (\sqrt{L^2+z_C^2}-c\Delta t)^2 &= (L+a)^2+(z_C+b'+v\Delta t)^2 \end{aligned} \tag{5.3}$$

式が長くなるため、$(L^2+z_C^2)-((L+a)^2+(z_C+b')^2)=X$、$\sqrt{L^2+z_C^2}+\beta(z_C+b')=Y$ と置いた。また、$\beta=\dfrac{v}{c}$ である。二次方程式の根の公式を用いて、式 (5.3) から $v\Delta t$ の値を求める。

$$\begin{aligned} 0 &= (c^2-t^2)\Delta t^2-(2c\sqrt{L^2+z_C^2}+2v(z_C+b'))\Delta t+X \\ \Delta t &= \frac{2c\sqrt{L^2+z_C^2}+2v(z_C+b')\pm\sqrt{(2c\sqrt{L^2+z_C^2}+2v(z_C+b'))^2-4(c^2-v^2)X}}{2(c^2-v^2)} \\ \Delta t &= \frac{Y\pm\sqrt{Y^2-(1-\beta^2)X}}{c(1-\beta^2)} \\ v\Delta t &= \frac{\beta}{(1-\beta^2)}\left\{Y\pm\sqrt{Y^2-(1-\beta^2)X}\right\} \end{aligned} \tag{5.4}$$

この式 (5.4) の $v\Delta t$ を用いると、求めるべき座標 $z_A'=z_C+b'+v\Delta t$ が求められる。繰り返すが、この物体の変形で座標が変わるのは運動方向の座標の z のみであり、x 座標は変化しない。また A 点は、基準の C 点より z 軸の負方向のため、b' は負の値である。

5.5　2 次元物体変形シミュレーション

2 次元での式でシミュレーションを行い、今までの考察通りの動きをするかを確かめる。設定された辺の長さを持つ立方体が画面左から右方向へ高速運動した時の、観測者が見ている物体の変形を真上から見た図を用いて議論する。立方体の 1 辺の長さ、立方体の運動している直線上から観測者までの距離、光速の何倍の速度で運動するかの 3 つを設定する。黒線が元々の物体、青色が運動方向にローレンツ収縮しただけの物体で、赤色がシミュレーション結果の形状変化が起きた後の物体を表している。

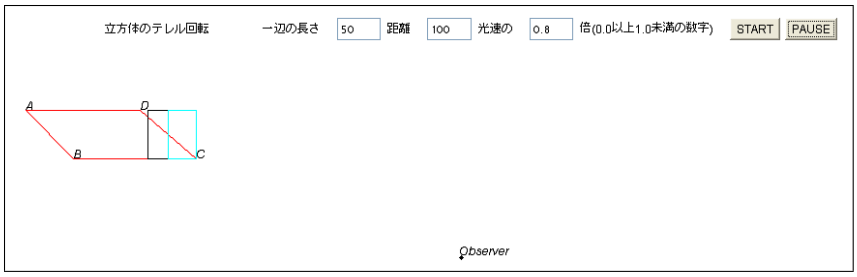


図 23: 物体と観測者に距離がある場合に近づくと物体は伸びて見え変形も大きい

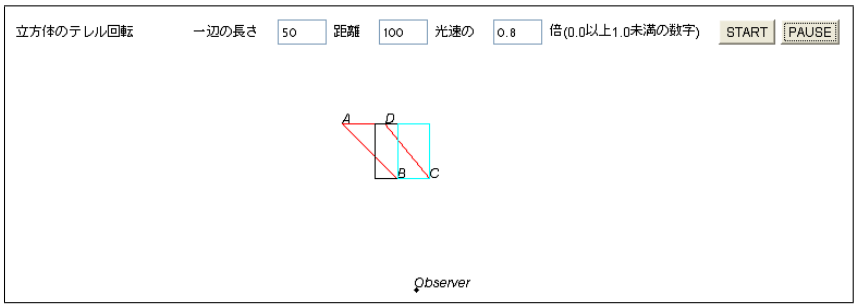


図 24: 図 23 からさらに近づき、すれ違う瞬間には元の大きさより縮む

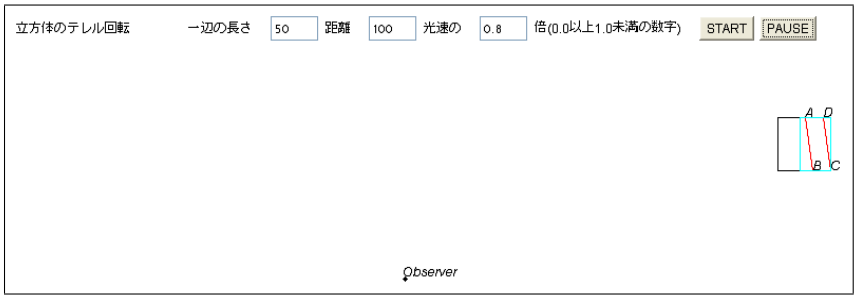


図 25: 物体と観測者が相対的に離れていくと、物体はさらに縮み物体の変形も小さい

図 23 から図 25 までは、画像に表示されている一辺 50、距離 100、速さ 0.8 という設定でシミュレーションを行っている。物体が観測者へ近づいている時は、図 23 のように、物体が伸びて、テレル回転も大きく起きている。だが、物体が観測者の正面へ近づくにつれて物体の横幅 BC が元の大きさに戻り、図 24 ではローレンツ収縮とほぼ同じ大きさまで縮んだ。しかし、ここでは物体の変形は大きいままである。物体が観測者から遠ざかっていく図 25 では、物体がローレンツ収縮以上に縮み、テレル回転も小さくなっている。物体と観測者がすれ違う直前までは、物体の変形は大きかったが、すれ違った直後から物体の変形は徐々に小さくなり図 25 のようになった。

また、辺の長さと距離を同じ設定のままで、速度を $0.4c$ に落としてシミュレーションさせると、物体の変形も少なくなった。

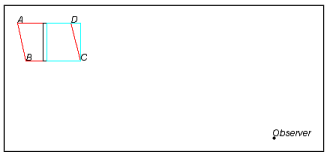


図 26: $\beta = 0.4$ で接近

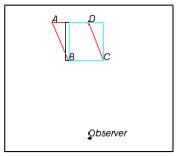


図 27: $\beta = 0.4$ ですれ違う

図 28: $\beta = 0.4$ で遠ざかる

これらのシミュレーションは、観測者が静止し物体が β に設定された速度で運動していた。次に観測者を運動させても同じように変化するかシミュレーションを行った。物体の一辺、距離、速度の設定は全て図 23 から図 25 までと同じである。ただし、先ほどのシミュレーションと同じ順序で、物体と観測者が接近し、その後離れるようにするため、運動する向きは逆にした。図 23 から図 28 では、物体が左から右へ運動しているが、今回は観測者が右から左へ運動する。

その結果、物体が運動しても観測者が運動しても同じように物体が変形した。物体と観測者のどちらが運動しても結果は変わらず、相対的な運動で見え方が変わることが解った。

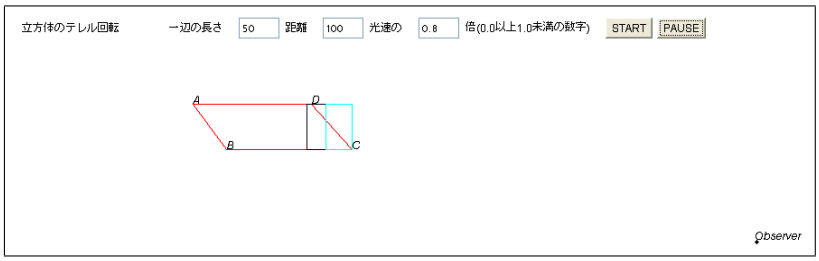


図 29: 観測者が近づいていく物体との距離がある時は物体の変形も大きい

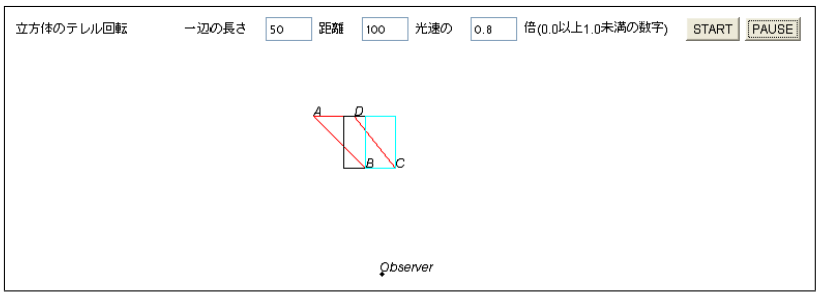


図 30: 観測者がすれ違う瞬間、図 24 と同じでように縮んだ

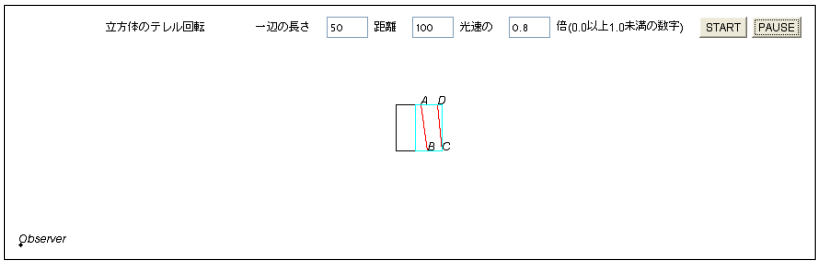


図 31: 観測者が遠ざかる場合も図 25 と同じく物体は縮み変形も小さくなった

また、今回作成した2次元の式を使ったプログラムで、物体の線 AB が元の線と比べて、変形した角度を調べ、Terrell が書いた文献にあるテレル回転の回転率と比較してみる [7, p.3]。

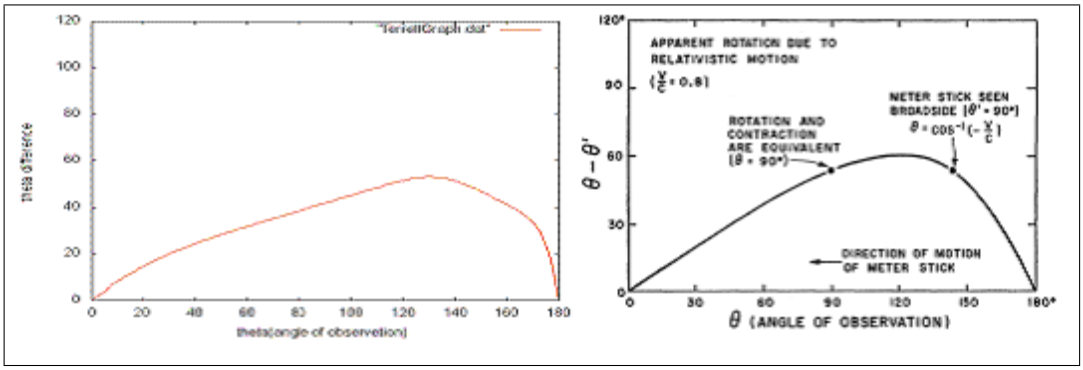


図 32: テレル回転率の比較

図 32 左が 2 次元の物体変形の式から出した角度のグラフで、図 32 右が Terrell の文献から引用したテレル回転の角度のグラフで、共に速度は $0.8c$ である。横軸は観測者と物体までの角度で、観測者から遠ざかるように運動する時の角度が 0 から $\frac{1}{2}\pi$ である。逆に観測者に近づくように運動する時の角度が $\frac{1}{2}\pi$ から π である。縦軸が元の物体と比べて線 AB が傾いて見えたかを表している。Terrell のグラフでは最大 60° だが、自分で作成したプログラムでは最大 54° であった。しかし、グラフ全体を見ると、同じような特徴のグラフになっていることがわかる。そのため、今回導出した式は、十分信用できるものである。

5.6　簡潔化した式の考察

5.2.5 で説明した式 (5.1) では、まるで速度と物体の形だけで回転率が決まるような式になっていた。しかし、観測者と物体の距離の変化でも回転率は変化する。また、物体には高さがあるため、その高さによっても回転率は変化するはずである。式 (5.1) は、物体の高さが 0 に近く、物体の運動する直線上と観測者の距離が 0 であり、物体が観測者とすれ違う瞬間の、だいたいの回転率ということになる。もし、これらの限られた条件で、観測者の目の前を図 24 や図 27 のように点 B が通過したとする。

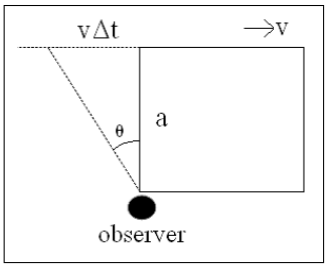


図 33: 式 (5.4) を用いた場合 a と $v\Delta t$ でできる角度が回転率になる

この時の条件で、式 (5.4) を使い、回転率を計算する。計算を単純化させるために、物体の高さと観測者と物体の高低差に加え、物体までの距離も 0 とする。式 (5.4) に、$L = 0$、$y = 0$、$z_C = 0$、$a = a$、$h = 0$、$b' = 0$ を代入すると、$X = -a^2$ と $Y = 0$ になるため、$v\Delta t$ と回転率は次式になる。

$$\begin{aligned} v\Delta t &= -a\beta \\ \tan\theta &= \beta \end{aligned} \tag{5.5}$$

$v\Delta t$ は、座標計算のため負の値になるが、回転率のみ調べるため $v\Delta t$ の絶対値を用いて $\tan\theta$ を計算した。計算した結果は、式 (5.1) の $\sqrt{1-\beta^2}$ 倍したものになった。ここまで条件を限定したが式 (5.1) には一致したなかったが、距離が 0 で運動する観測者とすれ違う瞬間では、回転率が $\tan\theta = \beta$ と速度のみに比例する値になることが解った。

5.7　3 次元の物体変形の式の導出

同様に 3 次元での式を導出する。条件は同じで、物体の運動方向は z 軸と平行である。

3 次元になっても、考え方は 2 次元と同じであり、物体の変形による座標の変化は運動方向の z 座標のみである。2 次元の時との違いは、3 次元であるため、高さの y 座標の差、基準から求める点までの高さの差 h があるため、2 次元の式とは少し違ってくる。

$$\begin{aligned} c(T-\Delta t) &= \sqrt{(L+a)^2+(y+h)^2+z_A'^2} \\ cT &= \sqrt{L^2+y^2+z_C^2} \\ z_A' &= z_C+b'+v\Delta t \end{aligned} \tag{5.6}$$

式 (5.6) から 2 次元の時と同じように $v\Delta t$ を求める。

$$\begin{aligned} cT-c\Delta t &= \sqrt{(L+a)^2+(y+h)^2+z_A'^2} \\ \sqrt{L^2+y^2+z_C^2}-c\Delta t &= \sqrt{(L+a)^2+(y+h)^2+(z_C+b'+v\Delta t)^2} \\ (\sqrt{L^2+y^2+z_C^2}-c\Delta t)^2 &= (L+a)^2+(y+h)^2+(z_C+b'+v\Delta t)^2 \end{aligned} \tag{5.7}$$

式が長くなるため、$(L^2+y^2+z_C^2)-((L+a)^2+(y+h)^2+(z_C+b')^2)=X$、$\sqrt{L^2+y^2+z_C^2}+\beta(z_C+b')=Y$ と置いた。また、$\beta=\dfrac{v}{c}$ である。

$$\begin{aligned} \Delta t &= \frac{2c\sqrt{L^2+y^2+z_C^2}+2v(z_C+b')\pm\sqrt{(2c\sqrt{L^2+y^2+z_C^2}+2v(z_C+b'))^2-4(c^2-v^2)X}}{2(c^2-v^2)} \\ v\Delta t &= \frac{\beta}{(1-\beta^2)}\left\{Y\pm\sqrt{Y^2-(1-\beta^2)X}\right\} \end{aligned} \tag{5.8}$$

この式 (5.8) の $v\Delta t$ を用いると、2 次元と同じように求めるべき座標 $z_A'=z_C+b'+v\Delta t$ が求められる。また、ここで高さである y, h に 0 を代入すると、30 ページにある 2 次元の式 (5.4) に一致する。

5.8　3 次元物体変形シミュレーション

この 3 次元のシミュレーションは、導き出した式 (5.8) を使った物体の変形シミュレーションに加えて、光の横ドップラー効果も取り入れたシミュレーションになっている。

図 34 の上で初期設定を行う。　と　で、図 34 で見えている面と見えていない奥行きの面の 2 色を設定する。そして、　で光速に比べてどれほどの速度で運動するかを $-1.0 \sim 1.0$ の範囲で設定する。速度が正であれば、物体と観測者は接近し、負であれば遠ざかる場合のシミュレーションになっている。表示する物体に関しては、　で z 軸に平行な横幅と、x 軸に平行な奥行き、y 軸に平行な高さの 3 点の設定を行う。そして、その物体が観測者からどれだけ離れているかを、　で z 座標は座標差、x 座標は距離、y 座標は観測者高さの差で設定を行う。また、横面の色が z 軸に平行な観測者には元々見えている面の色で、奥面の色が x 軸に平行な観測者には見えていなかった面の色の設定である。このシミュレーションは x, y, z 座標の内、y, z 座標を表示しているため、実際の観測者と物体は　で設定した「距離」分だけ x 座標が離れていることになる。

図 34: 3 次元シミュレーションの設定画面

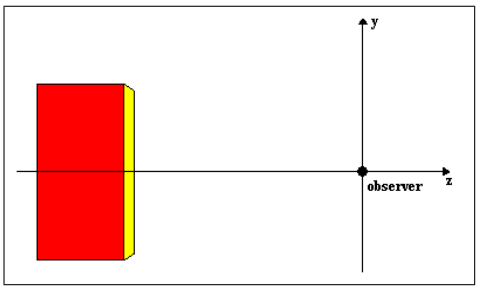


図 35: 図 34 を真横から見たイメージ図

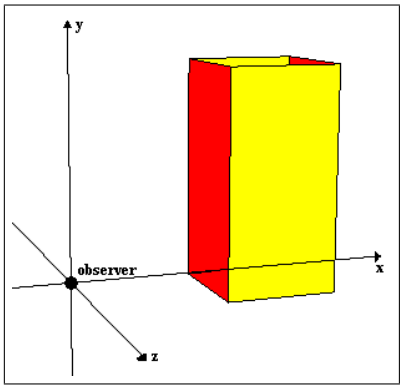


図 36: 図 34 を斜め上から見たイメージ図

図 37 と図 38 と図 39 は、全て速度と観測者との高さの差以外は図 34 と同じで、物体のサイズは横幅 100、奥行き 100、高さ 200 であり、観測者との距離は、座標者-250、距離 200 である。色も図 34 と同じで横面が赤色で、奥面の色が黄色である。そして、速度 $0.36c$ で、3 つの図は物体と観測者の y 座標差を変化させた物である。速度を 0 から $0.36c$ に上げると、長方形だった赤い物体が変形し、色もドップラー効果で変化した。また、見えていなかった黄色の奥面も見えている。しかし、テレル回転が小さいので、観測者には奥側の面はまだ見えていない。速度を上げると奥側の面も観測者に見えるはずだが、速度を上げすぎると、全て黒色になった。これではドップラー効果が解らないため、ドップラー効果も見えるように図では速度を $0.36c$ とした。

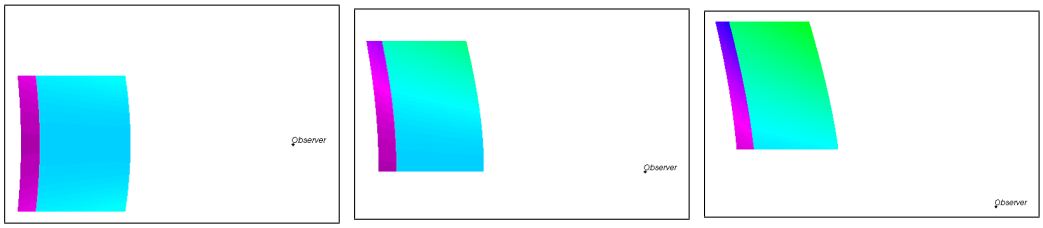


図 37: 観測者高さの差 0　図 38: 観測者高さの差 100　図 39: 観測者高さの差 200

今度は速度を $0.8c$ に上げてテレル回転のシミュレーションを行った。今度は奥面の変化を確かめたかったため、色が表示されるよう波長を調整したが、横面の色は黒色で構わなかったため波長を調整していない。速度を上げれば物体の変形も激しくなり、2 次元と同様テレル回転も大きく見られた。

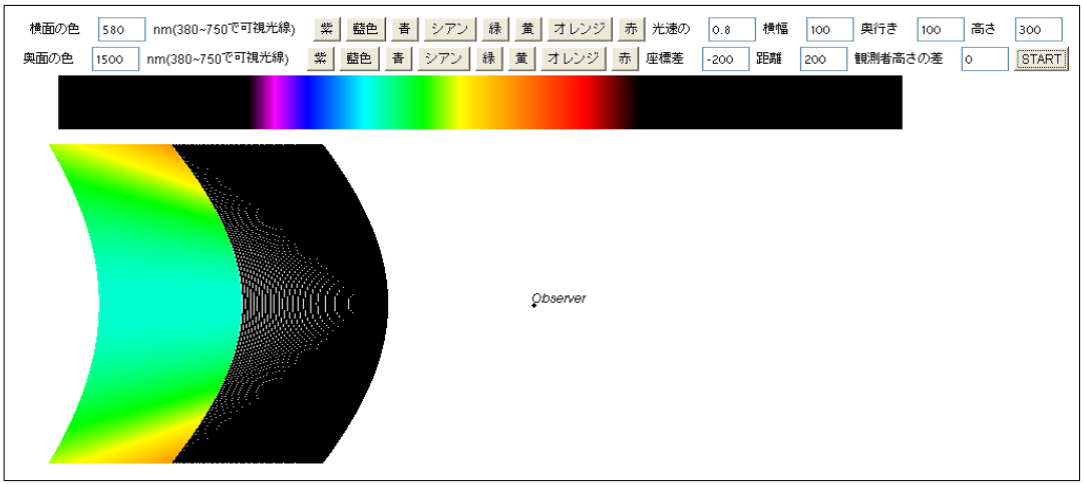


図 40: 観測者と物体が相対的に接近している速度 0.8c のシミュレーション

図 40 と同じ設定で、今度は速度を負の値にし、物体と観測者が遠ざかる場合のシミュレーションを行った。図 41 では、テレル回転を見るために、速度と奥側の面の色の波長のみ変更した。また、図 42 は、図 41 の設定から物体と観測者を遠ざけるため座標差のみ変更した。速度を負の値にし、お互いが遠ざかる場合では物体の長さが運動方向に縮んでいる。図 41 は、物体と観測者の距離が近いためテレル回転も見られるが、図 42 の距離まで遠ざかると見えている奥側の面も狭くなっている。物体と観測者の距離が遠ざかるにつれてテレル回転の回転率も減少していったためだ。

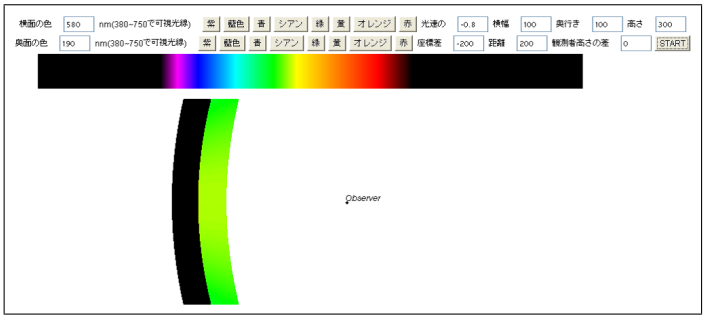


図 41: 観測者と物体が遠ざかる場合では縮んでいる

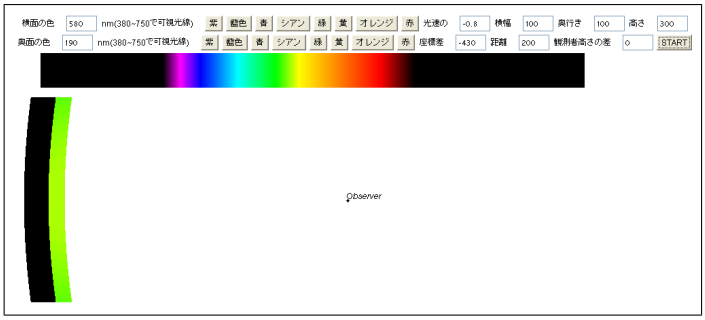


図 42: 同じ速度でもお互いが遠ざかるとテレル回転も小さくなる

今後の課題として、これに光行差及び明るさの変化も取り入れなければならないのだが、RGB で明るさを表現しているため、速度を上げ光が集まると全て白色として表示されてしまう問題も解決しなければシミュレーション結果が白一色という事態もあり得る。また、可視光線の範囲が非常に範囲が狭いため、波長を調整しなければ物体の色は速度を上げれば紫外線になってしまう。眼には見えない光を発する物体とはどのように見えるのか考えなければならない。解決策としては、光行差では明るさを変化させないことや、生き物は赤外線も発していると聞いたことがあるので、複数の波長を物体の色として設定するなどが考えられる。また、表示される赤外線と紫外線の違いも分かるように表示するなど、どこまでシミュレーションに取り入れるかは課題として残された。

6　準光速世界の擬似撮影

これまで使ってきた式を使い、3D で簡単な街などを作成し、その中で準光速の世界をシミュレーションする。ここからの 3D の街の作成及びシミュレーションには「OpenGL」を使用していく。

6.1　OpenGL とは

OpenGL(Open Graphics Library の略) とは、2D/3D グラフィックスを生成するための、OS に依存しないグラフィックス API(アプリケーションプログラミングインターフェース) である。Windows に限らず、UNIX や MacOS などでも動作し、言語も C 言語のみではなく java にも対応している。OpenGL は、クライアント / サーバ · モデルやステート · マシンという特徴があるため、C 言語や java に慣れているのであれば、注意が必要である。今回の使用言語は C++で、OpenGL の拡張機能である GLUT(OpenGL Utility Toolkit) や GLEW(OpenGL Extension Wrangler Library) も使っていく。

OpenGL を使う利点は、2D/3D グラフィックスを作成する上で、様々な機能が提供されていることだ。その 1 つの例として、3D グラフィックスを作成する際には、カメラの位置や向きなども設定でき、遠近法などの計算も全て自動で行いディスプレイに表示する。イメージ図として作成した図 35 と図 36 も OpenGL を用いて作成した図である。

図 43 は赤線で描かれたティーポットを斜め上から見ている状態だ。図 44 では、プログラムのカメラ位置と向きを設定する 1 行を変更しただけのものだ。OpenGL では、カメラの場所を変えるだけでディスプレイに表示する絵を自動的に描いてくれる。なお、赤線で描かれているティーポットは、GLUT が提供しているもので、予め用意されていた物体情報である。

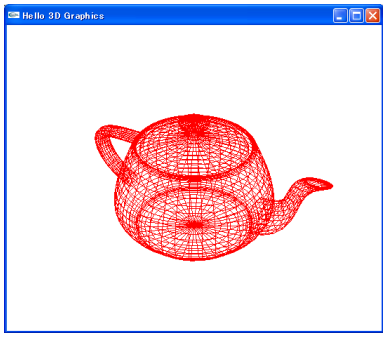


図 43: [9] サンプルプログラム 10 表示結果

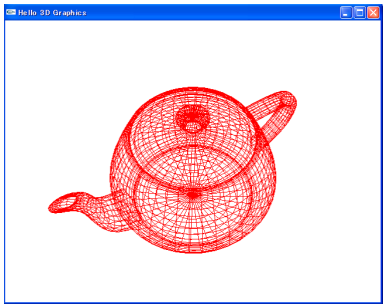


図 44: カメラの位置を変更した結果

6.2 OpenGL による 3D シミュレーション実現に向けて

シミュレーションには、観測者からどれだけ距離が違うかで 1 点 1 点結果が変化する。そのため、物体を描くには、その物体も小さな点の集合として描く必要がある。テストでは大きな点で描いても構わないが、見た目が粗くなってしまうため、できるだけ小さい点で繋げる必要がある。

また、どの座標に物体が配置され、どのような波長を持つかを 1 つ 1 つ設定しなければならない。まずは MAP を簡単に作成し、保存できるようにする必要がある。そのため、C 言語を用いて、MAP を自動作成するプログラムを作成した。

まず、最初に作成する空間の大きさを指定する。それが、終わったらどの座標に、どんな波長を持った物体を配置するかを設定する。範囲を指定する始点と終点、その直方体の範囲に物体を配置するのか消すのか、それに加えてその物体の発する波長と明るさの 5 種類のデータを入力することで、その範囲に物体を配置するようにしたい。入力は必ず始点 x< 終点 x、始点 y< 終点 y、始点 z< 終点 z となるように入力する。設定が終わると、テキストファイルに出力し保存する。シミュレーションでは、このテキストファイルを読み込み、その情報通りに物体を設置し建物を作成する。このプログラムで注意することは、作成途中の情報は配列を用いて記憶させているため、使用するパソコンの環境次第では、大きい領域を取れないということだ。このプログラムをそのまま使用する際に、コンパイルエラーが出た場合は、ヘッダファイルにある最大領域のサイズを小さくすると解決する可能性がある。

```
//空間の最大サイズ
#define MAX_LENGTH 30  //横幅 (x 軸)
#define MAX_DEPTH  220 //奥行き (z 軸)
#define MAX_HEIGHT 5   //高さ (y 軸)
```

OpenGL では、プログラム上でカメラの座標と向きを設定し表示するが、このままでは別の視点から見たい場合は、一度実行中のプログラムを終了し、座標を変更してからコンパイルし直す必要がある。そのため、実行中でもカメラの座標移動及び向きの変更をできるようにした。

図 45: 作成した空間の初期位置の景色

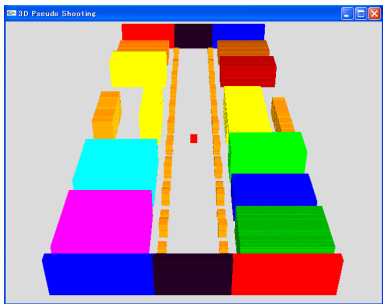


図 46: 作成した空間を上空から見下ろした景色

図 45 は、自動作成するプログラムを用いて実際に作成したもので、プログラムを実行させた初期位置の景色だ。なお、この空間の大きさは横幅 30、高さ 5、奥行き 70 である。図 46 は、作成した空間を上空から全体を見下ろした景色だ。

図 47: 図 45 からカメラを右上に向けた景色

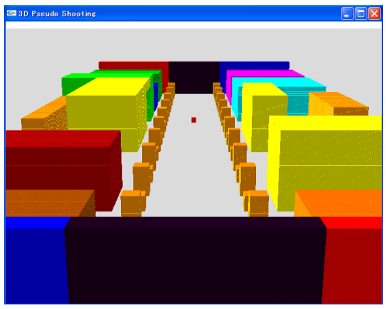


図 48: 少し上昇し初期位置の逆方向を見た景色

図 47 や図 48 は、カメラの座標や向きを変更した景色だ。この機能を使って、これからのシミュレーションがうまく動いているかを確認していく。

表 1: キーの割り当てと役割

↑キー	前進する
↓キー	後進する
←キー	左へ平行移動
→キー	右へ平行移動
PageUp キー	上昇
PageDown キー	下降
a キー	左へ 15° 回転
d キー	右へ 15° 回転
w キー	少し上を向く
x キー	少し下を向く
4 キー	左へ 90° 回転
6 キー	右へ 90° 回転
2 キー	後ろを向く
Home キー	カメラの座標を初期位置に戻す
r キー	カメラの向きを初期方向へ戻す
f キー	正面を向く
g キー	カメラの高さ座標を 1 にする
end キー	大きく前進する (現段階の設定では矢印キーの 50 倍)
8 キー	Z 軸負方向へ平行移動
9 キー	Z 軸正方向へ平行移動
Esc キー	実行しているプログラムを終了させる

6.3　OpenGL を用いた 3D の物体の変形シミュレーション

テキストファイルを読み込み、3D で表示するプログラムに式 (5.8) を用いて、物体の変形のみのシミュレーションを行った。物体などの情報は、図 46 などと同じで、速度は $0.8c$ であり、運動方向は z 軸負方向で、図で説明すると図 49 の正面、図 55 や図 56 の図上である。

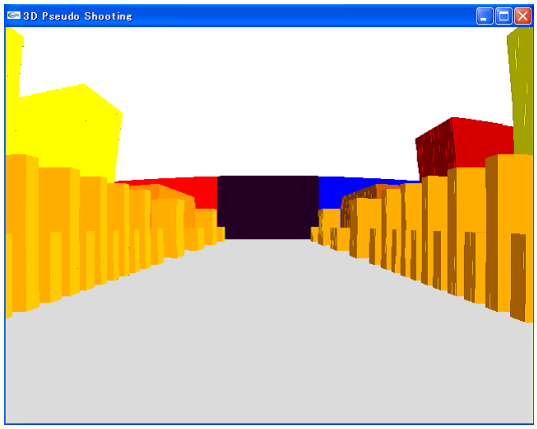


図 49: 速度 $0.8c$ の前方の物体は歪んで見える

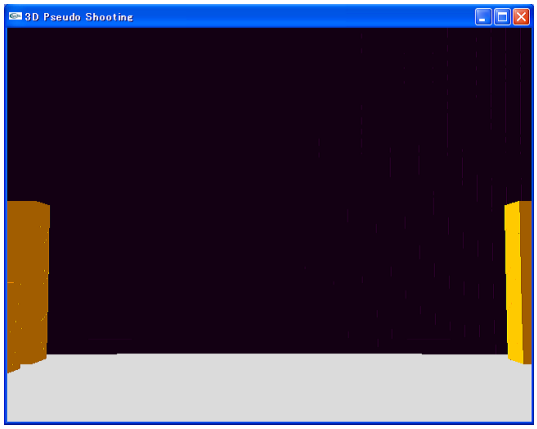


図 50: 後方ではすぐに壁があるように見える

図 49 が観測者視点の前方の景色で、図 55 が後方の景色である。図 49 をよく見ると、高さのある物体が歪んでいる。また、図 45 では等間隔に並んでいた左右にあるオレンジ色の直方体も、近くにあるものは運動方向に縮み、直方体同士の間隔も狭い。だが、遠くにある直方体は縮まず、遠くに設置されているにも関わらず直方体同士の間隔も広い。また、これらの画像は、観測者が図 46 の赤い立方体の位置に着いた瞬間の景色になる。観測者が特定の座標に着いた時に見える景色の光は、厳密には観測者には届いていない。光速に対して無視できないほどの高速運動をすると、観測者のいる座標の景色と観測者が実際に見ている景色にズレが生じるのである。そのため、図 46 の赤い立方体と同じ座標に着いた瞬間でも、この観測者の後方には図 55 のように、すぐ後ろに壁があるように見える。

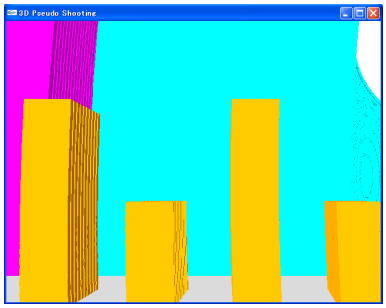


図 51: 速度 0.8c 左側は全体的に縮んで見える

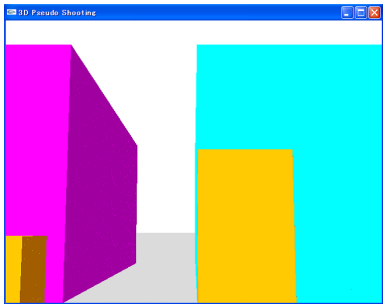


図 52: 速度 0 の時の図 51 付近の景色

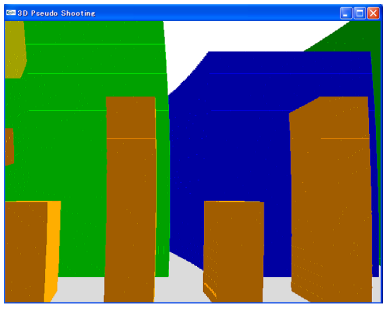


図 53: 速度 0.8c 右側も縮んで見えている

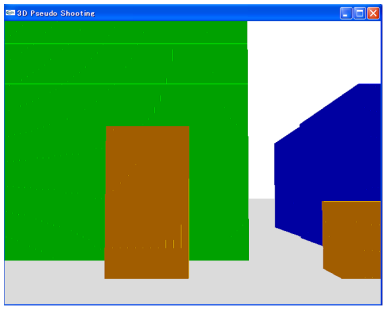


図 54: 速度 0 の時の図 53 付近の景色

図 51 は、観測者視点の景色の図 49 左側の景色で、図 53 は右側の景色である。0.8c の高速運動をしている観測者の左右の景色は、ローレンツ収縮のため運動方向上に縮んでいる。元々の景色では、図 52 や図 54 のように、奥側の直方体の間も広く、青や紫の大きな直方体の奥側も全て見えているが、運動をするとローレンツ収縮のため見えなくなっている。図 51 や図 53 の、目の前にあるオレンジ色の直方体が、奥側の大きな直方体と違いローレンツ収縮しているだけに見えるのは、物体の変形は、観測者からの距離が離れるほど効果が現れるため、観測者との距離が近いほどローレンツ収縮のみの結果に近づくためだ。観測者に近くとも、図 51 や図 53 の奥側の大きな直方体では、高さが上がるにつれて運動方向へ曲がっているのが解る。観測者に近いと、それだけ変形も小さいが、距離によって変形の割合が決まるということは変わらない。

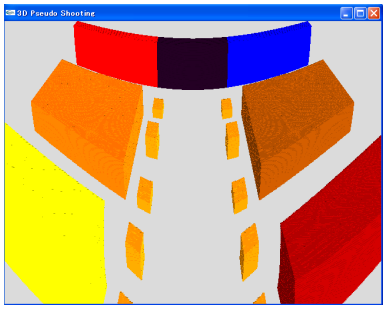


図 55: 図 49 の丁度上空から見下ろした景色

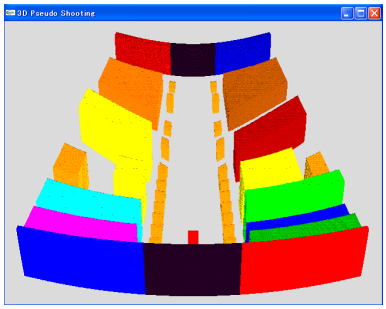


図 56: 速度 0.8c の結果図 49 の全体図

この観測者からは見えていない変化も確認するために、カメラの座標を動かしてみる。図 55 は、初期位置から真上に上昇し見下ろした景色だ。図 46 と比べると、直方体に並んでいた大きな物体や、横に真っ直ぐ長かった壁も大きく歪んでいるのが解る。この景色は、このカメラの座標で高速運動した景色ではなく、観測者の初期位置で高速運動した時に変形した景色を、現在のカメラの座標から見ているもので、カメラを移動させても景色は変化しない。図 56 は、図 46 と同じように、作成した空間全体を上空から見下ろしている。観測者から遠い前方にある物体は大きく歪み、運動方向上に伸びているが、観測者付近に近づくにつれて、物体がより縮んでいる様子が見られる。

図 57: 速度 0.8c では見えない奥行面が見える

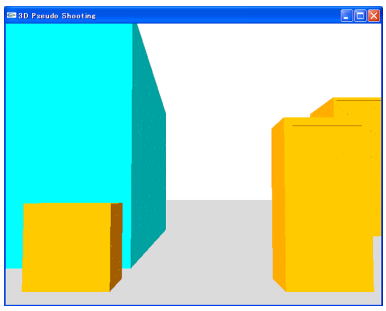


図 58: 図 57 の速度 0 の状態

図 57 は、観測者の初期座標から運動方向左を向いたものだ。図 58 は、速度 0 ではあるが、図 57 と同じ場所にいる。小さいオレンジ色の立方体とシアンの大きな直方体はぴったりと並んでいるため、図 57 の右側の小さなオレンジの直方体の奥行きの面が見えないのであれば、本来はシアンの直方体も奥行きが見えないはずだ。だが、0.8c もの高速運動をしたため、見えないはずの奥行の面が見え、まさにテレル回転が起きている。今までは、物体の変形を観測者視点ではなく、上から見た図などで図示していたが、今回で観測者視点からでもテレル回転が起きることが確認できた。

6.4　ドップラー効果を取り入れた 3D の物体の変形シミュレーション

図 49 と同様のマップで、ドップラー効果も取り入れたシミュレーションを行った。物体の位置情報や色情報などは同じだが、今回の速度は 0.2c である。

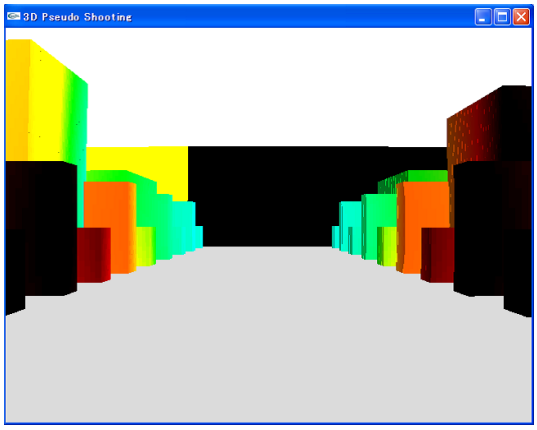


図 59: 速度 0.2c 前方の物体の変形と色の変化

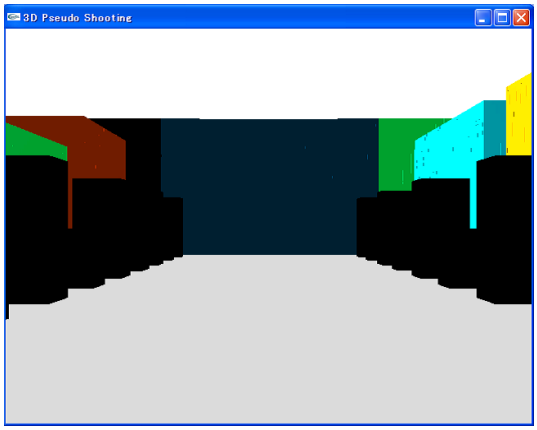


図 60: 速度 0.2c 後方では赤外線領域が多い

図 59 の観測者前方の景色では、オレンジ色だった中央の通路の両側に設置されていた小さい直方体の列が、前方の奥側ではシアンになり、そこから徐々に元のオレンジに変化し、赤外線領域にまでなっている。0.2c の速度でも、横ドップラー効果の影響で、前方の波長が伸びて観測され、赤外線になる。図 60 の観測者後方の景色では、中央の直方体の列は全て赤外線領域のため黒色で表示されている。

図 59 の左に、元々黄色だった建物があるが、その建物は半分が緑色に観測され、半分は元の色に近いままになっている。半分が元の色のままということは、この辺りの角度が速度 0.2c で横ドップラー効果を受けにくい角度ということになる。また、中央の小さな直方体の列は、元々がオレンジなので、赤外線で黒色として表示されているのは、横ドップラー効果の影響で波長が伸びて観測されているためであるが、波長が伸びて赤外線になった物体と、波長の変化があまり見られない黄色い大きな物体は同じぐらいの角度に設置されているように見える。横ドップラー効果は、速度と観測者からの角度によって、波長の変化が決まるが、図 59 では、同じぐらいの角度で波長の変化が起こらないものと、波長が伸びて観測されているものが表示されている。それどころか、図 61 に書き込んでいるが、運動方向を $\theta = 0$ として、角度が小さいものから波長の変化していないオレンジの小さい直方体が図の 4 つの中では一番角度が小さく、そこから波長が伸びた小さな直方体と波長が短くなった大きな物体が同じぐらいの角度に表示され、波長が変化していない黄色の大きな物体という順に表示されている。横ドップラー効果が、速度が一定であるなら、観測者からの角度で波長の変化が起こるはずである。この図 59 では、運動方向を $\theta = 0$ として、黄色から波長が短くなっている緑色の物体より小さい角度に波長の変化がないオレンジ色の小さな直方体がある。また、僅かではあるが波長が伸びている物体の角度より外側に、波長が短くなり緑色で

表示された物体がある。

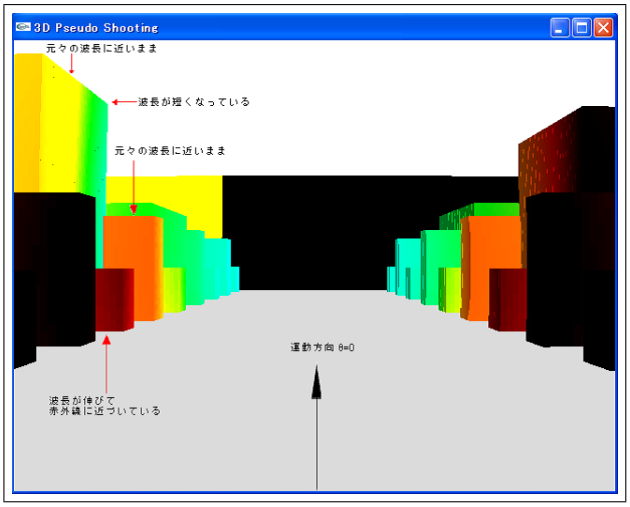


図 61: 波長が短くなっている物体より小さい角度で波長の変化がない物体がある

横ドップラー効果の可視化で、オレンジ色の波長を速度 0.2c でシミュレーションを行った。すると、速度 0.2c では波長の変化がないのは、観測者の横側である $\theta = 90°$ 付近である。また、赤外線が表示されているのは観測者の後方のみで、前方には赤外線は表示されていない。この可視化シミュレーションの光源の波長が 1 つのみだが、速度と角度が同じであれば波長の異なる複数の光源があったとしても、元の波長がどれほど変化するかは同じである。しかし、図 61 では、その通りに表示されていない。

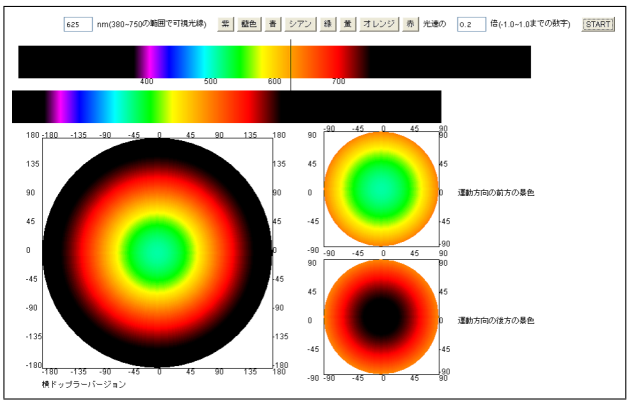


図 62: 速度 0.2c では後方 45° ぐらいでオレンジは赤外線になる

図 61 の原因は、物体の変形により、観測者が見ている座標と実際にいる座標が違うためだ。物体の変形が起きるのは、異なる物体からの光が観測者に届くまでの時間も異なるので、観測者へ同時に届く物体からの光の座標がずれるためである。しかし、物体から観測者へと向かう光は、元々の物体の座標から観測者がその光を受け取った座標まで真っ直ぐ向かう。そのため、図 61 のような観測者から景色上での物体までの角度と、物体から観測者が光を受け取る座標へ向かう光の角度には差がある。シミュレーションでは、観測者が図 46 の赤い立方体の座標へ着いた瞬間の観測者が見る景色を表示している。その座標へ着いた瞬間では、その座標の景色の光はまだ届いていないので、表示される景色はそれより前の景色である。そのため、物体からの光の角度は、図 63 の観測者が実際にいる黄色い立方体までの角度 θ_1 である。これは、赤い立方体の観測者が見ている観測者から物体までの角度 θ_2 とは違う。これが、図 61 で起きている角度と横ドップラー効果の矛盾の正体である。

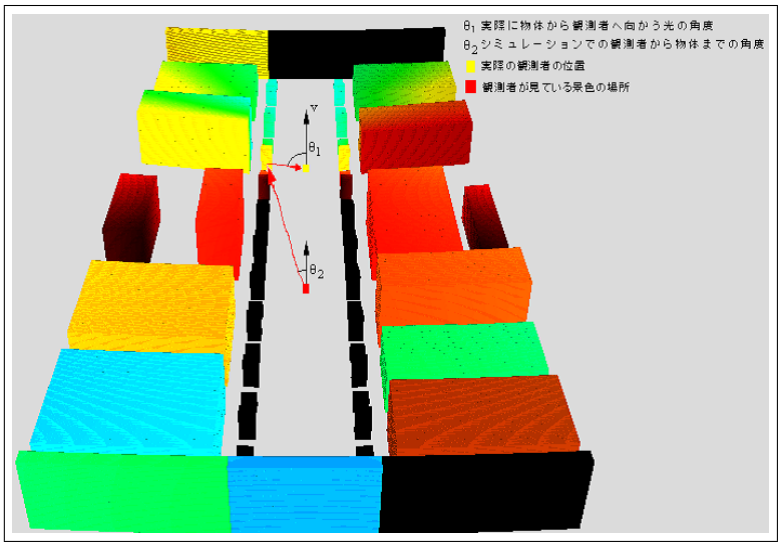


図 63: 見えている物体までの角度と実際の物体からの角度は違う

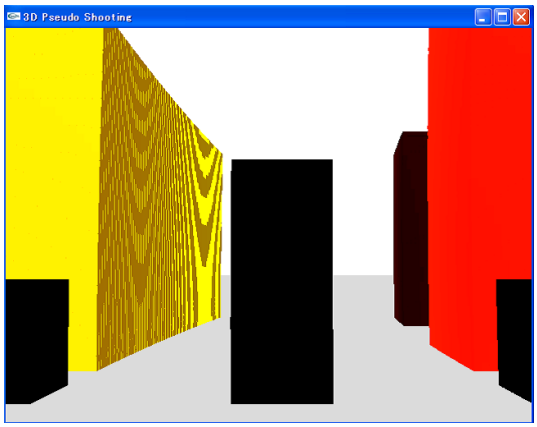


図 64: 速度 0.2c 図 59 の左側の景色

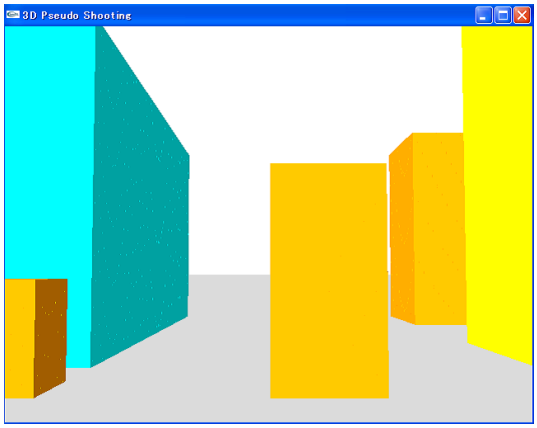


図 65: 速度 0 の時の図 64 付近の景色

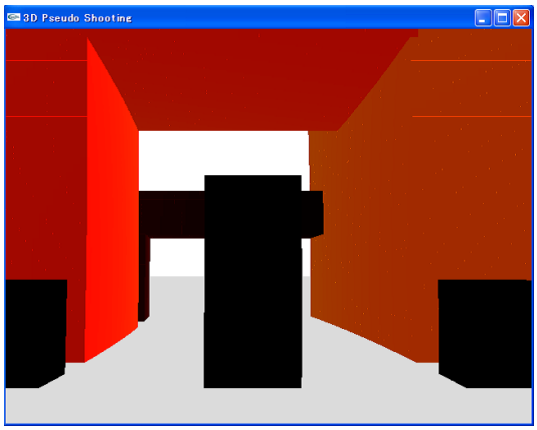


図 66: 速度 0.2c 図 59 の右側の景色

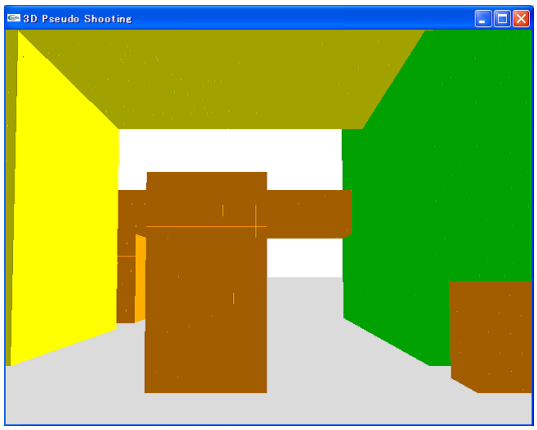


図 67: 速度 0 の時の図 66 付近の景色

物体の形状変化も取り入れると、観測者の運動方向から $\theta = 90°$ である横側の景色は、横ドップラー効果の角度 $\theta = 90°$ とは違うため、横側の景色では色が徐々に変化する物体は見られない。横側の景色は、全体的に波長が伸びているため、図 64 ではシアン色だった左側の大きな物体は黄色になり、黄色だった物体は赤色へと変化している。図 66 では、元々黄色と緑は波長が近いことと、赤の波長の範囲が広いため、黄色と緑で全く違う色で繋がっていた物体が、色が近いため 1 つの大きな物体になっているようにも見える。2 つの物体の波長が近いことで、元々の色が全く違って見えても、観測者の速度と角度次第では同じ色に見えてしまうことがある。また、前方側ですでに赤外線になっているので小さい直方体は横側でも赤外線になっている。

今回は、速度を 0.2c へ落としているため、物体の形状変化は小さい。それでも、図 59 の左奥の大きな物体は、高さが上がるにつれて歪んで見える。

6.5　マップを変更しての物体の変形とドップラー効果を含めたシミュレーション

今度は、物体の位置や形は同じだが、物体が持つ波長情報を修正し、新しいマップで同じシミュレーションを行った。物体の位置や形情報は同じままであるが、物体が持つ波長情報を変更した。観測者運動方向の奥側から右側を紫から順にし、左側を赤から順に波長の長さの順番になるよう物体の波長情報を変更した。このマップを使い、再び速度 0.2c でシミュレーションを行った。運動方向右側は、波長が短い順に並んでいるため、前方に運動すると前は紫外線になりやすく、後ろは赤外線になりやすい。逆に左側は波長が長いものから並んでいるため、前は右側に比べて紫外線になりにくく、後ろは赤外線になりにくい。

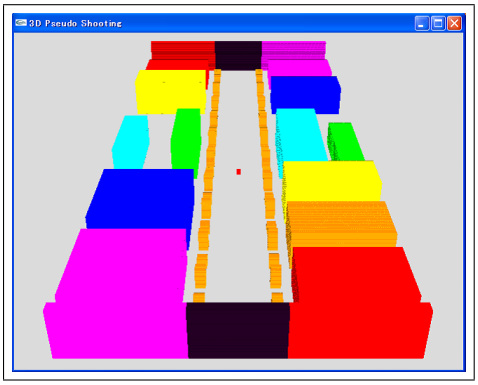


図 68: 新しいマップの全体図は波長の長さの順番で波長を設定した

速度 0.2c で前方へ高速運動すると、前方の景色の図 69 右側の奥は紫外線になったが、左側は黄色へ変化しただけで可視光線の範囲にある。後方の景色の図 70 では、左側は赤外線になっているのに比べ、右側はシアン色になっているだけである。

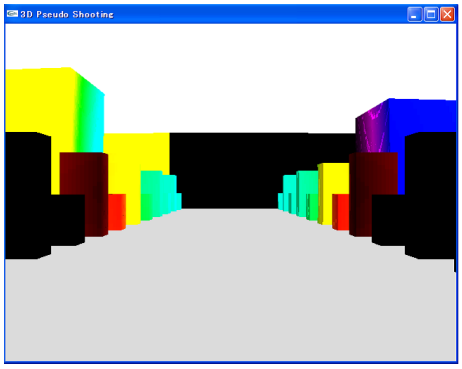


図 69: 左は長い波長なので紫外線になりにくいが右は紫外線になりやすい

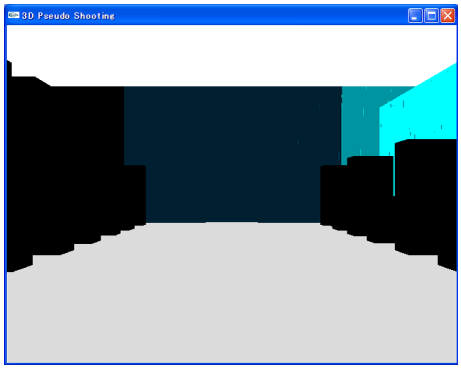


図 70: 後方でも運動方向左 (画像右) は反対と比べて赤外線になりにくい

このシミュレーションの全体図の図 71 を見ると、左側は紫外線及び赤外線になっているのは、オレンジの波長のみを持つ小さい直方体の列だけで、大きな物体は黒色で表示されている部分がない。右側は可視光線として色が表示されている物体の範囲は元の半分ほどしかなく、前方は紫外線で、後方では赤外線になっている。物体が持つ波長の情報を波長の長さで並べることにより、大きく違いが出る結果になった。

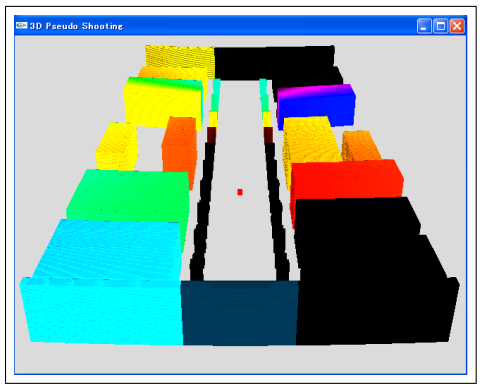


図 71: 右は波長の短い順のせいで紫外線と赤外線領域が多い

次に、速度を 0.8c にしてシミュレーションを行った。速度が 0.8 では、前方は全て紫外線になり、後方も全て赤外線になった。

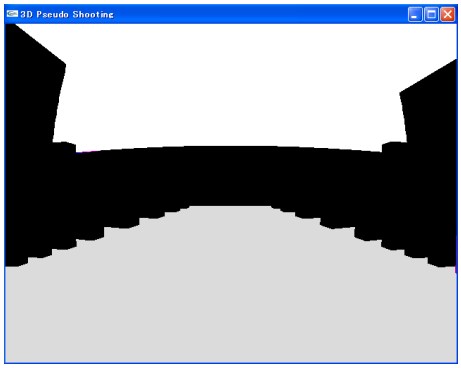
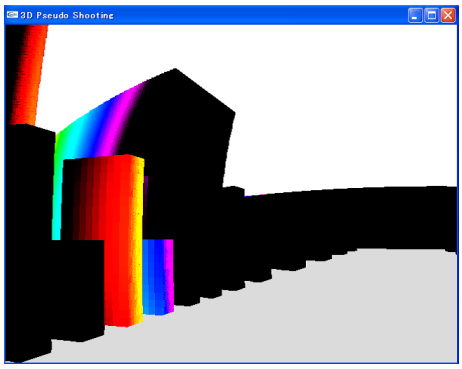


図 72: 速度 0.8c では全て紫外線になった

図 73: 後方でも全て赤外線になり目の前には壁がある

速度が 0.8c では、真横を向いても全て赤外線となっていたため、今回は真横ではなく、運動方向から 15° 横を向いた時の景色を示す。波長が長いものから並んでいる左側でも、速度 0.8c では狭い範囲しか可視光線は残らなかった。右側では、可視光線として認識できる範囲は狭く、短い波長から順に並べたはずの大きな物体で、可視光線を認識できる部分は図 75 の右上の一部分のみになっている。図 75 で可視光線として認識できている他の物体は、小

さい直方体の列のみである。また、図 74 では、紫から緑までであるがスターボウに近い状態になっている。実際に高速運動をするとスターボウが確認できる可能性があるが、このシミュレーションでは 1 つの物体に 1 つの波長を持たせるという状況下で行っているため、必ずこのように見えるとは限らない。



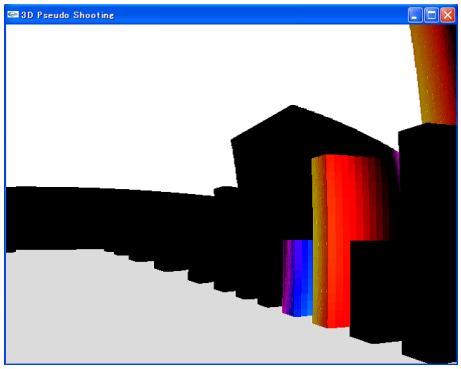


図 74: 左では狭い範囲で可視光線が残っている　図 75: 右側は図 74 より可視光線の範囲は狭い

全体図の図 76 を見ると、波長の並びが前方から短い順で並んでいる右側はほとんどが紫外線と赤外線になっている。波長が長い順に並べた左側でも一部の角度では可視光線となっている物体が残っているが、観測者からは見えていない場所だ。それでも邪魔な物体がなければ、右側より多く可視光線の物体が観測できたはずだ。速度 0.2c では、波長の並びで大きく違う結果が出ていたが、速度を上げるごとに紫外線と赤外線の領域が増えるため、差がなくなっていくようだ。

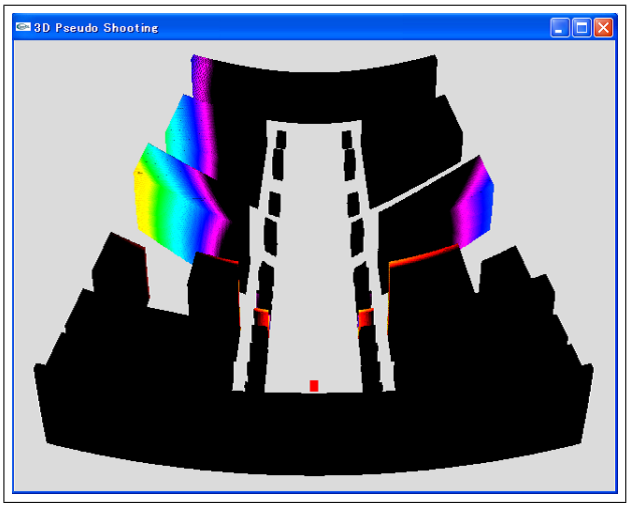


図 76: 全体を見ても波長の並びが短い順の右側はほぼ可視光線が残っていない

7　まとめ

本研究では、光の横ドップラー効果及び光行差に明るさ変化を取り入れたシミュレーションと、OpenGL を用いた 3D による見かけ上の物体の形状変化及びドップラー効果によって準光速世界の再現を行った。光の横ドップラー効果及び光行差に明るさの変化を取り入れたことによって、光速に近づくにつれ明るさが増し、元々の色の判別もできない世界になることが解った。物体の形状変化では、光のドップラー効果以外でも、運動する前方と後方では全く違う変化になり、前方の景色ではテレル回転も確認することができた。また、1 つの物体が運動方向上に長い場合は、その物体上でリング状ではないものの、部分的なスターボウが確認できた。3D のシミュレーションでは、光行差及び明るさ変化を取り入れることはできなかったことや、物体 1 つにつき 1 つの波長情報しか持たないことから、準光速世界の完全な再現には遠いことも事実である。

不思議宇宙のトムキンス [8] では、トムキンスが相対性理論の夢を見ていた。その夢の世界では、光の速度が $20km/h$ で、自転車で走るだけで相対性理論の世界を見ることができるというものだ。トムキンスが自転車で走ると、周りの景色が縮んで見えていたが、夢から覚めると、実際にはそれだけではなく、通り過ぎる人や景色は回転して見えるだろうと、教授から聞かされた。当時、ジョージ・ガモフが書いた不思議の国のトムキンスでは、見え方を考慮されていなかったため、周りの景色が縮んで描かれていた。テレル回転が発見された後、見え方の研究が進むにつれトムキンスの話は、多くの研究者から指摘があった。そのため、ラッセル・スタナードが不思議の国のトムキンスと原子の国のトムキンスを合わせ、不思議宇宙のトムキンスとして出版される際には、教授が回転して見えるだろうとトムキンスへ話す場面が追加されるなど、改訂や加筆がされた。

確かに、準光速運動の前方の景色は、伸びて見えたり、歪んで見えたり、回転して見えるだろう。しかし、逆に準光速運動の後方の景色はどうだろうか。本研究を進めると、当初の不思議宇宙のトムキンスのように、縮むだけの世界が広がっているように見えることがわかった。不思議宇宙のトムキンスでは、前方や横の景色が縮んでいるとあったが、それが後方の景色であったなら、このジョージ・ガモフが書いた世界も正しい場面も有りえた。

そして、もう 1 つジョージ・ガモフが書いた世界が正しいという可能性がある。物体の変形は、速度が一定であれば観測者との距離が離れていなければ物体の変形は限りなく小さくなる。トムキンスが夢で見た街の人々が縮んで見えたというのが、トムキンスが自転車で走っているすぐ傍を通りすぎる人を描いていたとする。すると、テレル回転が起きる十分な距離がないため、すれ違う人は回転して見えるほど物体の変形は起きず、ローレンツ収縮のみの景色に近づいていくはずである。だが、離れた場所にある建物は速度を上げると大きく変形するはずなので、トムキンスは縮んで見えた人々に目を奪われ、離れた場所にある建物などを見ていなかったと仮定するなら、トムキンスの見た景色は正しいとも言えるだろう。ただし、ドップラー効果など他の効果もあるため、そのまま縮んだ景色が見えるということはない。

今度の課題として、今回は光行差を取り入れなかったため、まずは光行差と明るさの変化を取り入れることが挙げられる。また、マップの作成及び修正に時間がかかるため、簡単に

作成や修正ができるツールの開発も必要だ。マップを簡単に作成できれば、web やアプレットとして公開することで、誰でも気軽に準光速世界の景色を楽しむことができるだろう。また、シミュレーションを動画にすることで、観測者が前方へ動くと共に景色の変化が見られれば、どのように景色が変化しているかが解りやすいだろう。また、実際の準光速世界を再現するために、より細かいマップを作製できることが望ましい。現在では、HSV モデルなどで明るさをどう変化させるのが、一番人が感じる明るさに近いのかなど、まだ明るさの変化を数値化することができていない。他にも、物体が持つ波長の情報についても理解を深める必要がある。同じ黄色の物体でも、どのような波長を持ち、人に黄色として認識されているのかは、物質によっても異なるはずである。今回は、マップに特定の波長を持つ物体を配置しシミュレーションを行ったが、地面及び空についてはドップラー効果の計算を行っていない。空からはどのような波長の電磁波が人に届いているのかも考慮に入れなければ完全な再現はできない。だが、考慮すべき点が多いからこそ、このシミュレーションは奥が深く、研究の遣り甲斐があるものだと思う。

参考文献

[1] wikipedia、「周波数」http://ja.wikipedia.org/wiki/%E5%91%A8%E6%B3%A2%E6%95%B0 (2012 年 12 月現在)

[2] wikipedia、「HSV 色空間」http://ja.wikipedia.org/wiki/HSV%E8%89%B2%E7%A9%BA%E9%96%93 (2012 年 12 月現在)

[3] 和田純夫、相対論的物理学のききどころ、岩波書店（1996）

[4] 高橋 真聡、相対性理論がわかる、技術評論社（2011）

[5] 松田卓也 / 木下篤哉、相対論の正しい間違え方、丸善 (2001)

[6] U Kraus、”First-person visualizations of the special and general theory of relativity” EUROPEAN JOURNAL PHYSICS [p.1-p.6]（2007）

[7] James Terrell、”Invisibility of the Lorentz Contraction” PHYSICAL REVIEW 116 [p.1041-p.1045]（1959）

[8] George Gamow(ジョージ・ガモフ)/Russell Stannard(ラッセル・スタナード) 著、青木薫訳、不思議宇宙のトムキンス、白揚社 (2001)

[9] MICC 著、はじめての OpenGL、工学社 (2010)

A　大きいマップを用いたシミュレーション

大きいマップを用いてシミュレーションを行った結果を簡単に示す。使用したマップの大きさは、横幅 100、高さ 30、奥行きは 200 である。

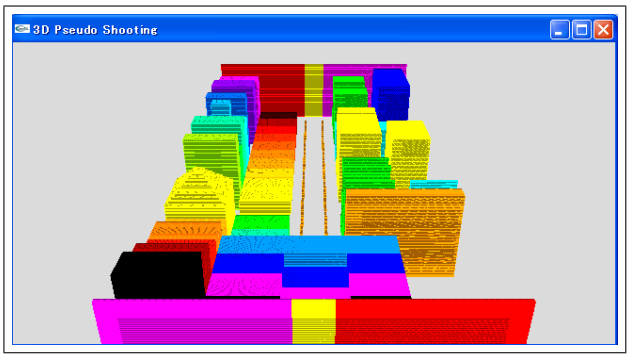


図 77: 大きいマップの全体図

マップデータが大きいせいで、色んな形を作ってみたが 1 つのシミュレーション結果に 5 分近くかかるため、サンプルになる画像データは少ない。観測者の運動方向の両側には、小さいマップでもあった小さい直方体を並べてあり、後方にはトンネル状の物体を設置してみた。

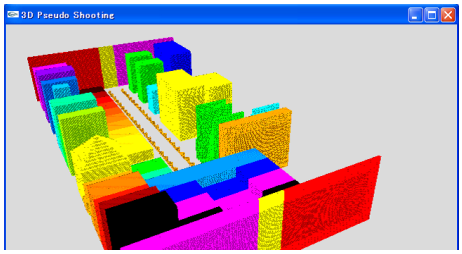


図 78: 図 77 のカメラ位置を変更した図

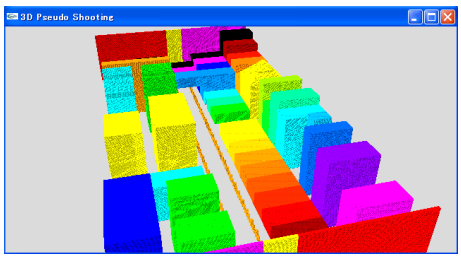


図 79: 図 78 と反対側の図

様々な形を作成したが、速度を上げた時、この辺りの変形がわかりやすい画像はない。シミュレーションに使用したプログラムソースを公開してあるため、興味がある方はシミュレーションしてみてほしい。

ここからは速度を 0.8c でドップラー効果無しのシミュレーション結果だ。小さいマップでは起こらなかったが、前方では元々の景色より、前方にある大きな壁の位置が遠くなっている。これが、接近するとローレンツ収縮した景色に近づく。そのため、実際よりも遠くにあるように見える景色から、ローレンツ収縮した景色へ一気に近づくわけであるから、観測者には速度以上で壁が接近してくるように見えるはずだ。観測者の横の景色では、後ろの壁が変形し、観測者の後ろにあるはずの壁が観測者よりも前方側へと曲がっているのが解る。

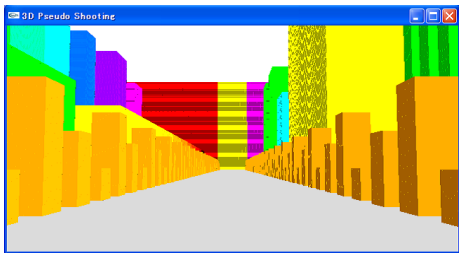


図 80: 速度 0 の時の観測者の前方の景色

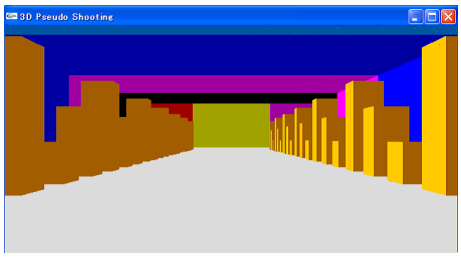


図 81: 速度 0 の時の観測者の後方の景色

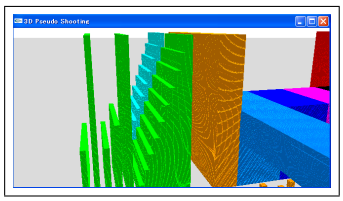


図 82: 別視点 1

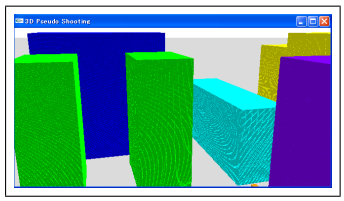


図 83: 別視点 2

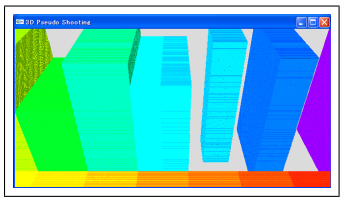


図 84: 別視点 3

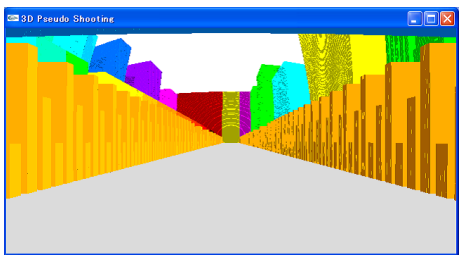


図 85: 速度 0.8c の観測者の前方の景色

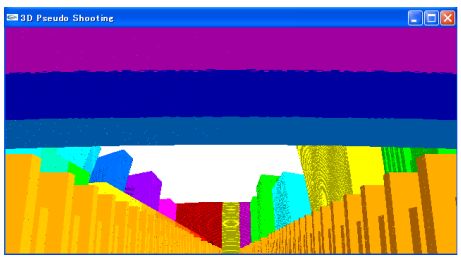


図 86: 速度 0.8c の観測者の前方やや上の景色

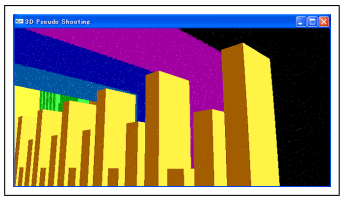


図 87: 速度 0.8c の観測者の前方右上の景色

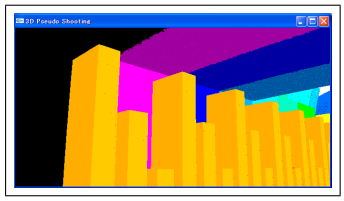


図 88: 速度 0.8c の観測者の前方左上の景色

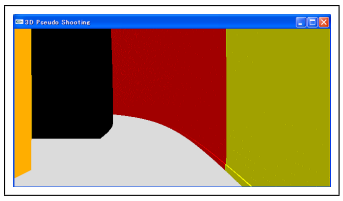


図 89: 速度 0.8c の観測者の右側の景色

先ほどのマップをそのままドップラー効果ありでシミュレーションしても、小さいマップの時とあまり変わらない景色になり面白くなかったため、少しマップを編集した。今までのマップでは、地面は物体の変形やドップラー効果の対象外としていたが、下もドップラー効果を取り入れるため、観測者の初期位置付近に小さなトンネルを設置した。

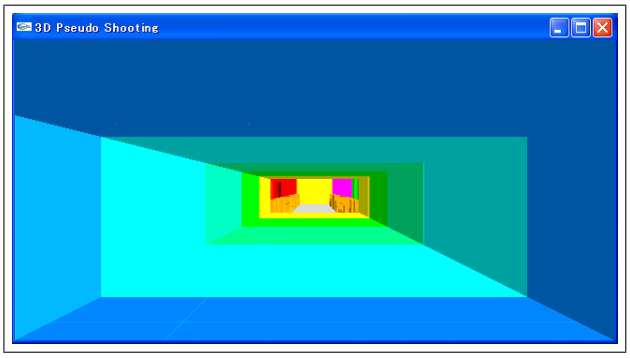


図 90: 特別設置したトンネルの速度 0 の観測者視点

視野にトンネルの横や上下も入るように狭いトンネルを通った時のシミュレーション結果を示す。

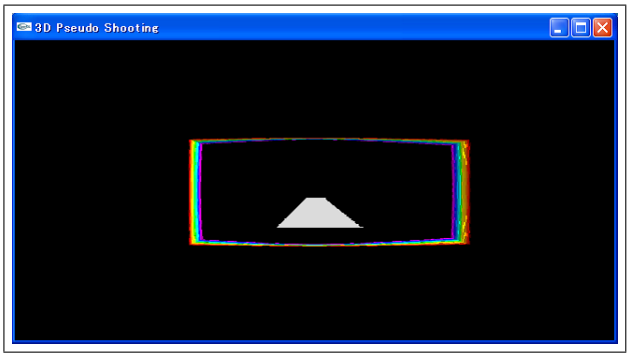


図 91: トンネル状では速度 0.8c でスターボウが観測できる

狭いトンネルの中では、はっきりとスターボウが観測することができた。本来は、地面も空でもドップラー効果は起こるため、小さいマップではなかったが、実際の世界では、スターボウが観測できる可能性があることがわかった。

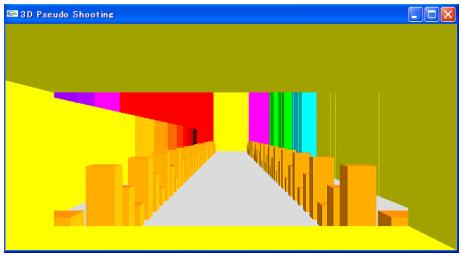


図 92: 特別設置したトンネルの出口付近

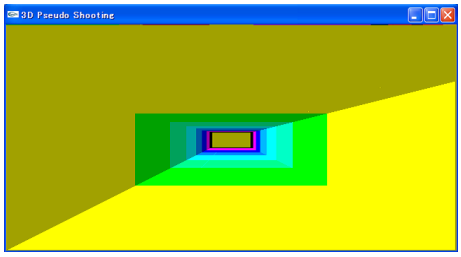


図 93: 図 92 の反対側の景色

B　ヘッダファイル

(シミュレーションの主な設定は、このヘッダファイルで行う)

```
//速度β
#define BETA 0.95

//計算する長さの割合調整　元の長さ/LengthPer
//100 なら 1/100 間隔で計算のため、100 × 100 で 1 万倍の計算
#define LengthPer 10

typedef int boolean;
#define true 1
#define false 0

#define OB false //観測者地点に赤点を表示するか
#define FIXATION true //景色の固定をするか (座標移動毎に景色の計算をしない)
#define DOPPLER false //ドップラー効果を入れるか

//空間の最大サイズ
#define MAX_LENGTH 30  //横幅　(x 軸)
#define MAX_DEPTH  70  //奥行き (z 軸)
#define MAX_HEIGHT 5   //高さ　(y 軸)

//1 つの場所から発せられる波長の数の上限
#define MAX_Lambda 2

//カメラの座標
#define CAMERAx 15
#define CAMERAy 1
#define CAMERAz 20

//カメラの座標から視点までの距離
#define CGlength 15.0

//カメラの向き
#define GAZEx CAMERAx
#define GAZEy CAMERAy
```

```
#define GAZEz CAMERAz-CGlength

//ポイントで表示する時の点のサイズ
#define PointSize 100.f

//物体の変形に使用する基準点　カメラの座標からどれだけ離れているか
#define CRITERIA_X  5.0
#define CRITERIA_Y  0.0
#define CRITERIA_Z -2.0

//カメラの回転率
#define ROTATION 15.0 //この角度分回転させる
#define M_PI 3.14159265358979
#define TurnoverRate M_PI*ROTATION/180.0

//移動距離
#define Stride 1.0

//光源の位置
#define LigPosX 0.4f
#define LigPosY 0.2f
#define LigPosZ 0.3f
#define LigPosW 0.f

//光源の設定 環境光
#define AMB_R 0.5f
#define AMB_G 0.5f
#define AMB_B 0.5f
#define AMB_A 1.f

//光源の設定 拡散光
#define DIFF_R 1.f
#define DIFF_G 1.f
#define DIFF_B 1.f
#define DIFF_A 1.f

//光源の設定　鏡面光
#define SPEC_R 1.f
#define SPEC_G 1.f
#define SPEC_B 1.f
#define SPEC_A 1.f

//材質の鏡面光の指数
#define glSHIN 100.2f

//地面の色設定
//鏡面光
#define GROUND_SPEC_R 0.8f
#define GROUND_SPEC_G 0.8f
#define GROUND_SPEC_B 0.8f
```

```
#define GROUND_SPEC_A 1.f
//環境光と拡散光
#define GROUND_AandD_R 0.8f
#define GROUND_AandD_G 0.8f
#define GROUND_AandD_B 0.8f
#define GROUND_AandD_A 1.f



//---------------------------------色の波長設定---------------------------------
//可視光線の範囲
#define LIMIT_PURPLE 380
#define LIMIT_RED    750

//国際照明委員会（CIE）が 1931 年に定めた CIE 標準表色系
//赤 700 緑 546.1 青 435.8
#define RED   700
#define GREEN 546
#define BLUE  436

//適当に決めただいたいの波長
#define ORANGE 625
#define YELLOW 570
#define CYAN   500
#define INDIGO 420
#define PURPLE 405
/*メモ
波長 577〜597nm がオレンジ　オレンジ色の光（波長 625 - 590 nm）
波長 570〜585nm が黄色
波長 500〜520nm がシアン
波長 430〜450nm が藍色
波長 380〜430nm が紫
*/
/*旧バージョン
#define ORANGE 605
#define YELLOW 580
#define CYAN   490
#define INDIGO 420
#define PURPLE 405
*/


typedef struct{
boolean object;
int WaveSum;
int lambda[MAX_Lambda];
double Yi[MAX_Lambda];
}XYZobject;

//遠ざかると物体が縮むためどれだけ重なるかによって配列の長さが変わる …（MAX_Lambda）
```

```
typedef struct{
boolean object;
double z;
int WaveSum;
int lambda[MAX_Lambda];
double Yi[MAX_Lambda];
}XYobject;

typedef struct{
boolean object;
boolean visibility;
int WaveSum;
int lambda[MAX_Lambda];
double Yi[MAX_Lambda];
boolean left;   //X 軸負方向の面を描く必要があるか
boolean right;  //X 軸正
boolean buttom; //Y 軸負
boolean top;    //Y 軸正
boolean front;  //Z 軸負
boolean back;   //Z 軸正
}XYZvisibility;

typedef struct{
double R;
double G;
double B;
}RGB;

typedef struct{
double V;
double S;
double H;
}HSV;

typedef struct{
boolean object;
double R;
double G;
double B;
}XYZ;

int round1(double str);
void Ground();
HSV RGBtoHSV(RGB rgb);
RGB HSVtoRGB(HSV hsv);
int redJudg(int H0);
int greenJudg(int H0);
int blueJudg(int H0);
RGB DopplerSynthesis(double X, double Y, double Z, int lambda[], double Yi[], int sum);
double Lorentz(double l, double vc);
```

```
double Deformation(double LL, double Y, double zC, double a, double h, double b, double vc);
```

C　マップ作成のプログラムソース

```
#include <stdio.h>
#include <stdlib.h>
#include <my/myhead.h>

#pragma warning(disable : 4996)

/*
  ファイルの仕様
  1 行目　　:XYZ 座標の大きさ (1 行目は XYZ の数値 3 つのみ)
  2 行目以降:座標と物体の有無と波長と明るさ、この座標からいくつ波長が出ているか
            x, y, z, n, λ, Yi, sum を (数値 7 つ) 書き込む
*/
int main(int argc, char *argv[]){
int x1, y1, z1, x2, y2, z2;
int MAXx=0, MAXy=0, MAXz=0;
boolean ob;
char name[64];
FILE *fp;
XYZobject xyz[MAX_LENGTH][MAX_HEIGHT][MAX_DEPTH];
int x, y, z, tmpX=-1, tmpY=-1, tmpZ=-1;
int lambda, sum, count;
double Yi; //明るさ　0〜1.0


if(argc > 2){
printf("引数は無し、または読み込むテキストファイル名 1 つのみです。\n");
scanf("%d", &x);
exit(1);
}

/*argc=1 ならば座標を新たに生成し初期化する*/
if(argc==1){
do{
printf("3D 空間を作成します。空間の大きさを指定してください。\n");
printf("横幅 x(%d 以下), 高さ y(%d 以下), 奥行き z(%d 以下) の順番に 3 つ入力してください。
\n", MAX_LENGTH, MAX_HEIGHT, MAX_DEPTH);
scanf("%d %d %d", &MAXx, &MAXy, &MAXz);
}while(!(1<=MAXx && MAXx<=MAX_LENGTH && 1<=MAXy && MAXy<=MAX_HEIGHT && 1<=MAXz && MAXz<=MAX_DEPTH)

for(x=0; x<MAXx; x++){
for(y=0; y<MAXy; y++){
for(z=0; z<MAXz; z++){
xyz[x][y][z].object = false;
xyz[x][y][z].WaveSum = 0;
```

```
xyz[x][y][z].lambda[0] = 0;
xyz[x][y][z].Yi[0] = 0.0;
}
}
}

printf("初期化が完了しました。\n\n");

}else{
if ((fp = fopen(argv[1], "r")) == NULL){
fprintf(stderr, "ファイル%s がありません。\n", argv[1]);

//表示停止用
scanf("%d", &x);

exit(1);

}else{
/*引数があるのでファイル読み込み*/
fscanf(fp, "%d %d %d", &MAXx, &MAXy, &MAXz);

while(fscanf(fp, "%d %d %d %d %lf %d", &x, &y, &z, &ob, &lambda, &Yi, &sum) != EOF){

//前回の tmp と座標が同じであれば複数波長が設定されているため配列使用
if(!(tmpX==x && tmpY==y && tmpZ==z)){
//前回と座標が違う

count = 0;

xyz[x][y][z].object = ob;
xyz[x][y][z].WaveSum = sum;
xyz[x][y][z].lambda[count] = lambda;
xyz[x][y][z].Yi[count] = Yi;

}else{
//波長が複数設定されている

count++;

xyz[x][y][z].lambda[count] = lambda;
xyz[x][y][z].Yi[count] = Yi;
}

tmpX = x;
tmpY = y;
tmpZ = z;
}

/*最後にファイルを閉じる*/
fclose(fp);
```

```
printf("読み込みが完了しました。\n\n");
}
}

/*初期化及び読み込み完了　街の作成へ*/
/*開始座標＞終点座標＞物体の有無＞ RGB の 4 回にわけて入力*/
/*手抜きのため、入力座標は小→大の順でなければエラーを出す*/
while(1){
printf("始点座標は全て 0 以上、x=%d y=%d z=%d 未満で入力してください。\n", MAXx, MAXy, MAXz);
printf("始点の座標を入力してください。(-1 -1 -1 入力で終了)\n");
printf("始点と終点は全て、始点 < 終点の順番となるように入力してください。\n");

scanf("%d %d %d", &x1, &y1, &z1);

/*入力が座標内か判定 正しければ終点入力へ*/
if((0<=x1 && x1<MAXx) && (0<=y1 && y1<MAXy) && (0<=z1 && z1<MAXz)){
/*正しく終点座標入力 or-1-1-1 入力まで繰り返し*/
while(1){
printf("終点座標は x:%d 以上%d 未満 y:%d 以上%d 未満 z:%d 以上%d 未満\n", x1, MAXx, y1, MAXy, z1, MAXz);
printf("終点の座標を入力してください。(-1 -1 -1 入力で始点入力に戻る)\n");

scanf("%d %d %d", &x2, &y2, &z2);

if(((0<=x2 && x2<MAXx && x1<=x2) && (0<=y2 && y2<MAXy && y1<=y2) && (0<=z2 && z2<MAXz && z1<=z2))
 || (x2==-1 && y2==-1 && z2==-1))
break;

printf("入力エラーです。もう一度入力してください。\n");
}
}else{
/*決められた以外の範囲が指定された　終了するか判定する*/
if(x1==-1 && y1==-1 && z1==-1)
break;
}

/*-1-1-1 か、正しい座標が入力された。*/
if(x2!=-1 && y2!=-1 && z2!=-1){
while(1){
printf("物体を作成するなら%d、範囲内の物体を削除するなら%d を入力してください。\n", true, false);
scanf("%d", &ob);
if(ob==false || ob==true)
break;
}

if(ob == 1){
//物体作成が確定した
count = 0;

/*続けて波長及び明るさ入力*/
```

```
while(MAX_Lambda > count){
while(1){
printf("波長 (nm) と明るさ (0〜1.0) を入力してください。\n");
scanf("%d %lf", &lambda, &Yi);

//正しく入力されたか
if(0.0 <= Yi && Yi <= 1.0)
break;
else
printf("入力エラーです。\n 正しく入力してください。\n");
}

/*範囲内の物体の作成*/
for(x=x1; x<=x2; x++){
for(y=y1; y<=y2; y++){
for(z=z1; z<=z2; z++){
xyz[x][y][z].object = true;
xyz[x][y][z].WaveSum = count + 1;
xyz[x][y][z].lambda[count] = lambda;
xyz[x][y][z].Yi[count] = Yi;
}
}
}

count++;

if(MAX_Lambda > count){
//この座標から複数の波長が出ているなら続けて波長の設定を行う
printf("この座標から、複数の光が出ている場合は、続けて波長の設定ができます。\n");
printf("追加で波長の設定を行う場合は 1、波長の設定を終了するなら 0 を入力してください。\n");
scanf("%d", &ob);
if(ob==0)
break;

printf("現在の座標から出る波長は、あと%d 回まで追加で設定を行えます。\n", MAX_Lambda - count);
}
}

printf("範囲内の物体の作成が完了しました。\n");
}else{
/*物体を消去*/
for(x=x1; x<=x2; x++){
for(y=y1; y<=y2; y++){
for(z=z1; z<=z2; z++){
xyz[x][y][z].object = false;
xyz[x][y][z].WaveSum = 0;
}
}
}
```

```
printf("範囲内の物体消去が完了しました。\n");
}
}
}


/*最後にファイルへ出力して終了*/
while(1){
    printf("出力するファイル名を入力してください。\n");
scanf("%s", name);

if ((fp = fopen(name, "w")) == NULL) {
fprintf(stderr, "ファイル%s が開けません。\n", name);
}else
break;
}

/*問題点：ここで最初の 1 行だけ xyz のサイズ情報のみを入力しておく*/
fprintf(fp, "%d\t %d\t %d\n", MAXx, MAXy, MAXz);

for(x=0; x<MAXx; x++){
for(y=0; y<MAXy; y++){
for(z=0; z<MAXz; z++){
count = 0;
fprintf(fp, "%d\t %d\t %d\t %d\t %d\t %f\t %d\n",
x, y, z, xyz[x][y][z].object,
xyz[x][y][z].lambda[count], xyz[x][y][z].Yi[count], xyz[x][y][z].WaveSum);

if(xyz[x][y][z].WaveSum > 1){ /*複数波長が設定されている場合*/
count = 1;

while(xyz[x][y][z].WaveSum > count){
fprintf(fp, "%d\t %d\t %d\t %d\t %d\t %f\t %d\n",
x, y, z, xyz[x][y][z].object,
xyz[x][y][z].lambda[count], xyz[x][y][z].Yi[count], xyz[x][y][z].WaveSum);
count++;
}
}
}
}
}

fclose(fp);

return 0;
}
```

D　OpenGL を使った横ドップラー効果含むシミュレーション (3Ddoppler.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <my/myhead.h>
#include <gl/glut.h>

#pragma warning(disable : 4996)

#define DRAW_LENGTH 1.0/LengthPer

/*
プログラムの仕様
画面が表示されて Esc キーを押すと終了
矢印キーで前後左右へ平行移動
PageUp で上昇し、PageDown で下降
Home キーで初期位置へ戻る
g キーで、高さを 1 にする

adwx キーで視線の変更
  a:視線を左へ回転させる
  d:視線を右へ回転させる
  w:視線を上へ向ける
  x:視線を下へ向ける
f キーで正面を見る (高さのみ変更)
r キーで初期の視線設定に戻す
*/

//空間のデータ
XYZvisibility xyz[MAX_LENGTH][MAX_HEIGHT][MAX_DEPTH];
XYobject xyzSave1[MAX_LENGTH*LengthPer][MAX_HEIGHT*LengthPer];
XYobject xyzSave2[MAX_LENGTH*LengthPer][MAX_HEIGHT*LengthPer];
//読み込んだ空間の大きさ
int MAXx=0, MAXy=0, MAXz=0;

//カメラの位置座標
double CameraX=CAMERAx, CameraY=CAMERAy, CameraZ=Lorentz(CAMERAz, BETA);
//カメラの視線の座標
double GazeX=GAZEx, GazeY=GAZEy, GazeZ=GAZEz;
//カメラの向きの角度 (座標 z 軸平行に負方向を向くと 0 度になるよう調整)
double thetaXZ=acos((GAZEz-CAMERAz)/CGlength);

//物体の変形に使用する基準の座標
//マクロのカメラ初期値を代入すればカメラ移動しても景色変化なし
//実際のカメラの座標を代入すれば移動すると景色変化有り
double observerX=CAMERAx, observerY=CAMERAy, observerZ=Lorentz(CAMERAz, BETA);
double CriteriaX=observerX+CRITERIA_X, CriteriaY=observerY+CRITERIA_Y, CriteriaZ=observerZ+CRITERI
```

```
//光源 (環境光拡散光鏡面光位置)
GLfloat lightAmb[] = {AMB_R, AMB_G, AMB_B, AMB_A};
GLfloat lightDiff[] = {DIFF_R, DIFF_G, DIFF_B, DIFF_A};
GLfloat lightSpec[] = {SPEC_R, SPEC_G, SPEC_B, SPEC_A};

//光源の位置
GLfloat lightPos[] = {LigPosX, LigPosY, LigPosZ, LigPosW};

//材質 (環境光拡散光鏡面光鏡面指数)
GLfloat Spec[] = { 0.628281f, 0.555802f, 0.366065f, 1.f};
GLfloat Shin = glSHIN;

//ファイルの読み込み
void Architecture(FILE *fp){
int x, y, z, ob, lambda, sum;
int count=0, tmpSum, tmpx, tmpy, tmpz;
double Yi;

fscanf(fp, "%d %d %d", &MAXx, &MAXy, &MAXz);

    while(fscanf(fp, "%d %d %d %d %lf %d", &x, &y, &z, &ob, &lambda, &Yi, &sum) != EOF){
count = 1;

        xyz[x][y][z].object = ob;
        xyz[x][y][z].lambda[0] = lambda;
        xyz[x][y][z].Yi[0] = Yi;
        xyz[x][y][z].WaveSum = sum;

tmpSum = sum;
tmpx = x;
tmpy = y;
tmpz = z;

//同じ座標に複数波長が設定されていた場合
while((tmpSum>count) && (tmpx==x && tmpy==y && tmpz==z) && MAX_Lambda>count){
fscanf(fp, "%d %d %d %d %lf %d", &x, &y, &z, &ob, &lambda, &Yi, &sum);

xyz[x][y][z].lambda[count] = lambda;
    xyz[x][y][z].Yi[count] = Yi;

count++;
}
}
}

void cube(double x, double y, double z, RGB rgb){
float Rf, Gf, Bf;

//直接光が当たっていない面の色設定
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT);
```

```
Rf = (float)(rgb.R*0.9);
Gf = (float)(rgb.G*0.9);
Bf = (float)(rgb.B*0.9);
glColor4f(Rf, Gf, Bf, 1.f);

//光が当たっている面の材質設定
glColorMaterial(GL_FRONT_AND_BACK, GL_DIFFUSE);
Rf = (float)rgb.R;
Gf = (float)rgb.G;
Bf = (float)rgb.B;
glColor4f(Rf, Gf, Bf, 1.f);

//立方体の描画
glPushMatrix();
glTranslatef(x, y, z);
glutSolidCube(DRAW_LENGTH);
glPopMatrix();
}

void clearSave1(){
int x, y;
for(x=0; x<MAX_LENGTH*LengthPer; x++){
for(y=0; y<MAX_HEIGHT*LengthPer; y++){
xyzSave1[x][y].object = false;
}
}
}

void clearSave2(){
int x, y;
for(x=0; x<MAX_LENGTH*LengthPer; x++){
for(y=0; y<MAX_HEIGHT*LengthPer; y++){
xyzSave2[x][y].object = false;
}
}
}

void Save1(int x, int y, int z, double X, double Y, double Z){
int arrayX, arrayY;  //新しい２次元配列の配列番号用
double xDif, yDif, zDif; //基準と計算する座標との差
double CoordinateZ1, oldZ;
RGB rgb;

//代入する配列番号の計算　誤差をなくすために念のため四捨五入
arrayX = round1(X/DRAW_LENGTH);
arrayY = round1(Y/DRAW_LENGTH);

//距離差の計算
xDif = X - CriteriaX;
yDif = Y - CriteriaY;
```

```
zDif = Z - Lorentz(CriteriaZ, BETA);

//新しい z 座標を計算　引数：基準の XYZ, そこからの座標差 xyz、速さ beta の 7 つ
xyzSave1[arrayX][arrayY].z = Deformation(CRITERIA_X, CRITERIA_Y, CRITERIA_Z, xDif, yDif, zDif, BET

//ドップラー計算 引数：XYZ 座標差　速度　波長　明るさ　光の合計 7 つ
rgb = DopplerSynthesis(observerX-X, observerY-Y, observerZ-Z, xyz[x][y][z].lambda, xyz[x][y][z].Yi

//描写 前に描いた z 座標に物体があるならそこまでの間を埋める
//前に書いた z 座標は毎回計算させることで解決した
if(z>0){
if(xyzSave2[arrayX][arrayY].object==true){
zDif = (Z-DRAW_LENGTH) - Lorentz(CriteriaZ, BETA);
oldZ = Deformation(CRITERIA_X, CRITERIA_Y, CRITERIA_Z, xDif, yDif, zDif, BETA);
for(CoordinateZ1=oldZ; CoordinateZ1 < xyzSave1[arrayX][arrayY].z; CoordinateZ1 += DRAW_LENGTH){
cube(X, Y, CoordinateZ1, rgb);
}
}}
cube(X, Y, xyzSave1[arrayX][arrayY].z, rgb);

//配列に保存
xyzSave1[arrayX][arrayY].object = true;
xyzSave1[arrayX][arrayY].WaveSum = xyz[x][y][z].WaveSum;
memcpy(xyzSave1[arrayX][arrayY].lambda, xyz[x][y][z].lambda, MAX_Lambda);
memcpy(xyzSave1[arrayX][arrayY].Yi, xyz[x][y][z].Yi, MAX_Lambda);
}

void Save2(int x, int y, int z, double X, double Y, double Z){
int arrayX, arrayY;  //新しい 2 次元配列の配列番号用
double xDif, yDif, zDif; //基準と計算する座標との差
double CoordinateZ2, oldZ;
RGB rgb;

//代入する配列番号の計算　誤差をなくすために念のため四捨五入
arrayX = round1(X/DRAW_LENGTH);
arrayY = round1(Y/DRAW_LENGTH);

//距離差の計算
xDif = X - CriteriaX;
yDif = Y - CriteriaY;
zDif = Z - Lorentz(CriteriaZ, BETA);

//新しい z 座標を計算　引数：基準の XYZ, そこからの座標差 xyz、速さ beta の 7 つ
xyzSave2[arrayX][arrayY].z = Deformation(CRITERIA_X, CRITERIA_Y, CRITERIA_Z, xDif, yDif, zDif, BET

//ドップラー計算 引数：XYZ 座標差　速度　波長　明るさ　光の合計 7 つ
rgb = DopplerSynthesis(observerX-X, observerY-Y, observerZ-Z, xyz[x][y][z].lambda, xyz[x][y][z].Yi

//描写 前に描いた z 座標に物体があるならそこまでの間を埋める
//前に書いた z 座標は毎回計算させることで解決した
```

```
if(z>0){
if(xyzSave1[arrayX][arrayY].object==true){
zDif = (Z-DRAW_LENGTH) - Lorentz(CriteriaZ, BETA);
oldZ = Deformation(CRITERIA_X, CRITERIA_Y, CRITERIA_Z, xDif, yDif, zDif, BETA);
for(CoordinateZ2=oldZ; CoordinateZ2 < xyzSave2[arrayX][arrayY].z; CoordinateZ2 += DRAW_LENGTH){
cube(X, Y, CoordinateZ2, rgb);
}
}}
cube(X, Y, xyzSave2[arrayX][arrayY].z, rgb);

//配列に保存
xyzSave2[arrayX][arrayY].object = true;
xyzSave2[arrayX][arrayY].WaveSum = xyz[x][y][z].WaveSum;
memcpy(xyzSave2[arrayX][arrayY].lambda, xyz[x][y][z].lambda, MAX_Lambda);
memcpy(xyzSave2[arrayX][arrayY].Yi, xyz[x][y][z].Yi, MAX_Lambda);
}

void XYdrawSave1(int z, double Z, int count){
int x, y;
double Y, X;
double sideTop, sideButtom; //側面の上限と下限

for(y=0; y<=MAXy-1; y++){
for(x=0; x<=MAXx-1; x++){
if(xyz[x][y][z].visibility == true){
if((count%LengthPer==0 && xyz[x][y][z].front==true) || ((count+1)%LengthPer==0 && xyz[x][y][z].bac
if(xyz[x][y][z].front==true && count%LengthPer==0){  //front のみ描く
for(Y=y; Y<y+1-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
for(X=x; X<x+1-DRAW_LENGTH/10.0; X+=DRAW_LENGTH){
Save1(x ,y, z, X, Y, Z);
}}
}else{ if(xyz[x][y][z].back==true && (count+1)%LengthPer==0){  //back のみ描く
for(Y=y; Y<y+1-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
for(X=x; X<x+1-DRAW_LENGTH/10.0; X+=DRAW_LENGTH){
Save1(x ,y, z, X, Y, Z);
}}
}}
}else{
//上下左右を描く
sideTop = 1.0;
sideButtom = 0.0;

//top を描く
if(xyz[x][y][z].top==true){
sideTop -= DRAW_LENGTH;
Y=y+1.0-DRAW_LENGTH;
for(X=x; X<x+1-DRAW_LENGTH/100.0; X+=DRAW_LENGTH){
Save1(x ,y, z, X, Y, Z);
}
}
```

```
//buttom を描く
if(xyz[x][y][z].buttom==true){
sideButtom += DRAW_LENGTH;
Y=y;
for(X=x; X<x+1-DRAW_LENGTH/10.0; X+=DRAW_LENGTH){
Save1(x ,y, z, X, Y, Z);
}
}

//right を描く
if(xyz[x][y][z].right==true){
X=x+1-DRAW_LENGTH;
for(Y=y+sideButtom; Y<y+sideTop-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
Save1(x ,y, z, X, Y, Z);
}
}

//left を描く
if(xyz[x][y][z].left==true){
X=x;
for(Y=y+sideButtom; Y<y+sideTop-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
Save1(x ,y, z, X, Y, Z);
}
}
}
}}}
}

void XYdrawSave2(int z, double Z, int count){
int x, y;
double Y, X;
double sideTop, sideButtom; //側面の上限と下限

for(y=0; y<=MAXy-1; y++){
for(x=0; x<=MAXx-1; x++){
if(xyz[x][y][z].visibility == true){
if((count%LengthPer==0 && xyz[x][y][z].front==true) || ((count+1)%LengthPer==0 && xyz[x][y][z].bac
if(xyz[x][y][z].front==true && count%LengthPer==0){  //front のみ描く
for(Y=y; Y<y+1-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
for(X=x; X<x+1-DRAW_LENGTH/10.0; X+=DRAW_LENGTH){
Save2(x ,y, z, X, Y, Z);
}}
}else{ if(xyz[x][y][z].back==true && (count+1)%LengthPer==0){  //back のみ描く
for(Y=y; Y<y+1-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
for(X=x; X<x+1-DRAW_LENGTH/10.0; X+=DRAW_LENGTH){
Save2(x ,y, z, X, Y, Z);
}}
}}
}else{
```

```
//上下左右を描く
sideTop = 1.0;
sideButtom = 0.0;

//top を描く
if(xyz[x][y][z].top==true){
sideTop -= DRAW_LENGTH;
Y=y+1.0-DRAW_LENGTH;
for(X=x; X<x+1-DRAW_LENGTH/100.0; X+=DRAW_LENGTH){
Save2(x ,y, z, X, Y, Z);
}
}

//buttom を描く
if(xyz[x][y][z].buttom==true){
sideButtom += DRAW_LENGTH;
Y=y;
for(X=x; X<x+1-DRAW_LENGTH/10.0; X+=DRAW_LENGTH){
Save2(x ,y, z, X, Y, Z);
}
}

//right を描く
if(xyz[x][y][z].right==true){
X=x+1-DRAW_LENGTH;
for(Y=y+sideButtom; Y<y+sideTop-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
Save2(x ,y, z, X, Y, Z);
}
}

//left を描く
if(xyz[x][y][z].left==true){
X=x;
for(Y=y+sideButtom; Y<y+sideTop-DRAW_LENGTH/10.0; Y+=DRAW_LENGTH){
Save2(x ,y, z, X, Y, Z);
}
}
}
}}}
}

//テスト用の巨大な壁
void testSave2(int x, int y, int z, double X, double Y, double Z){
double xDif, yDif, zDif; //基準と計算する座標との差
double zz;
RGB rgb;

//距離差の計算
xDif = X - CriteriaX;
yDif = Y - CriteriaY;
```

```
zDif = Z - Lorentz(CriteriaZ, BETA);

//新しい z 座標を計算　引数：基準の XYZ, そこからの座標差 xyz、速さ beta の７つ
zz = Deformation(CRITERIA_X, CRITERIA_Y, CRITERIA_Z, xDif, yDif, zDif, BETA);

//ドップラー計算 引数：XYZ 座標差　速度　波長　明るさ　光の合計 ７つ
rgb = DopplerSynthesis(observerX-X, observerY-Y, observerZ-Z, xyz[x][y][z].lambda, xyz[x][y][z].Yi

//描写 前に描いた z 座標に物体があるならそこまでの間を埋める
cube(X, Y, zz, rgb);
}

//テスト用関数　横幅の広い板を設置
//1 個だけの立方体 11  0  3  1  625  1.000000  1
void TESTfunc(){
double X, Y, Z=0.0;
double startX, endX;
startX = CAMERAx - 100.0;
endX = CAMERAx + 100.0;

for(Z=0.0; Z<1.0; Z+=DRAW_LENGTH){
for(X=startX; X<endX; X+=DRAW_LENGTH){
for(Y=0.0; Y<10.0; Y+=DRAW_LENGTH){
testSave2(11, 0, 3, X, Y, Z);
}}}
}

//ディスプレイ関数で基準の座標の検索及び決定を行う
//-------- 各種コールバック関数------------//
void display(void){
int z;
int count=0, i;
double Z, lorentz, lorentzDrawLength;
//float Rf, Gf, Bf;
//RGB TMP;

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glLoadIdentity();
gluLookAt(CameraX, CameraY, CameraZ, GazeX, GazeY, GazeZ, 0.0, 1.0, 0.0);

//光源の位置設定
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

glEnable(GL_COLOR_MATERIAL);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, Shin);

//地面を描く
Ground();
```

```
//test //立方体の描画
if(OB==true){
glColor4f(1.f, 0.f, 0.f, 1.f);
glPushMatrix();
glTranslatef(CAMERAx, CAMERAy, Lorentz(CAMERAz, BETA));
glutSolidCube(0.5);
glPopMatrix();

glColor4f(1.f, 1.f, 0.f, 1.f);
glPushMatrix();
glTranslatef(CAMERAx, CAMERAy, Deformation(CRITERIA_X, CRITERIA_Y, CRITERIA_Z, -CRITERIA_X, -CRITE
//glutSolidCube(0.5);
glPopMatrix();
}

//TESTfunc();

if(FIXATION==false){
observerX=CameraX;
observerY=CameraY;
observerZ=CameraZ;
}else{
observerX=CAMERAx;
observerY=CAMERAy;
observerZ=CAMERAz;
}
CriteriaX=observerX+CRITERIA_X;
CriteriaY=observerY+CRITERIA_Y;
CriteriaZ=observerZ+CRITERIA_Z;


count = 0;

//Z 軸方向にはローレンツ収縮が発生する
lorentz = Lorentz(1.0, BETA);
lorentzDrawLength = lorentz * DRAW_LENGTH;

//Z 軸負がデフォルトの向きとする　Vector が正なら負方向に進む
//count が奇数なら Save 1 に保存する関数、偶数なら Save2 に保存する関数を呼び出す
for(z=0; z<=MAXz-1; z++){
for(i=0; i<LengthPer; i++){
Z = count * lorentzDrawLength;
if(count%2==0){
clearSave2();
XYdrawSave2(z, Z, count);
}else{
clearSave1();
XYdrawSave1(z, Z, count);
}
count++;
```

```
}
}

glDisable(GL_COLOR_MATERIAL);

glutSwapBuffers();
}

void reshape(int w, int h){
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

gluPerspective(30.0, double(w)/h, 1.0, 1000.0);
glMatrixMode(GL_MODELVIEW);
}

void idle(void){
glutPostRedisplay();
}

//-------- その他各種設定-----------------//
void otherInit(void){
glClearColor(1.f, 1.f, 1.f, 1.f);
glClearDepth(1.f);
glEnable(GL_DEPTH_TEST);

//光源設定 (環境拡散鏡面のみ)
glLightfv(GL_LIGHT0, GL_AMBIENT,  lightAmb);
glLightfv(GL_LIGHT0, GL_DIFFUSE,  lightDiff);
glLightfv(GL_LIGHT0, GL_SPECULAR, lightSpec);

//光源とライティング有効化
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);

//法線ベクトルの正規化
glEnable(GL_NORMALIZE);
}

//キーボード処理 (押したとき)
void keyboard(unsigned char key, int x, int y){
double height;

//[ESC] キーのとき
if(key == 27){
exit(0);
}

switch(key){
```

```
case 'a': //左回転
thetaXZ = thetaXZ + TurnoverRate;
if(thetaXZ>M_PI) thetaXZ = thetaXZ - M_PI*2;
GazeX = CameraX + (CGlength * sin(thetaXZ));
GazeZ = CameraZ + (CGlength * cos(thetaXZ));
break;
case 'd': //右回転
thetaXZ = thetaXZ - TurnoverRate;
if(thetaXZ<-1*M_PI) thetaXZ = thetaXZ + M_PI*2;
GazeX = CameraX + (CGlength * sin(thetaXZ));
GazeZ = CameraZ + (CGlength * cos(thetaXZ));
break;
case 'w': //少し上を向く
GazeY++;
break;
case 'x': //少し下を向く
GazeY--;
break;
case '4': //90 °左回転
thetaXZ = thetaXZ + M_PI/2.0;
if(thetaXZ>M_PI) thetaXZ = thetaXZ - M_PI*2;
GazeX = CameraX + (CGlength * sin(thetaXZ));
GazeZ = CameraZ + (CGlength * cos(thetaXZ));
break;
case '6': //90 °右回転
thetaXZ = thetaXZ - M_PI/2.0;
if(thetaXZ<-1*M_PI) thetaXZ = thetaXZ + M_PI*2;
GazeX = CameraX + (CGlength * sin(thetaXZ));
GazeZ = CameraZ + (CGlength * cos(thetaXZ));
break;
case '2': //後ろを向く
thetaXZ = thetaXZ - M_PI;
if(thetaXZ<-1*M_PI) thetaXZ = thetaXZ + M_PI*2;
GazeX = CameraX + (CGlength * sin(thetaXZ));
GazeZ = CameraZ + (CGlength * cos(thetaXZ));
break;
case 'f': //向く高さを正面にする
GazeY = CameraY;
break;
case 'r': //初期値の方向を向く
GazeX=CameraX;
GazeY=CameraY;
GazeZ=CameraZ-CGlength;
thetaXZ=acos((GAZEz-CAMERAz)/CGlength);
break;
case 'g': //カメラの位置を高さ 1 にする
height = 1.0 - CameraY;
CameraY = 1.0;
GazeY += height;
break;
```

```
case '8': //カメラの向きをそのままで運動方向へ移動
CameraZ--;
GazeZ--;
break;
case '9': //カメラの向きをそのままで後方へ移動
CameraZ++;
GazeZ++;
break;
}
}

//特殊キーを押したとき
void specialKey(int key, int x, int y){
double theta;

switch(key){
case GLUT_KEY_UP: //前進
theta = thetaXZ;
CameraX += Stride * sin(theta);
GazeX   += Stride * sin(theta);
CameraZ += Stride * cos(theta);
GazeZ   += Stride * cos(theta);
break;
case GLUT_KEY_DOWN: //後進
theta = thetaXZ - M_PI;
if(theta>-M_PI) theta = theta + M_PI*2;
CameraX += Stride * sin(theta);
GazeX   += Stride * sin(theta);
CameraZ += Stride * cos(theta);
GazeZ   += Stride * cos(theta);
break;
case GLUT_KEY_LEFT: //左に平行移動
theta = thetaXZ + (M_PI/2.0);
if(theta>M_PI) theta = theta - M_PI*2;
CameraX += Stride * sin(theta);
GazeX   += Stride * sin(theta);
CameraZ += Stride * cos(theta);
GazeZ   += Stride * cos(theta);
break;
case GLUT_KEY_RIGHT: //右に平行移動
theta = thetaXZ - (M_PI/2.0);
if(theta>-M_PI) theta = theta + M_PI*2;
CameraX += Stride * sin(theta);
GazeX   += Stride * sin(theta);
CameraZ += Stride * cos(theta);
GazeZ   += Stride * cos(theta);
break;
case GLUT_KEY_PAGE_UP: //上昇
CameraY++;
GazeY++;
```

```
break;
case GLUT_KEY_PAGE_DOWN: //下降
CameraY--;
GazeY--;
break;
case GLUT_KEY_HOME: //カメラ座標の初期化
CameraX=CAMERAx;
CameraY=CAMERAy;
GazeX = CameraX;
GazeY = CameraY;
CameraZ=CAMERAz;
GazeZ = CameraZ-CGlength;
break;
case GLUT_KEY_END: //test 用　50 マス前進
theta = thetaXZ;
CameraX += Stride*50.0 * sin(theta);
GazeX   += Stride*50.0 * sin(theta);
CameraZ += Stride*50.0 * cos(theta);
GazeZ   += Stride*50.0 * cos(theta);
break;
}
}

//面についての 6 変数の最適化
void surfaceOptimization(int x, int y, int z){
//X 軸負方向の面を描く必要があるか
if(x==0){
xyz[x][y][z].left = true;
}else{
if(xyz[x-1][y][z].object==false)
xyz[x][y][z].left = true;
else
xyz[x][y][z].left = false;
}

//X 軸正
if(x==MAXx-1){
xyz[x][y][z].right = true;
}else{
if(xyz[x+1][y][z].object==false)
xyz[x][y][z].right = true;
else
xyz[x][y][z].right = false;
}

//Y 軸負
if(y==0){
xyz[x][y][z].buttom = true;
}else{
if(xyz[x][y-1][z].object==false)
```

```
xyz[x][y][z].buttom = true;
else
xyz[x][y][z].buttom = false;
}

//Y 軸正
if(y==MAXy-1){
xyz[x][y][z].top = true;
}else{
if(xyz[x][y+1][z].object==false)
xyz[x][y][z].top = true;
else
xyz[x][y][z].top = false;
}

//Z 軸負
if(z==0){
xyz[x][y][z].front = true;
}else{
if(xyz[x][y][z-1].object==false)
xyz[x][y][z].front = true;
else
xyz[x][y][z].front = false;
}

//Z 軸正
if(z==MAXz-1){
xyz[x][y][z].back = true;
}else{
if(xyz[x][y][z+1].object==false)
xyz[x][y][z].back = true;
else
xyz[x][y][z].back = false;
}
}

//メモリ節約のための最適化
void Optimization(){
int x, y, z;

for(x=0; x<MAXx; x++){
for(y=0; y<MAXy; y++){
for(z=0; z<MAXz; z++){
if(xyz[x][y][z].object == true){
xyz[x][y][z].visibility = false;

if(x==0 || x==MAXx-1 || y==0 || y==MAXy-1 || z==0 || z==MAXz-1)
xyz[x][y][z].visibility = true;
else{if(!(xyz[x-1][y-1][z-1].object==true && xyz[x-1][y-1][z].object==true && xyz[x-1][y-1][z+1].o
  xyz[x-1][y][z-1].object  ==true && xyz[x-1][y][z].object  ==true && xyz[x-1][y][z+1].object  ==t
```

```
  xyz[x-1][y+1][z-1].object==true && xyz[x-1][y+1][z].object==true && xyz[x-1][y+1][z+1].object==t
  xyz[x][y-1][z-1].object  ==true && xyz[x][y-1][z].object  ==true && xyz[x][y-1][z+1].object  ==t
  xyz[x][y][z-1].object    ==true    && xyz[x][y][z+1].object    ==true &&
  xyz[x][y+1][z-1].object  ==true && xyz[x][y+1][z].object  ==true && xyz[x][y+1][z+1].object  ==t
  xyz[x+1][y-1][z-1].object==true && xyz[x+1][y-1][z].object==true && xyz[x+1][y-1][z+1].object==t
  xyz[x+1][y][z-1].object  ==true && xyz[x+1][y][z].object  ==true && xyz[x+1][y][z+1].object  ==t
  xyz[x+1][y+1][z-1].object==true && xyz[x+1][y+1][z].object==true && xyz[x+1][y+1][z+1].object==t
)){
//周り全てに物体がなかったので表示する必要有り
xyz[x][y][z].visibility = true;
}}

if(xyz[x][y][z].visibility == true){
surfaceOptimization(x, y, z);
}
}}}}
}

//--------- メイン関数------------//
int main(int argc, char *argv[]){
FILE *file;
int end;
//RGB rgb;

if ((file = fopen(argv[1], "r")) == NULL){
fprintf(stderr, "ファイル%s がありません。\n", argv[1]);
scanf("%d", &end);
exit(1);
    }else{
Architecture(file);
/*最後にファイルを閉じる*/
fclose(file);
printf("読み込みが完了しました。\n\n");

    }


//表示する必要のない物体を割り出す　最適化
Optimization();

glutInit(&argc,argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
glutInitWindowSize(640, 480);
glutCreateWindow("3D Pseudo Shooting");

//コールバック関数登録
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutIdleFunc(idle);
```

```
//その他設定
otherInit();

//通常キーが押された時の処理　離した場合はなし
glutKeyboardFunc(keyboard);
//特殊キーが押された時の処理　離した場合はなし
glutSpecialFunc(specialKey);
//キー・リピート無視
glutIgnoreKeyRepeat(GL_TRUE);

glutMainLoop();

return 0;
}
```

E　3Ddoppler.c のその他関数 (3DdopplerFunction.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <my/myhead.h>
#include <gl/glut.h>

#pragma warning(disable : 4996)

int round1(double str){
int intR;

intR = str+0.5;

return intR;
}


//地面を描く
void Ground(){
glColorMaterial(GL_FRONT_AND_BACK,GL_SPECULAR);
glColor4f(GROUND_SPEC_R, GROUND_SPEC_G, GROUND_SPEC_B, GROUND_SPEC_A);
glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);
glColor4f(GROUND_AandD_R, GROUND_AandD_G, GROUND_AandD_B, GROUND_AandD_A);

glNormal3f(0,1,0);
glBegin(GL_QUADS);
glVertex3f((float)MAX_LENGTH*(-100.f) ,-0.5f ,(float)MAX_DEPTH*(-100.f));
glVertex3f((float)MAX_LENGTH*100.f ,-0.5f ,(float)MAX_DEPTH*(-100.f));
glVertex3f((float)MAX_LENGTH*100.f ,-0.5f ,(float)MAX_DEPTH*100.f);
```

```
glVertex3f((float)MAX_LENGTH*(-100.f) ,-0.5f ,(float)MAX_DEPTH*100.f);
glEnd();
}

//RGB から HSV へ変換し、HSV を返す
HSV RGBtoHSV(RGB rgb){
int min, max;
HSV hsv;

if(rgb.R >= rgb.G){
if(rgb.R >= rgb.B)
max = rgb.R;
else
max = rgb.B;

if(rgb.G <= rgb.B)
min = rgb.G;
else
min = rgb.B;
}else{
if(rgb.G >= rgb.B)
max = rgb.G;
else
max = rgb.B;

if(rgb.R <= rgb.B)
min = rgb.R;
else
min = rgb.B;
}

hsv.V = max;

//max が 0 なら黒色確定
if(hsv.V == 0.0){
hsv.S=0.0;
hsv.H=0.0;

return hsv;
}

hsv.S = 255.0 * (max-min)/max;

//S=0 なら H は計算させない
if(hsv.S == 0){
hsv.H = 0;

return hsv;
}
```

```
//場合分け
if(rgb.R >= rgb.G){
if(rgb.R >= rgb.B)
hsv.H = 60.0 * (double)(rgb.G-rgb.B)/(max-min);
else
hsv.H = 60.0 * (4.0+(double)(rgb.R-rgb.G)/(max-min));
}else{
if(rgb.G >= rgb.B)
hsv.H = 60.0 * (2.0+(double)(rgb.B-rgb.R)/(max-min));
else
hsv.H = 60.0 * (4.0+(double)(rgb.R-rgb.G)/(max-min));
}

//Hを 0〜359 の範囲に収める
while(!(0.0 <= hsv.H && hsv.H < 360.0)){
hsv.H = hsv.H - (360.0 * (hsv.H/abs(hsv.H)));
}

return hsv;
}

//HSV から RGB へ変換し、RGB を返す
RGB HSVtoRGB(HSV hsv){
RGB rgb;
int i;
double F, M, N, K, V;


if(hsv.S==0){
rgb.R = hsv.V;
rgb.G = hsv.V;
rgb.B = hsv.V;

return rgb;
}

i = (int)floor(hsv.H/60.0);
F = hsv.H/60.0-i;

M = (hsv.V * (1.0 -           hsv.S/255.0));
N = (hsv.V * (1.0 -      F  * hsv.S/255.0));
K = (hsv.V * (1.0 - (1.0-F) * hsv.S/255.0));

V = hsv.V;

//printf("%d \n", i);

switch(i){
case 0:
rgb.R = V;
```

```
rgb.G = K;
rgb.B = M;
break;
case 1:
rgb.R = N;
rgb.G = V;
rgb.B = M;
break;
case 2:
rgb.R = M;
rgb.G = V;
rgb.B = K;
break;
case 3:
rgb.R = M;
rgb.G = N;
rgb.B = V;
break;
case 4:
rgb.R = K;
rgb.G = M;
rgb.B = V;
break;
case 5:
rgb.R = V;
rgb.G = M;
rgb.B = N;
break;
}

return rgb;
}

//------------------------------↓波長からRGB取得↓------------------------------
//波長から red の値を返す H0 比較用 H 計算用
int redJudg(int H0){
double reS=0;
double H = (double)H0;
int x;

if(LIMIT_RED>H && H>RED){

reS = (double)(LIMIT_RED - RED);
x = (int)(255.0 * ((-H + (double)LIMIT_RED)/reS));

}else{ if(RED>=H && H>=YELLOW){

x = 255;

}else{ if(YELLOW>H0 && H0>GREEN){
```

```
reS = (double)(YELLOW - GREEN);
x = (int)(255.0 - (255.0 * ((-H + (double)YELLOW)/reS)));

}else{ if(BLUE>H0 && H0>=PURPLE){

reS = (double)(BLUE - PURPLE);
x = (int)(255.0 * ((-H + (double)BLUE)/reS));

}else{ if(PURPLE>H0 && H0>LIMIT_PURPLE){

reS = (double)(PURPLE - LIMIT_PURPLE);
x = (int)(255.0 - (255.0 * ((-H + (double)PURPLE)/reS)));

}else{

x = 0;

}}}}}

return x;
}

//波長から green の値を返す
int greenJudg(int H0){
double grS = 0;
double H = (double)H0;
int x;

if(RED>=H0 && H0>YELLOW){

grS = (double)(RED - YELLOW);
x = (int)(255 * ((-H + (double)RED)/grS));

}else{ if(YELLOW>=H0 && H0>CYAN){

x = 255;

}else{ if(CYAN>=H0 && H0>BLUE){

grS = (double)(CYAN - BLUE);
x = (int)(255 - (255 * ((-H +(double)CYAN)/grS)));

}else{

x = 0;

}}}

return x;
```

```
}

//波長から blue の値を返す
int blueJudg(int H0){
double blS = 0;
double H = (double)H0;
int x;

if(GREEN>=H0 && H0>=CYAN){

blS = (double)(GREEN - CYAN);
x = (int)(255 * ((-H + (double)GREEN)/blS));

}else{ if(CYAN>H0 && H0>=PURPLE){

x = 255;

}else{ if(PURPLE>H0 && H0>LIMIT_PURPLE){

blS = (double)(PURPLE - LIMIT_PURPLE);
x = (int)(255 - (255 * ((-H +(double)PURPLE)/blS)));

}else{

x = 0;

}}}

return x;
}
//-------------------------------↑波長から RGB 取得↑-------------------------------

//本来ならここで複数の波長による色の合成を行う
//現状では 1 つだけの波長から RGB に変換し、明るさを調節するだけ
RGB DopplerSynthesis(double X, double Y, double Z, int lambda[], double Yi[], int sum){
RGB rgb;
HSV hsv;
int newWavelength;
double theta;

//警告回避用の無駄な計算
rgb.R = sum;

theta = acos(Z / sqrt(X*X + Y*Y + Z*Z));
if(DOPPLER==true)
newWavelength = (double)lambda[0] * (1.0-(BETA*cos(theta))) / sqrt(1.0-(BETA*BETA));
else
newWavelength = (double)lambda[0];
```

```
//波長から RGB へ変換
rgb.R = redJudg(newWavelength);
rgb.G = greenJudg(newWavelength);
rgb.B = blueJudg(newWavelength);


hsv = RGBtoHSV(rgb);

hsv.V = hsv.V * Yi[0];

rgb = HSVtoRGB(hsv);

rgb.R = rgb.R/255.0;
rgb.G = rgb.G/255.0;
rgb.B = rgb.B/255.0;

return rgb;
}


double Lorentz(double l, double vc){

l = l * sqrt(1.0-(vc*vc));

return l;
}


//テレルの関数名：Deformation 変形　物体の変形　新しい z 座標を返す
double Deformation(double LL, double Y, double zC, double a, double h, double b, double vc){
double zc = zC, L=LL, y=Y;
double vt;

vt = ( sqrt(L*L+y*y+zc*zc)+vc*(zc+b)
- sqrt(
pow(sqrt(L*L+y*y+zc*zc)+vc*(zc+b), 2.0)
- (1.0-vc*vc)*((L*L+y*y+zc*zc)-((L+a)*(L+a)+(y+h)*(y+h)+(zc+b)*(zc+b)))
   )
 ) * (vc/(1.0-vc*vc));

return zc+b+vt;
}
```