

ゲームグラフィックス特論

第6回 陰影付け

陰影付けモデル

光と物体表面との相互作用のモデル化

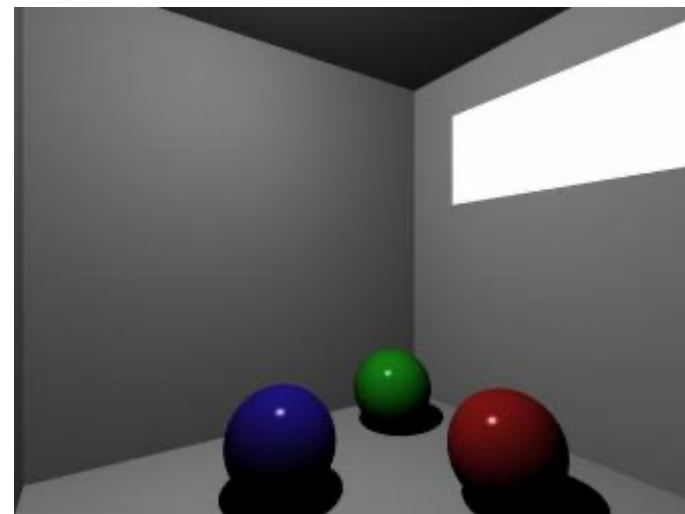
光源と光の知覚

- 光源から放射した光（光子, photon）が
 - 直接目に届く⇒**直接光**
 - 物体表面で反射して目に届く⇒**反射光**
- **反射光**を知覚することで形が知覚できる

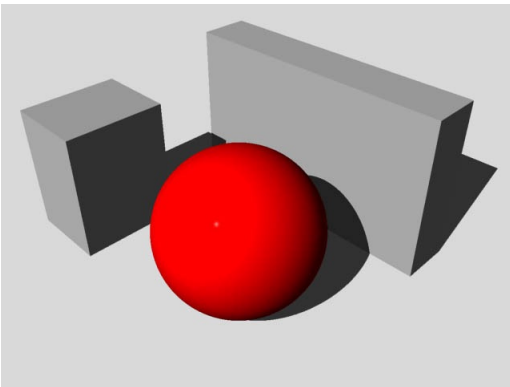
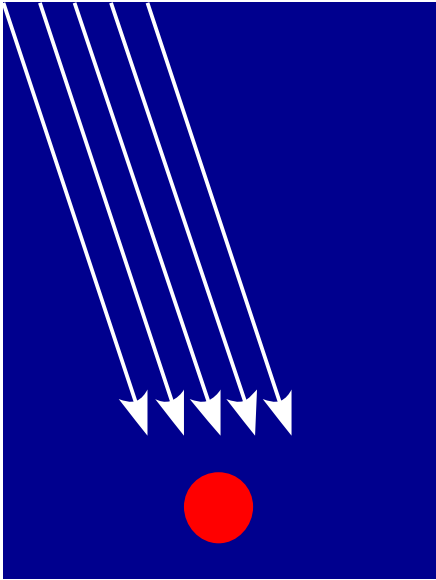
反射光がないと形がわからない



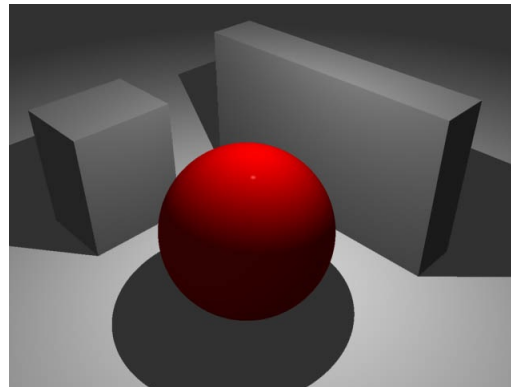
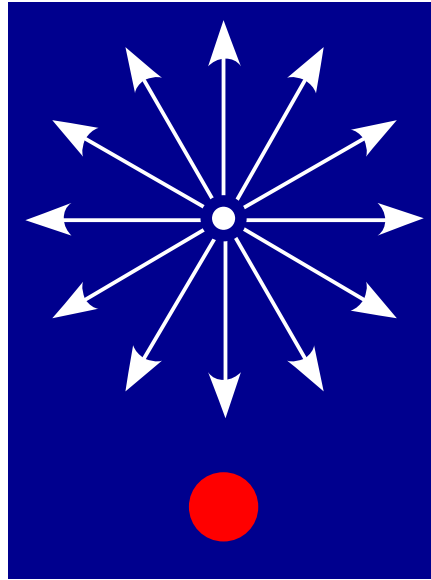
反射光によって形を知覚できる



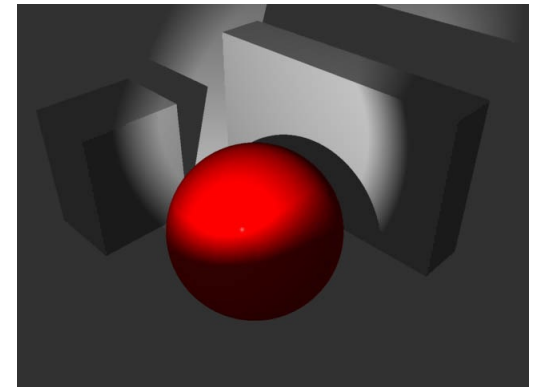
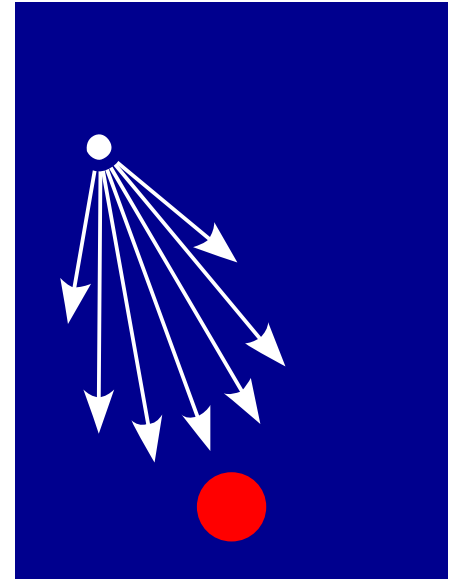
CG で使われる基本的な光源



平行光線



点光源



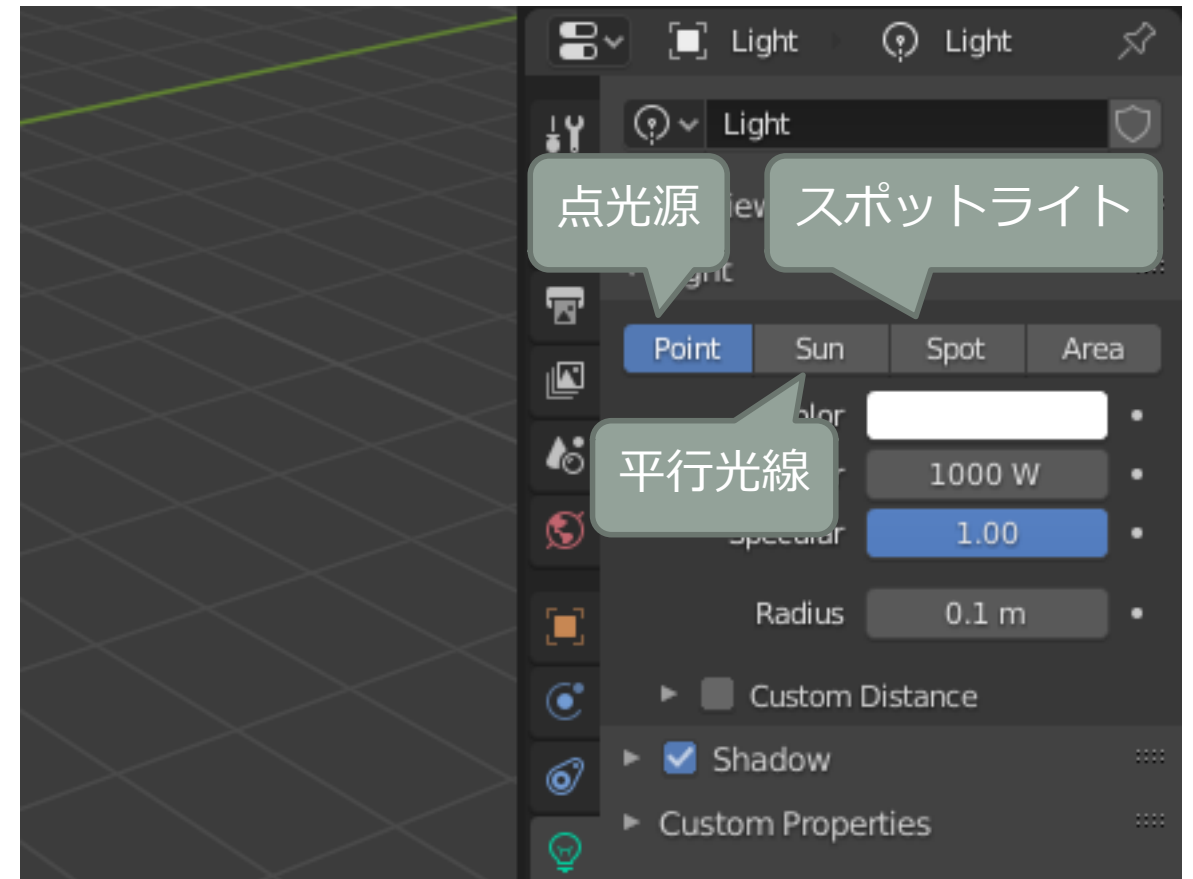
スポットライト

CG ソフトのユーザインタフェース

Maya 2019



Blender 2.83

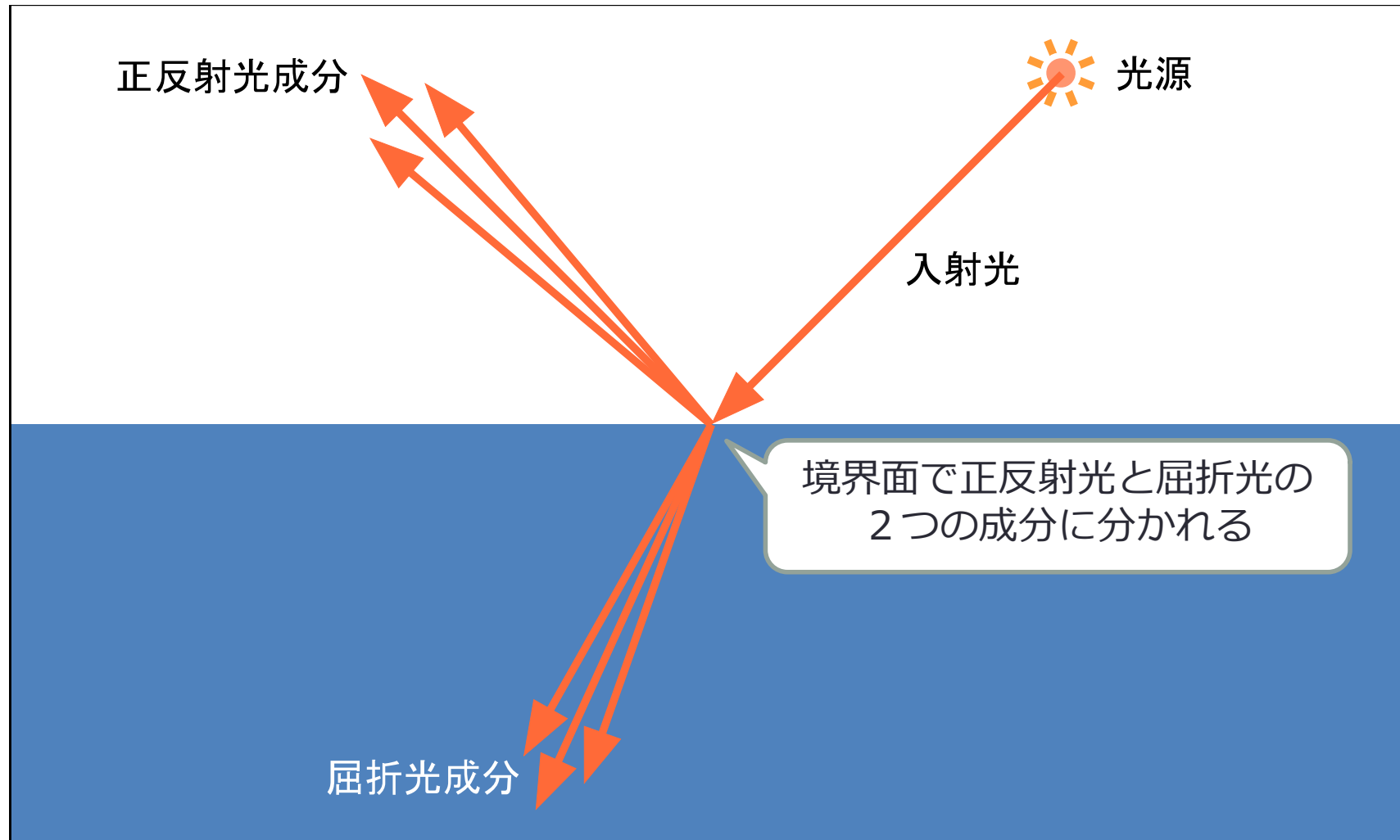


全方向光源 (Omni Lights)

- この3種類の光源の形状は全て「点」
 - 平行光線
 - 光源が無限の彼方にあるために放射方向が揃っていて大きさが**無限に小さく**なっている
 - 点光源
 - **全ての方向**に均等に光を放射する
 - スポットライト
 - **特定の方向**に光を強く放射する
- 全方向光源 (Omni Light)
 - 全方向に均等に光を放射する光源
 - 点光源の上位概念

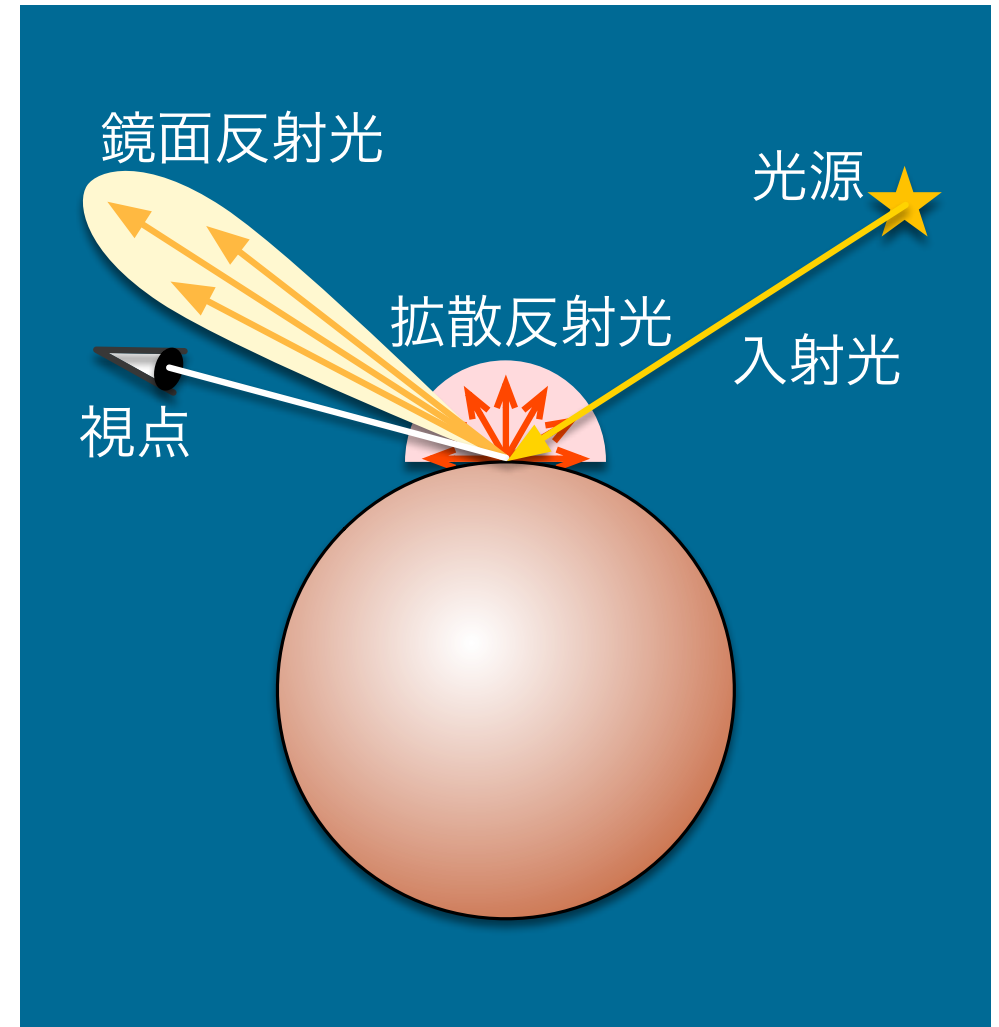
すなわち点

反射と屈折

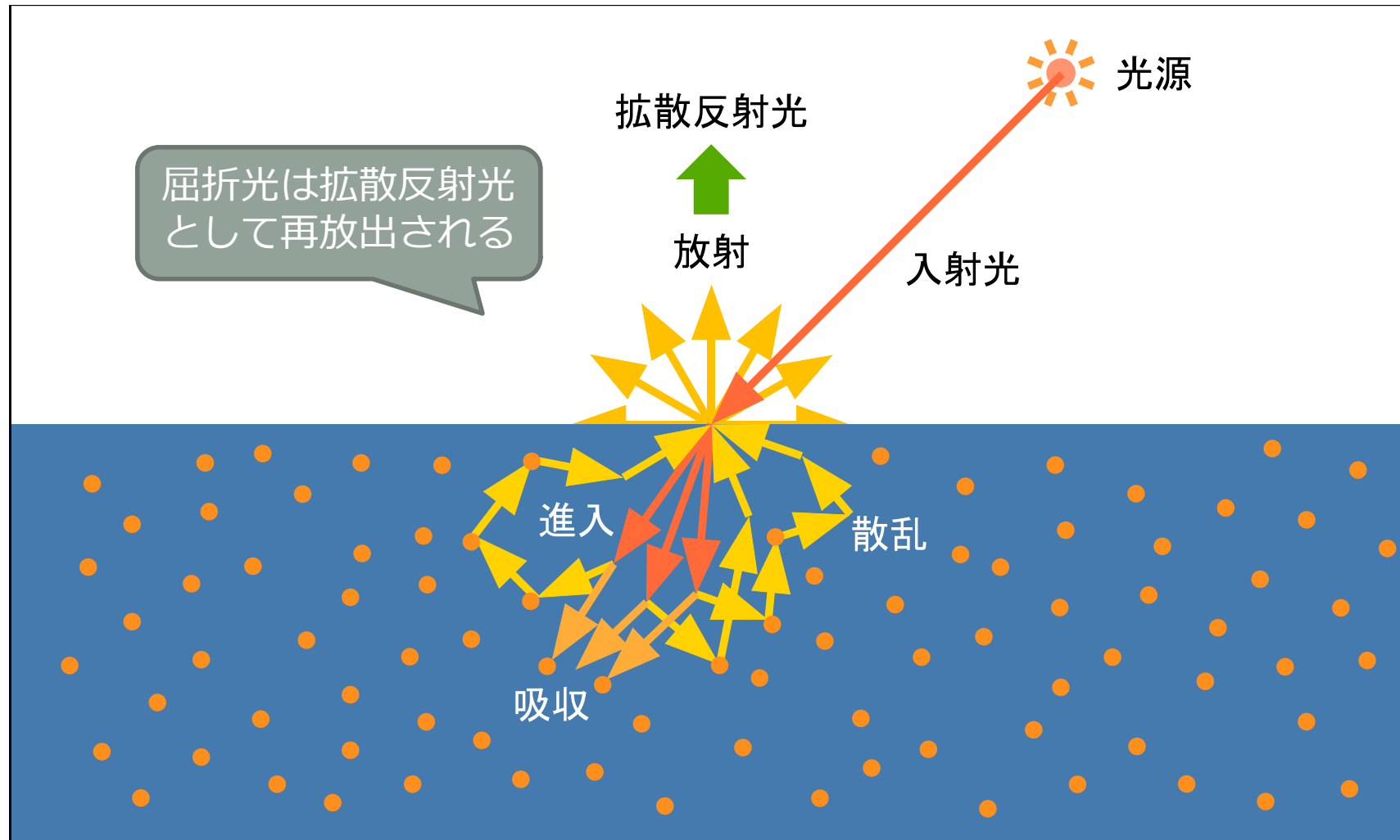


二色性反射モデル

- 反射光は2つの成分で構成される
- **拡散反射光**
 - 指向性を持たない
 - 全ての方向に均等に放射される
- **鏡面反射光**
 - 指向性を持つ
 - 正反射方向を中心に放射される



拡散反射光



拡散反射光の意味

- 入射光は屈折して物体内に進入し**散乱**と**吸収**を繰り返す
- 吸収されずに残った光が**入射点から**再び外部に**放射**される
 - **散乱**により指向性を失っている
 - すべての方向に対して均等に放射される（完全拡散反射面）
 - 反射光強度は視線の方向に依存しない
 - **吸収**により物体の色がついている
 - 入射光の色成分のうち吸収されなかったものが放射される

散乱と吸収

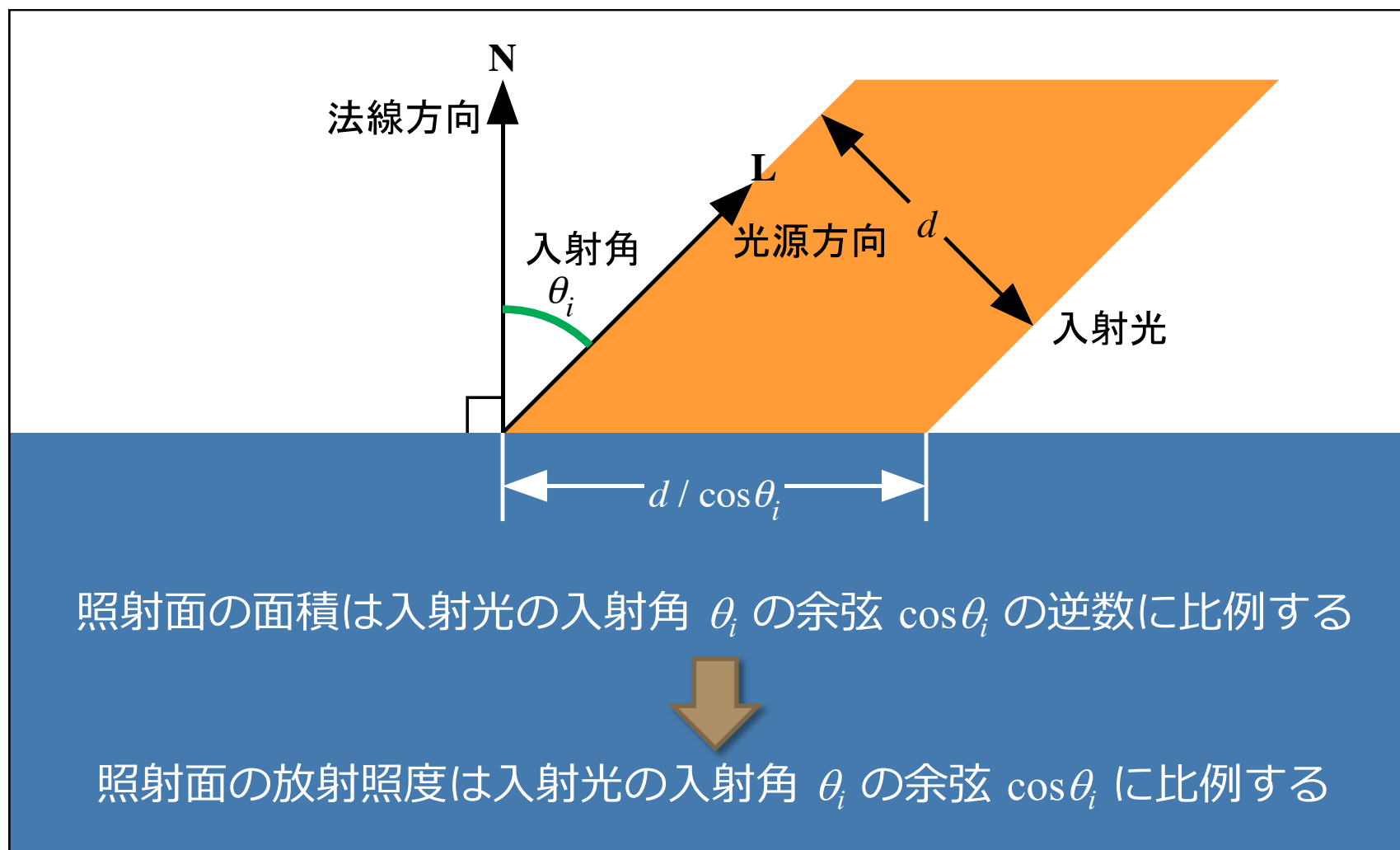
- **散乱**

- 光が光学的な**不連続性**に出会うことで発生する
- 二つの異なる光学特性を持つ材質の境界面
- 光は向きを変えるだけで光の量は変化しない

- **吸収**

- 物質の内部で発生する
- 光が他の種類のエネルギーに変換される（光が消失する）

Lambert の余弦法則



拡散反射光強度

- I_{diff} : 拡散反射光強度
- K_{diff} : 材質の拡散反射係数
- L_{diff} : 光源強度の拡散反射光成分
- \otimes : 要素ごとの積

$I_{diff}, K_{diff}, L_{diff}$
は RGB の三つ
の要素を持つ

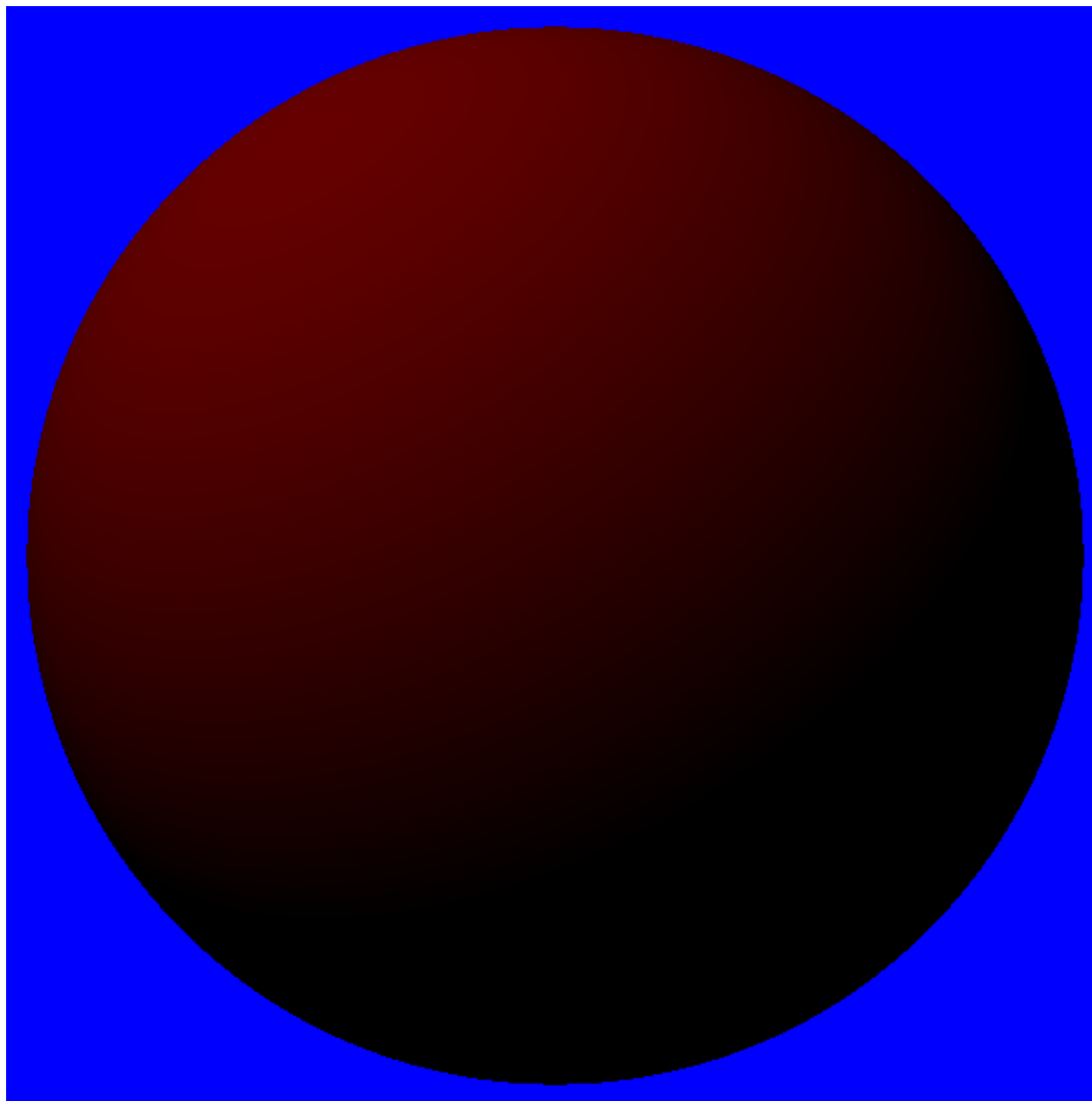
実際の光源にそんな成分がある訳ではない

$$I_{diff} = \cos \theta_i K_{diff} \otimes L_{diff} = (\mathbf{N} \cdot \mathbf{L}) K_{diff} \otimes L_{diff}$$

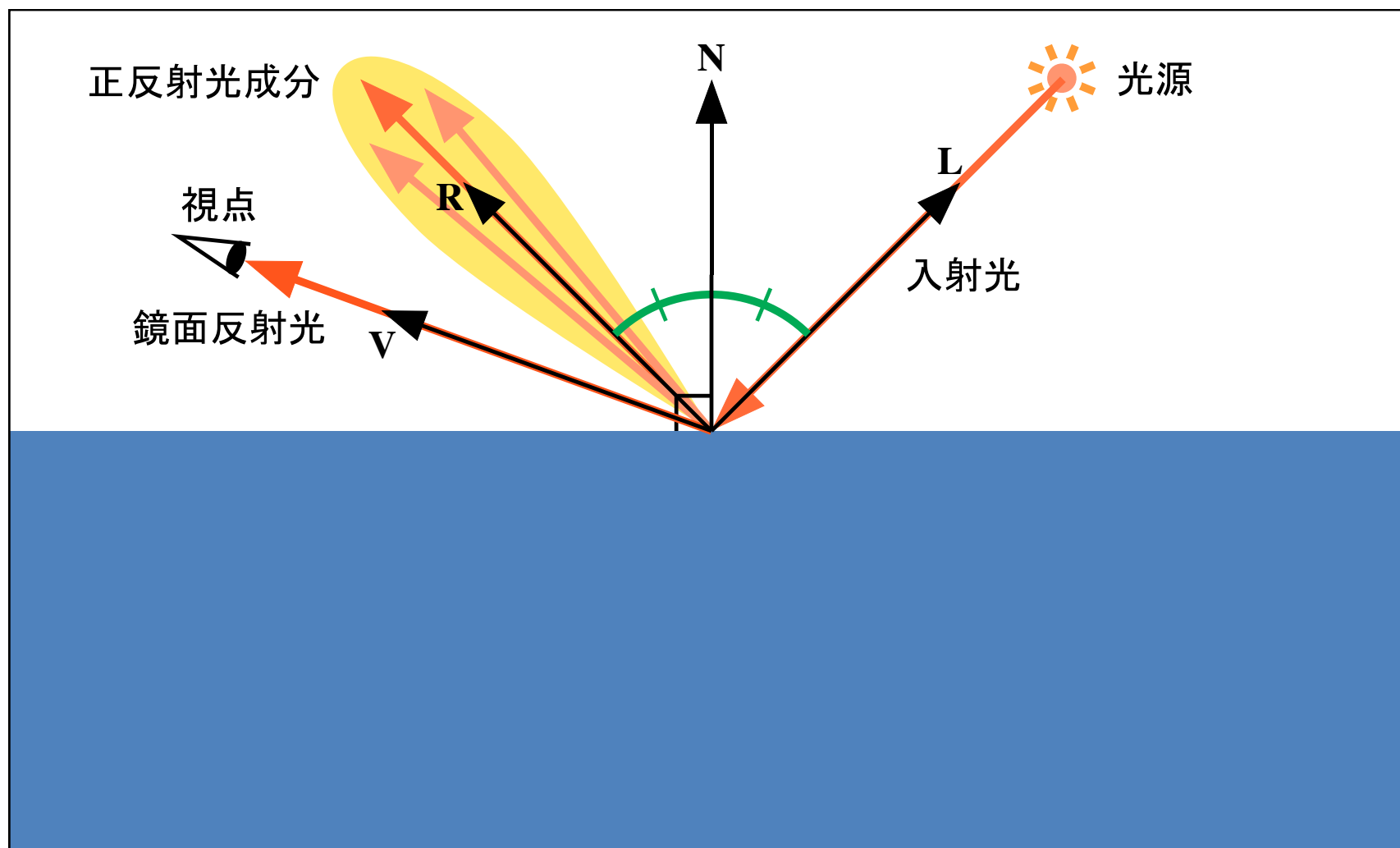
- \mathbf{N} と \mathbf{L} のなす角が $\pi / 2$ 以上なら拡散反射光強度は 0:

$$I_{diff} = \max(\mathbf{N} \cdot \mathbf{L}, 0) K_{diff} \otimes L_{diff}$$

拡散反射光による
陰影



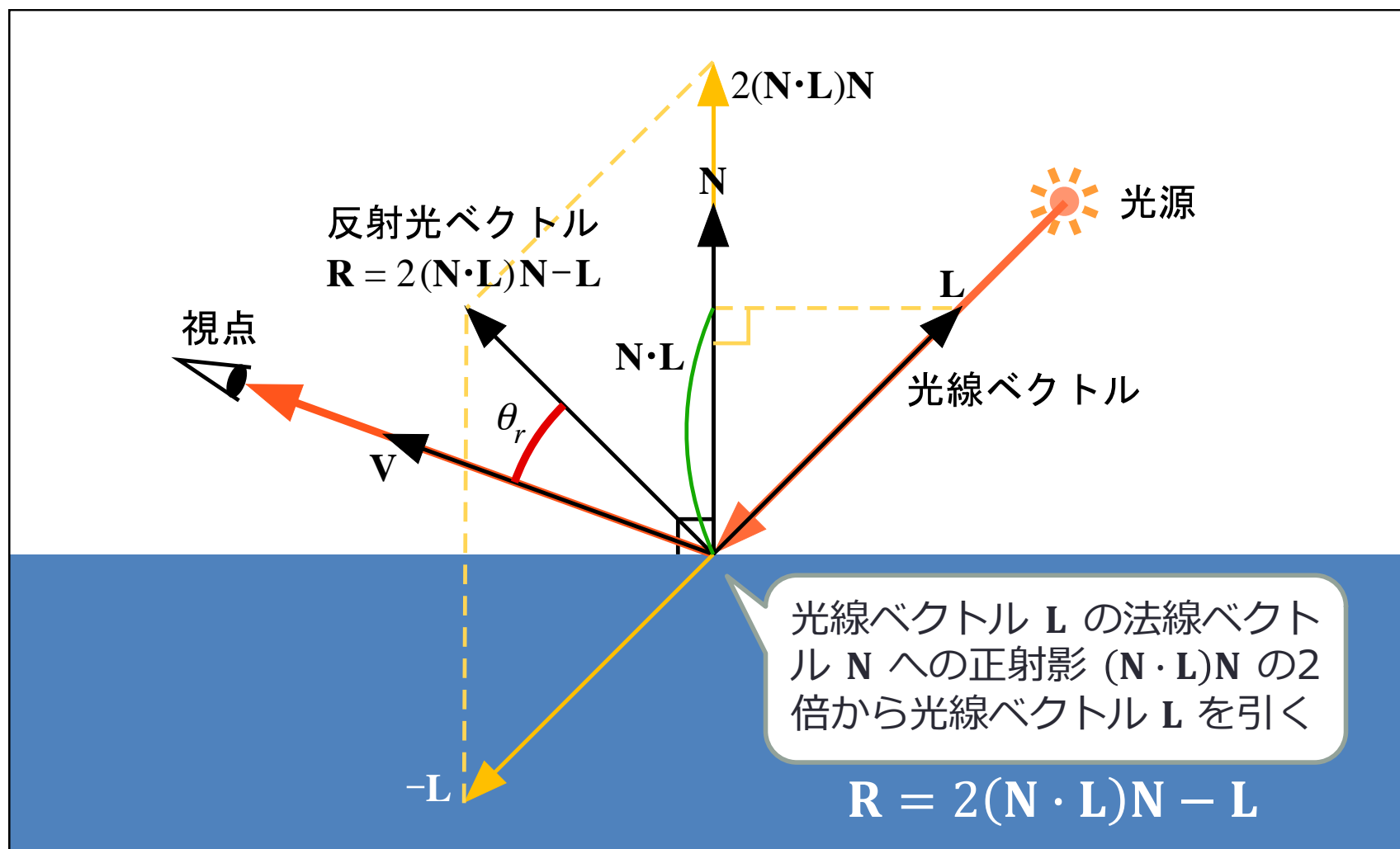
鏡面反射光



鏡面反射光の意味

- 入射光の正反射光のうち**視点方向に到達する成分**
 - 正反射光は正反射方向を中心に分布する
 - そう仮定しないと「見えない」
- 入射光の正反射光
 - 物体の内部には進入しない
 - 物体の色は付かない⇒光源の色がそのまま反映される
 - 物体表面の滑らかさの影響を受ける
- 物体表面上に**ハイライト**を生成して物体を輝かせて見せる

反射光ベクトルの求め方



鏡面反射光強度 (Phong のモデル)

- I_{spec} : 鏡面反射光強度
- K_{spec} : 材質の鏡面反射係数
- L_{spec} : 光源強度の鏡面反射光成分
- \otimes : 要素ごとの積

$I_{spec}, K_{spec}, L_{spec}$
は RGB の三つ
の要素を持つ

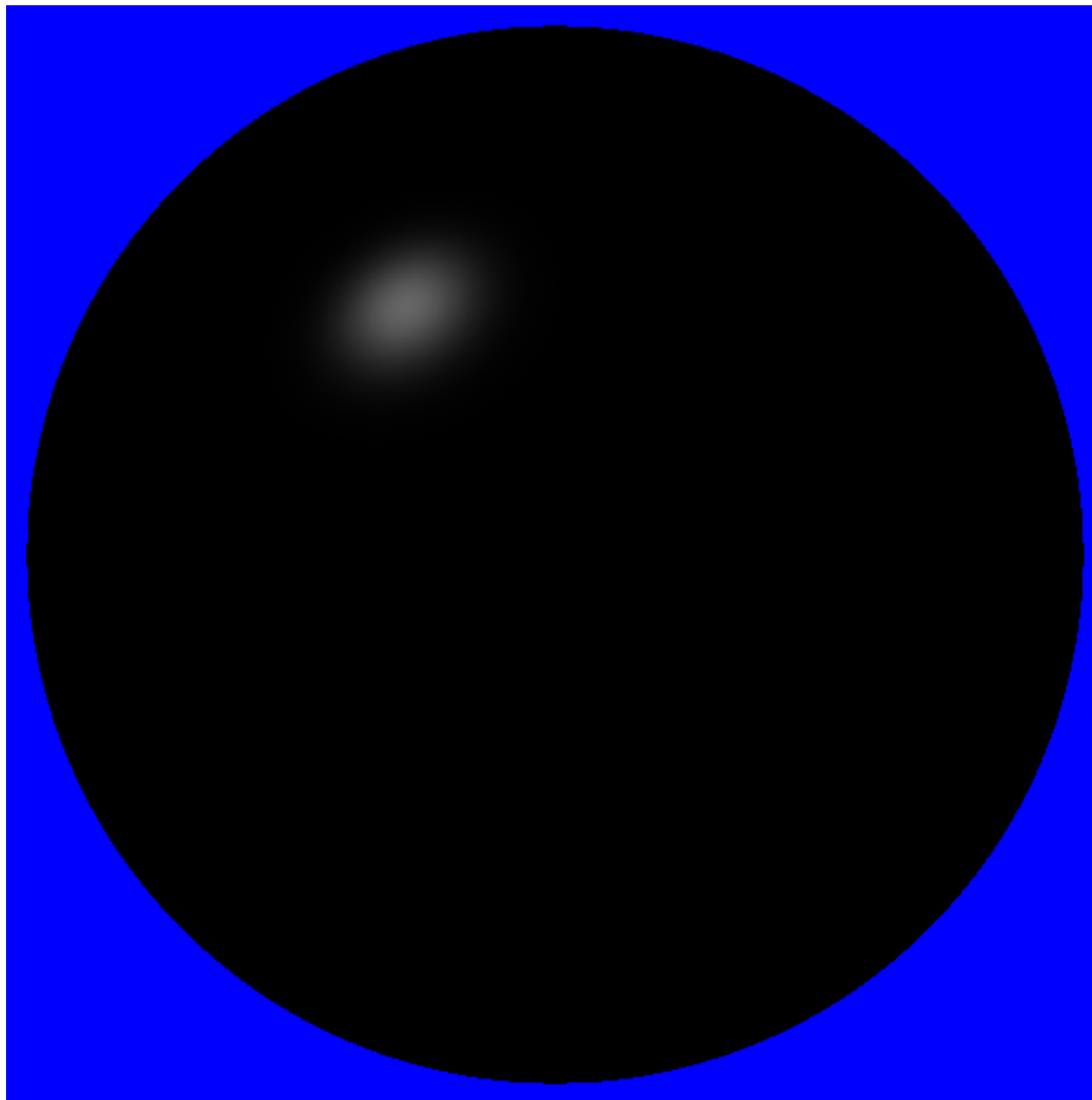
実際の光源にそんな成分がある訳ではない

$$I_{spec} = \cos^{K_{shi}} \theta_r K_{spec} \otimes L_{spec} = (\mathbf{R} \cdot \mathbf{V})^{K_{shi}} K_{spec} \otimes L_{spec}$$

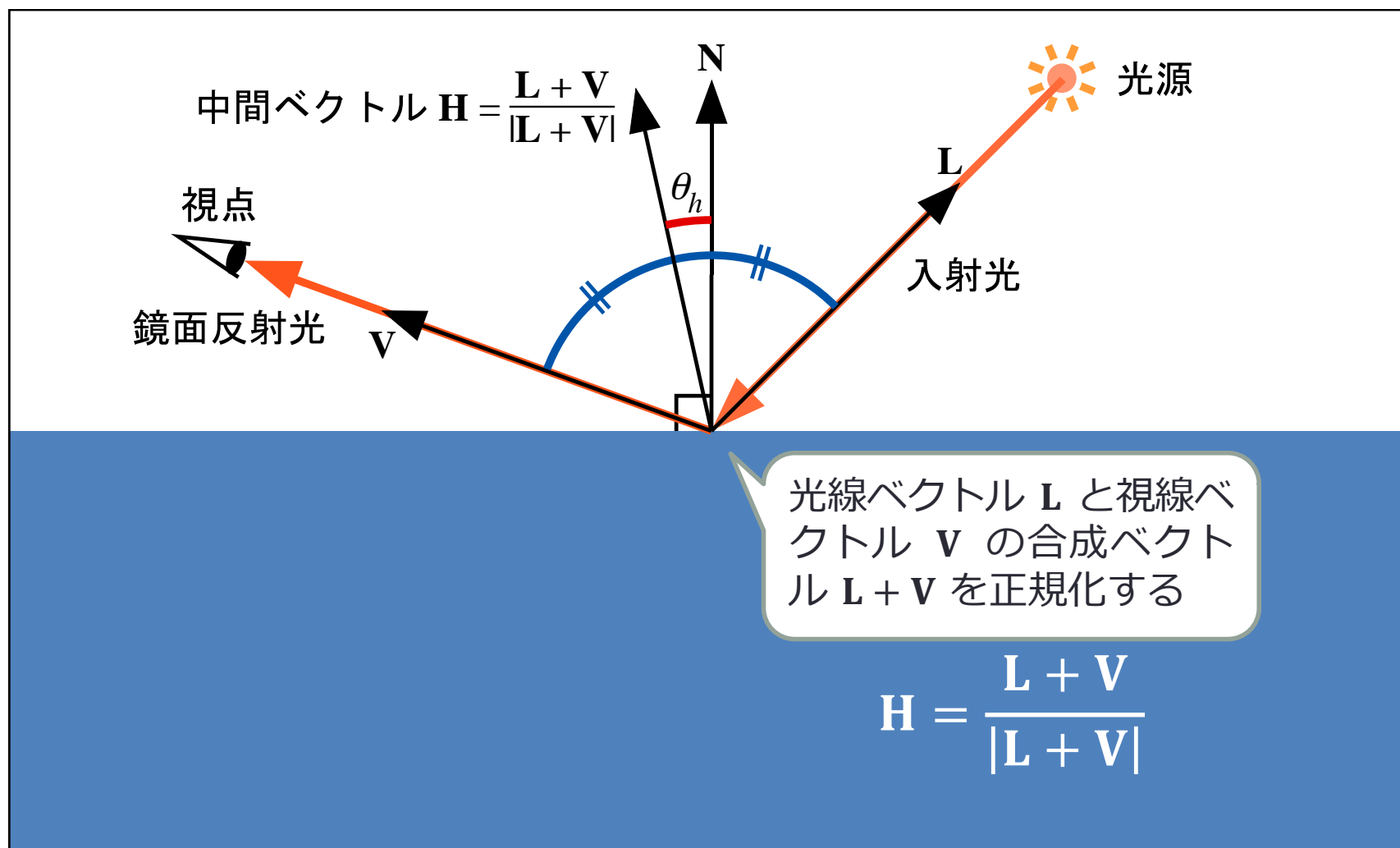
- \mathbf{R} と \mathbf{V} のなす角が $\pi / 2$ 以上なら拡散反射光強度は 0:

$$I_{spec} = \max(\mathbf{R} \cdot \mathbf{V}, 0)^{K_{shi}} K_{spec} \otimes L_{spec}$$

鏡面反射光による陰
影 (Phong)



正反射ベクトルの代わりに中間ベクトルを用いる



鏡面反射光強度 (Blinn のモデル)

- I_{spec} : 鏡面反射光強度
- K_{spec} : 材質の鏡面反射係数
- L_{spec} : 光源強度の鏡面反射光成分
- \otimes : 要素ごとの積

lobe (反射光分布の包絡形状) には Phong と同じ cosine モデルを使う

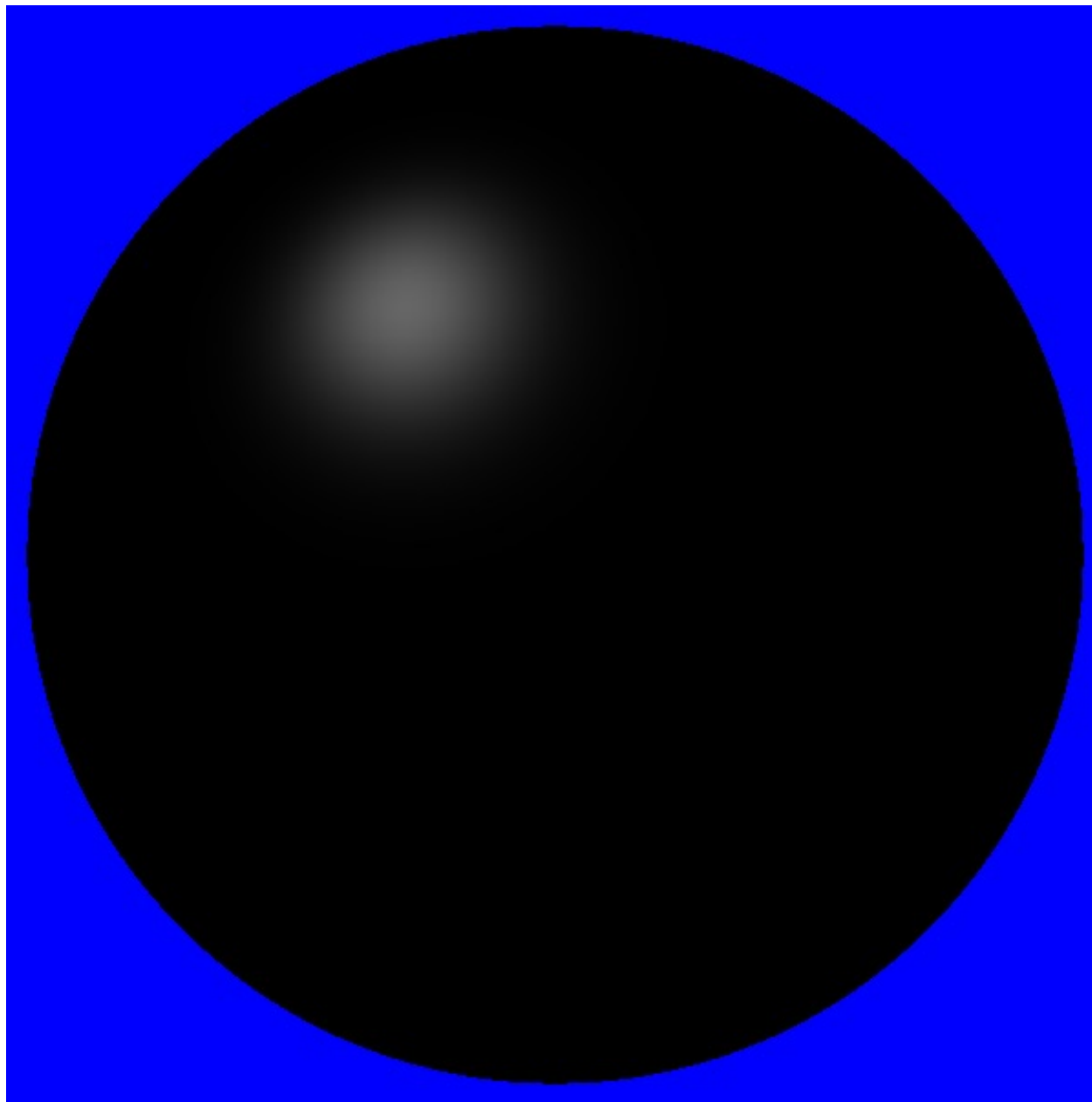
$$I_{spec} = \cos^{K_{shi}} \theta_h K_{spec} \otimes L_{spec} = (\mathbf{N} \cdot \mathbf{H})^{K_{shi}} K_{spec} \otimes L_{spec}$$

- \mathbf{N} と \mathbf{H} のなす角が $\pi / 2$ 以上なら拡散反射光強度は 0:

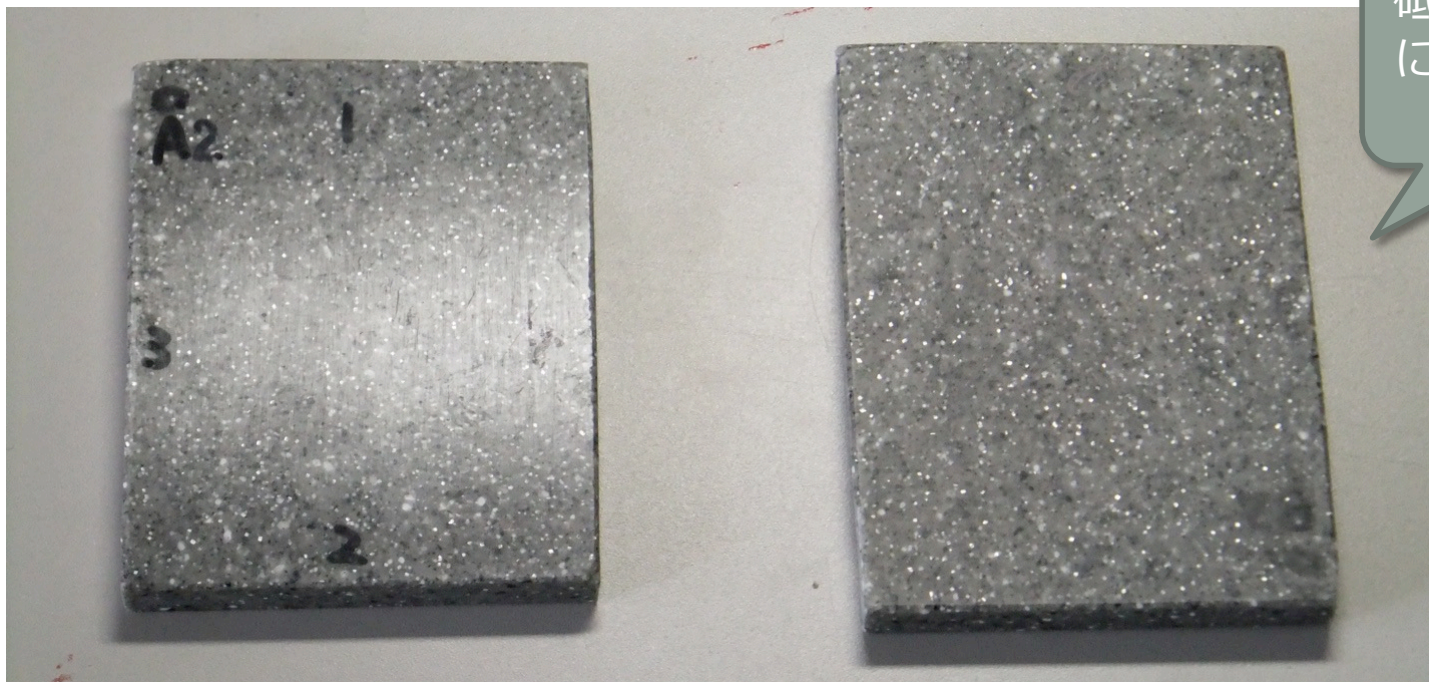
$$I_{spec} = \max(\mathbf{N} \cdot \mathbf{H}, 0)^{K_{shi}} K_{spec} \otimes L_{spec}$$

(OpenGL の固定機能ハードウェアで採用されていたモデル, **Blinn-Phong** モデル)

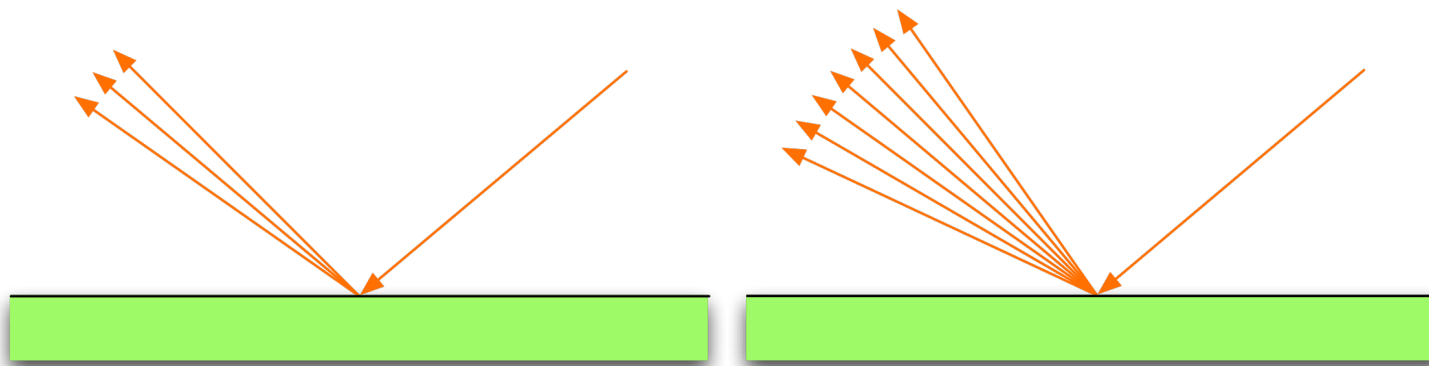
鏡面反射光による陰
影 (Blinn)



物体表面の滑らかさ

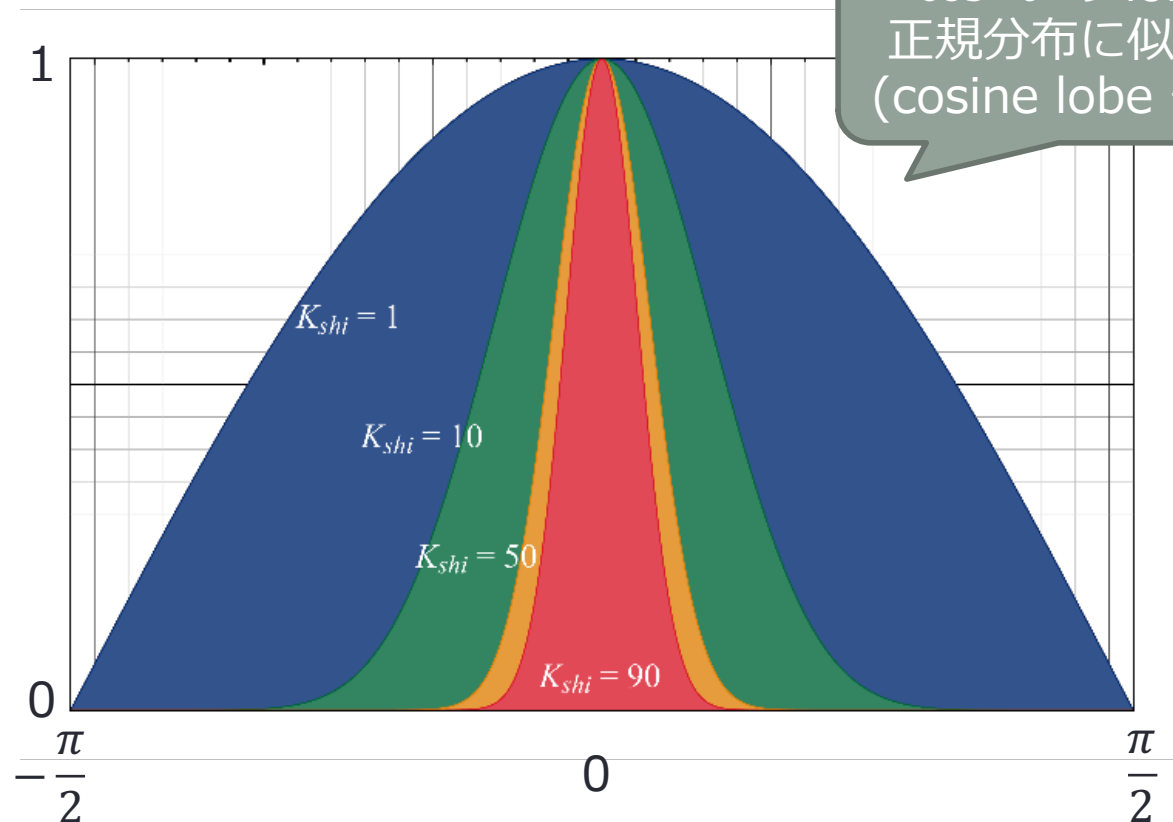


砥粒吹き付け
によるマット
仕上げ



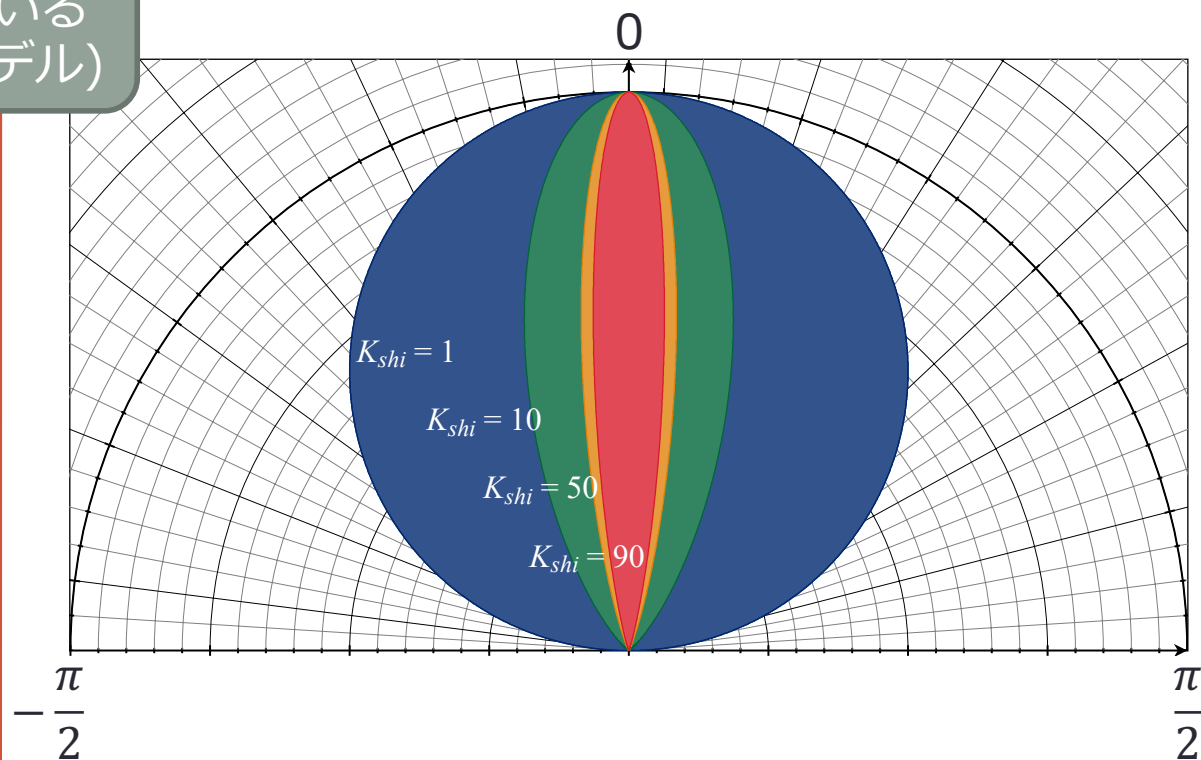
輝き係数

$$\cos^{K_{shi}} \theta$$

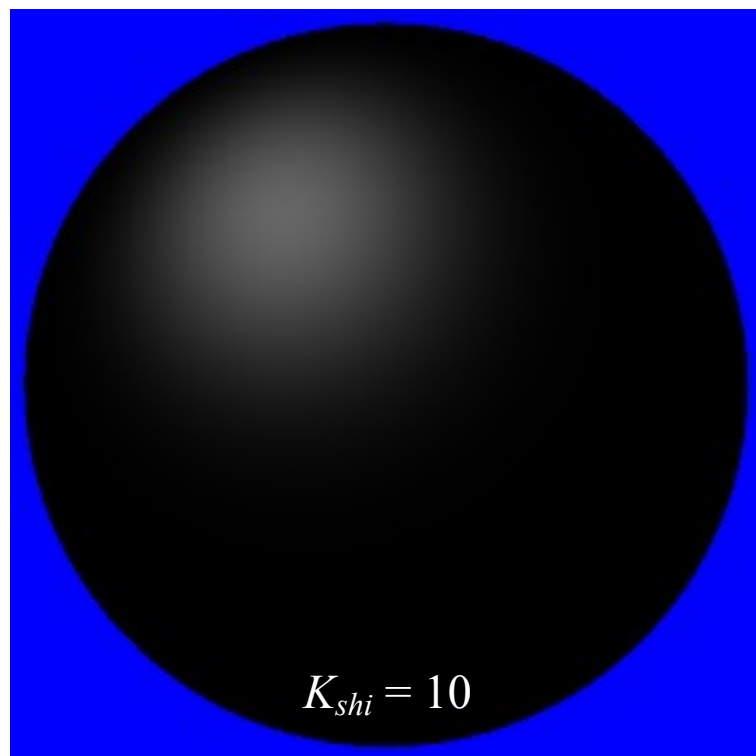


$\cos^n \theta$ の lobe は
正規分布に似ている
(cosine lobe モデル)

$$\cos^{K_{shi}} \theta \text{ の極座標表示}$$

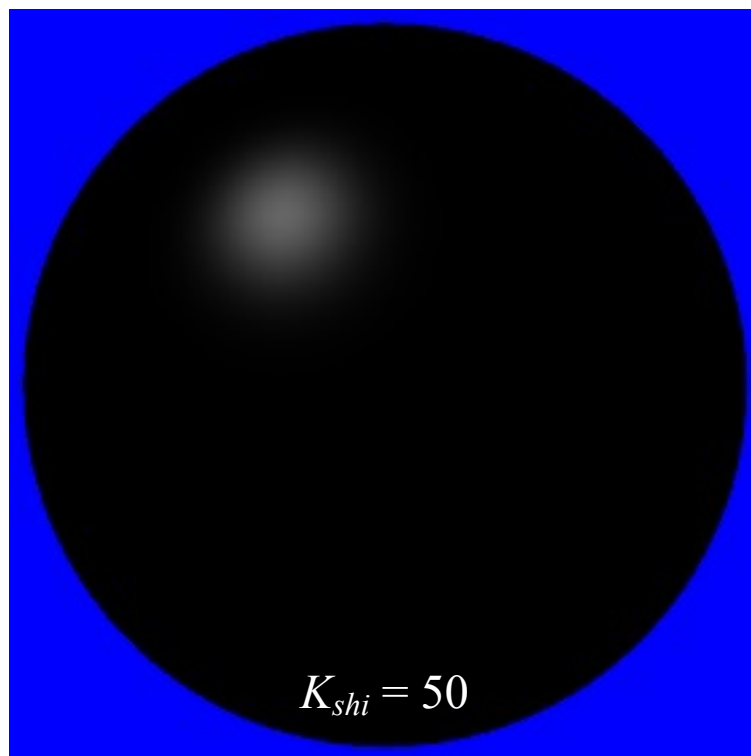


輝き係数によるハイライトの制御

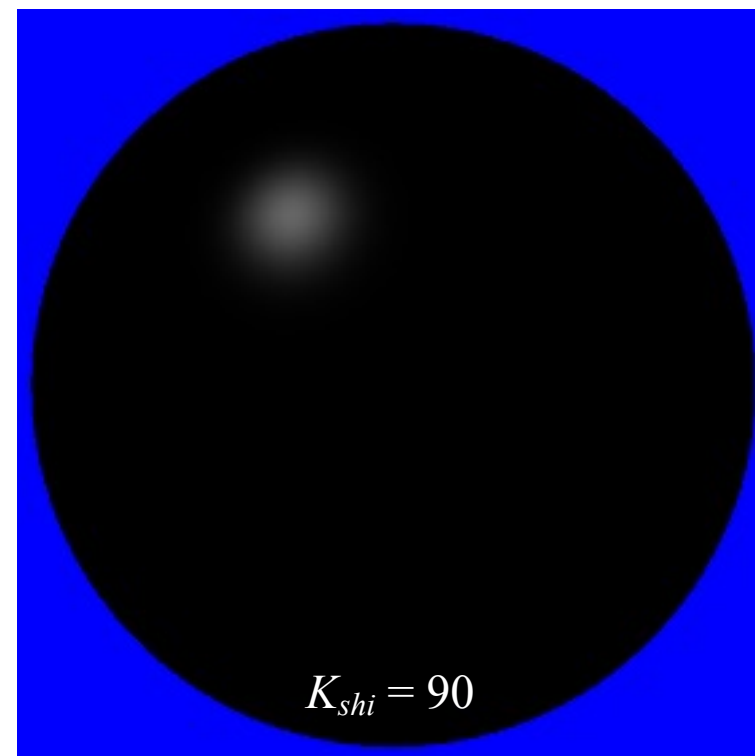


$K_{shi} = 10$

小



$K_{shi} = 50$



$K_{shi} = 90$

大



K_{shi}

その他の鏡面反射関数

- Schlick の近似

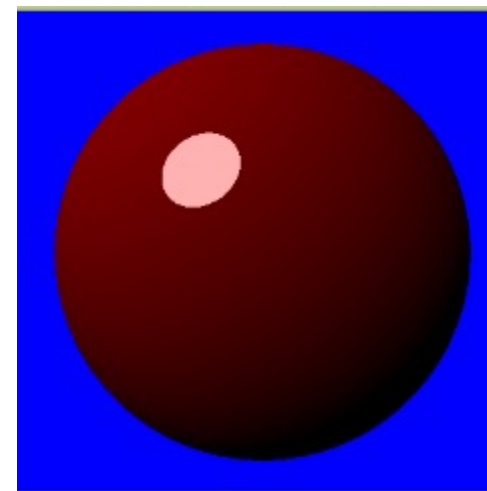
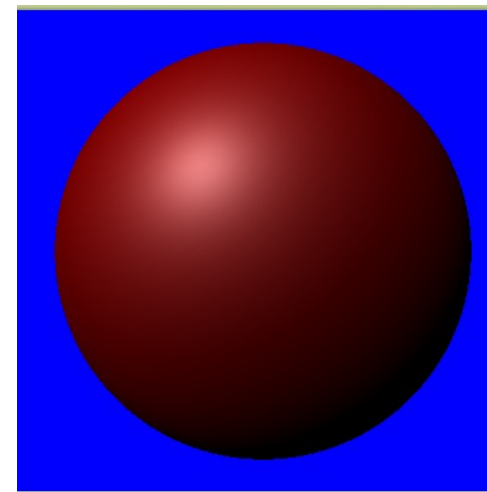
- $t = \cos \theta_r$ (Phong) または $t = \cos \theta_h$ (Blinn)

$$I_{spec} = \frac{t}{K_{shi} - tK_{shi} + t} K_{spec} \otimes L_{spec}$$

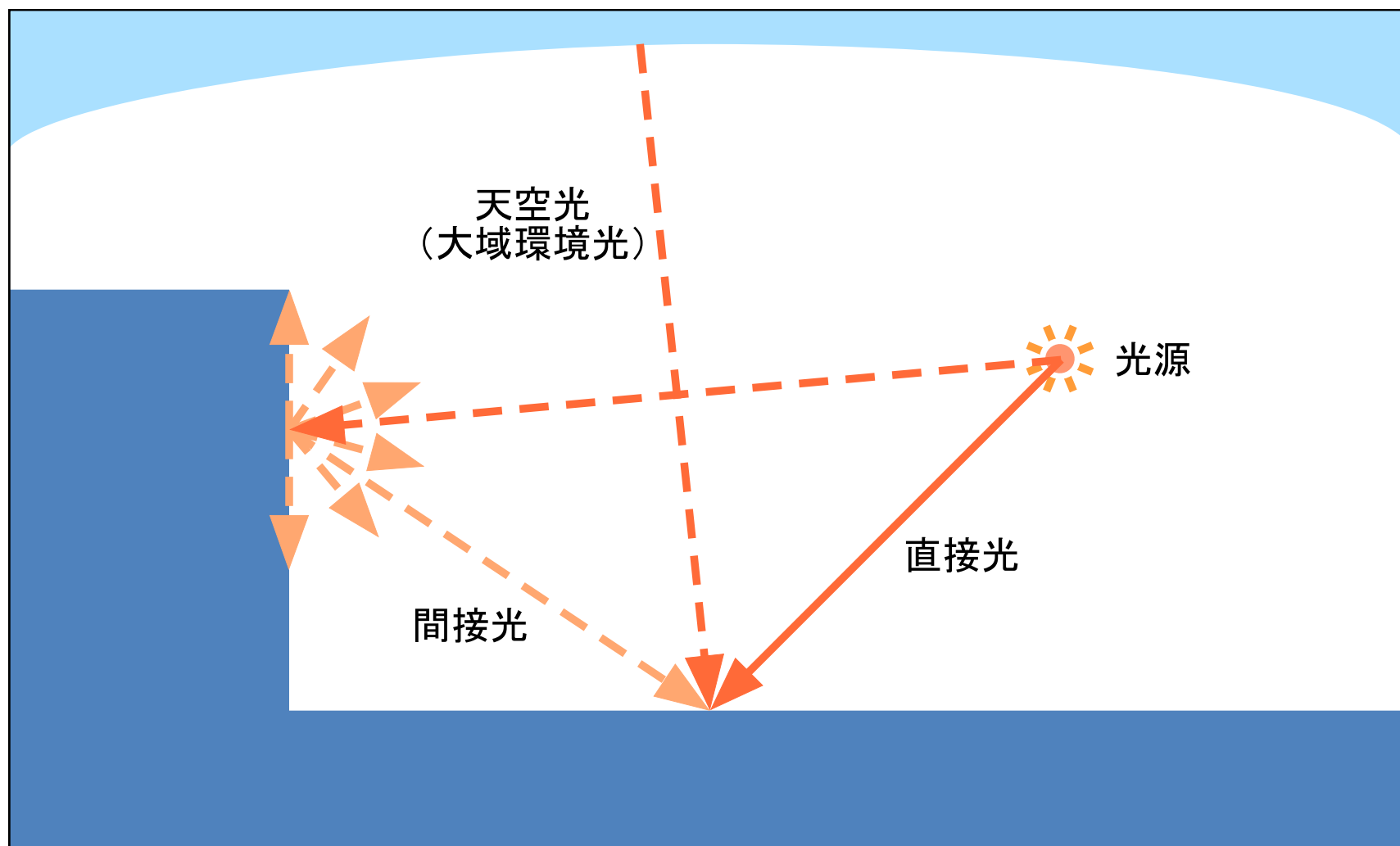
pow() を使わない

- 閾値 t

$$I_{spec} = [\max(\mathbf{N} \cdot \mathbf{H}, 0) - t] K_{spec} \otimes L_{spec}$$



環境光



環境光の経路

- 間接光の到来経路は非常に複雑
 - ある面から放射された反射光が再びその面に到達する場合もある
- 天空光は天空全体から到来する
 - すべての方向から入射する

これらの精密な計算をリアルタイムに行うことは困難



定数で近似する



環境光

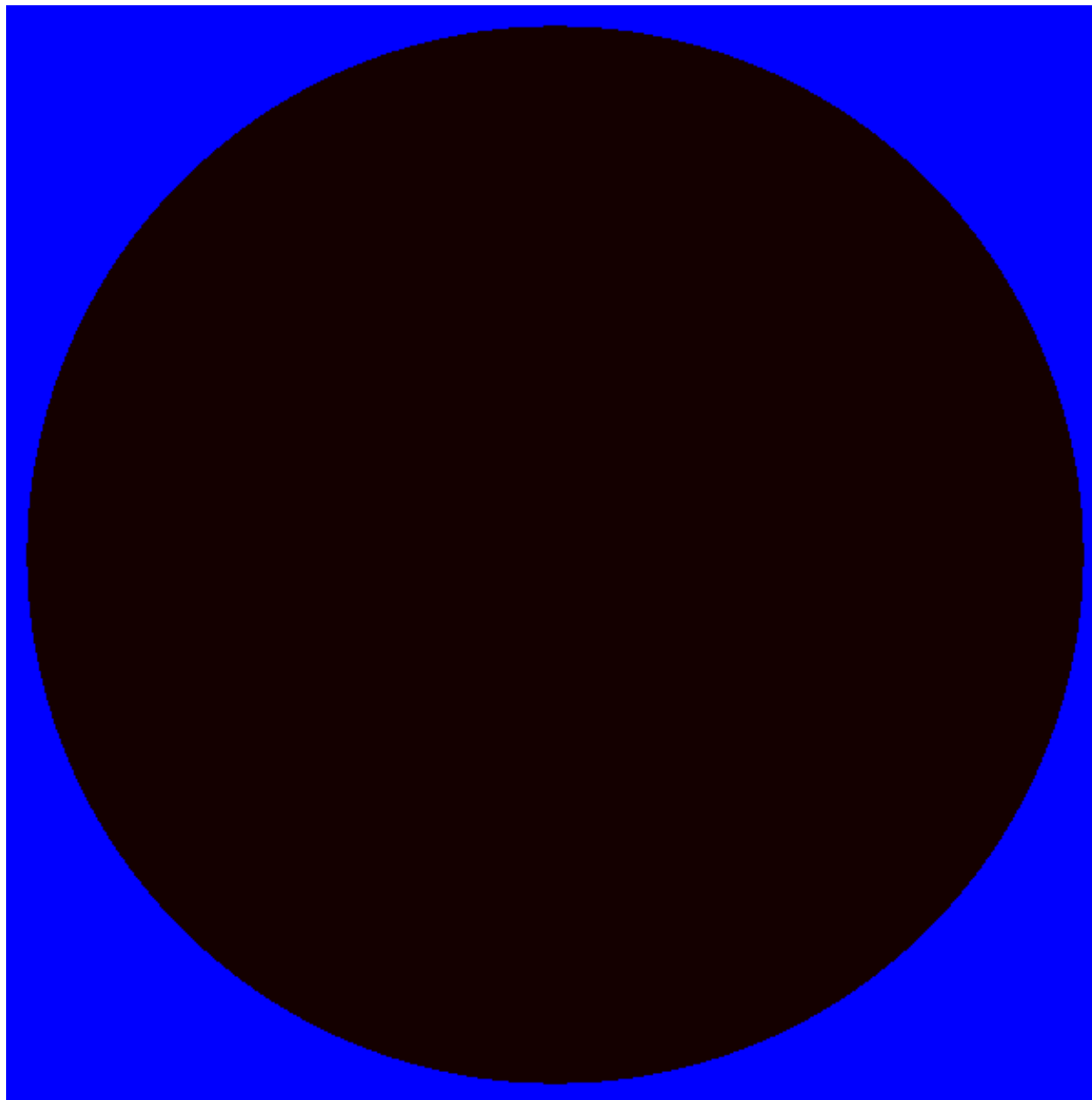
環境光の反射光強度

- I_{amb} : 環境光の反射光強度
- K_{amb} : 材質の環境光に対する反射係数
- L_{amb} : 光源強度の環境光成分
- \otimes : 要素ごとの積

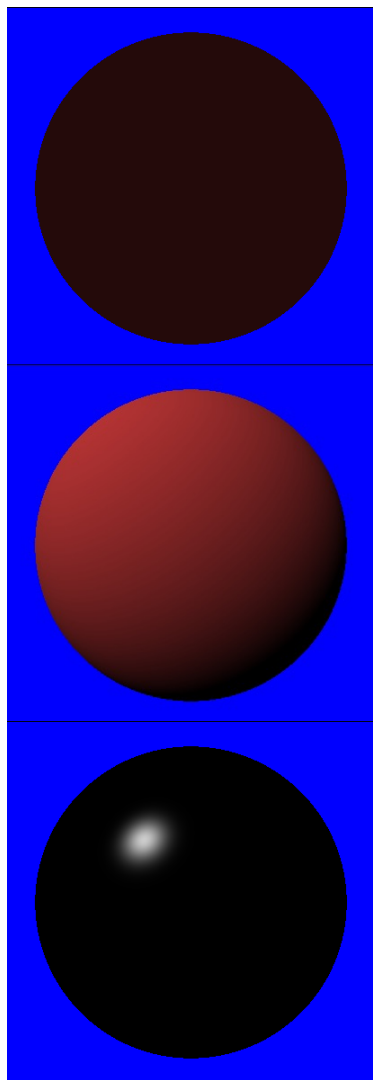
$$I_{amb} = K_{amb} \otimes L_{amb}$$

- K_{amb} は K_{diff} と同じにすることが多い
- L_{amb} は L_{diff} と成分比は同じでレベルを下げたものにする事が多い

環境光成分による陰影



照明方程式



環境光成分

I_{amb}

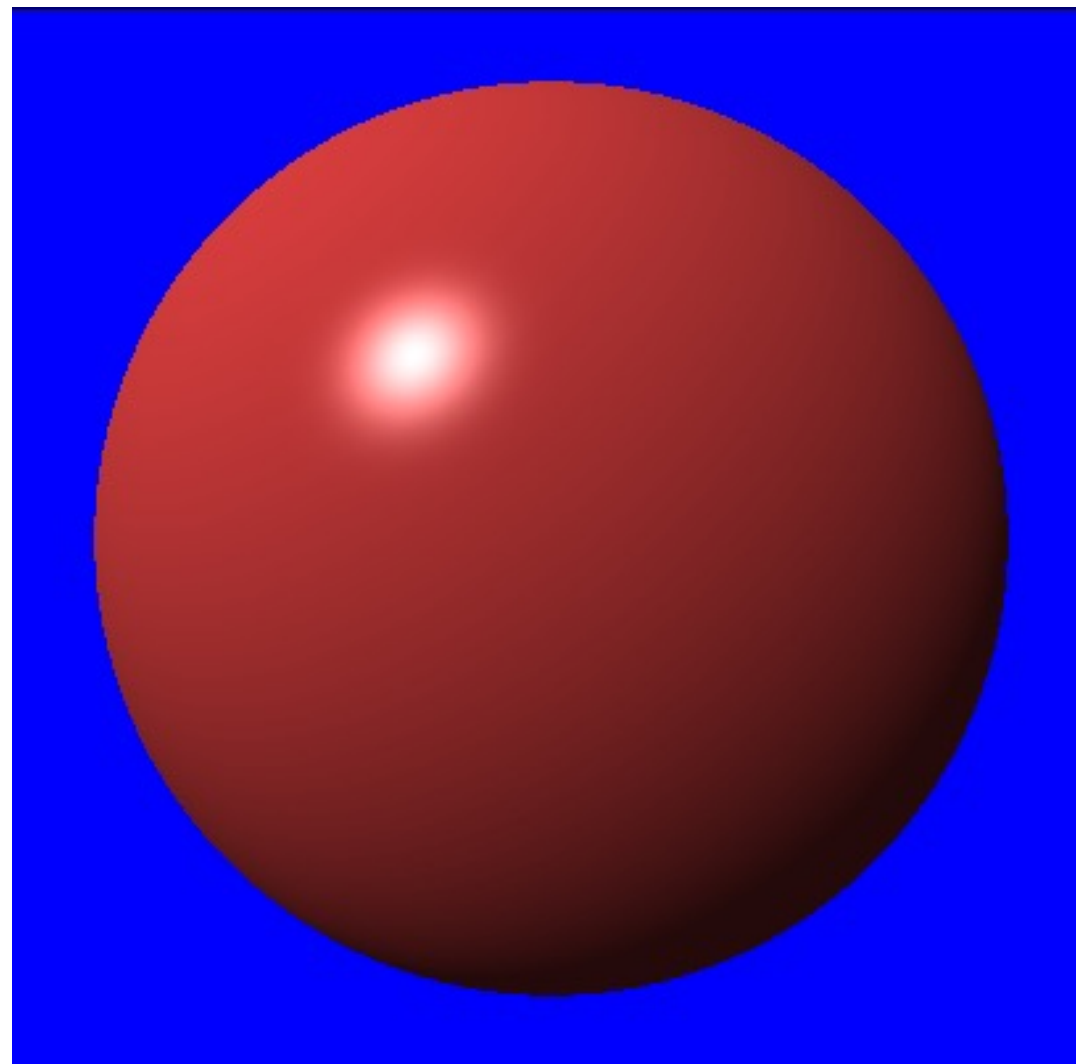
拡散反射光成分

I_{diff}

鏡面反射光成分

I_{spec}

$$I_{tot} = I_{amb} + I_{diff} + I_{spec}$$



距離に伴う減衰

- 点光源までの距離 r

$$r = |\mathbf{P}_l - \mathbf{P}|$$

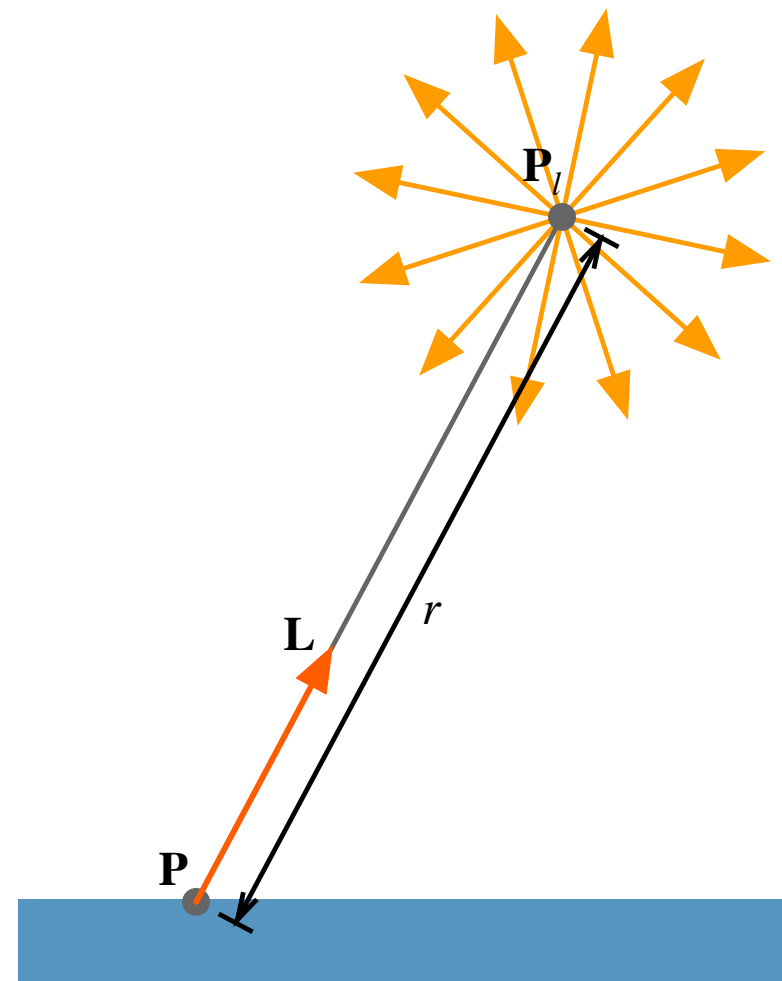
- 距離減衰 (distance falloff) 係数 d

$$d = \frac{1}{r^2}$$

物理的には距離
の二乗に反比例



減衰が急峻すぎる
(暗くなりすぎる)



距離減衰関数

- OpenGL/DirectX の固定機能パイプライン

$$d = \frac{1}{s_c + s_l r + s_q r^2}$$

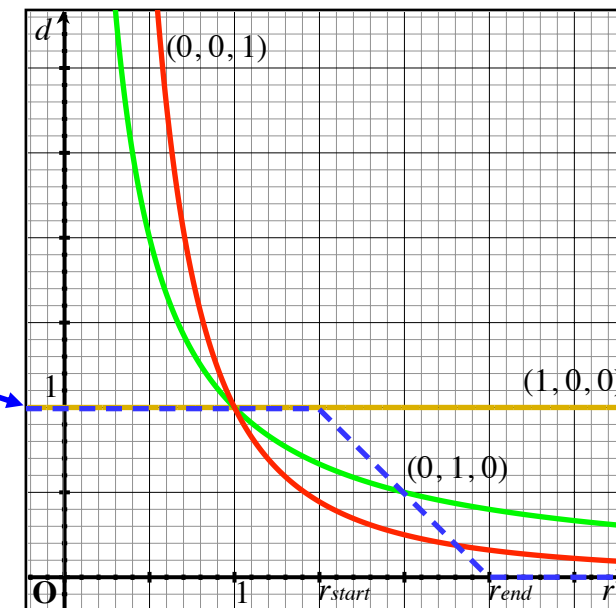
- とあるモデリングアプリケーション

$$d = \begin{cases} 1 & r \leq r_{start} \\ \frac{r_{end} - r}{r_{end} - r_{start}} & r_{start} < r < r_{end} \\ 0 & r \geq r_{end} \end{cases}$$

- 照明方程式

$$I_{tot} = I_{amb} + d(I_{diff} + I_{spec})$$

$$(s_c, s_l, s_q) = \begin{cases} (1, 0, 0) & \text{一定} \\ (0, 1, 0) & \text{反比例} \\ (0, 0, 1) & \text{物理的} \end{cases}$$



Pixar の映画で使われた距離減衰関数

$$d = \begin{cases} f_{max} e^{k_0 k_1 (r/r_c)} & (r \leq r_c) \\ f_c \left(\frac{r_c}{r}\right)^{s_e} & (r > r_c) \end{cases}$$

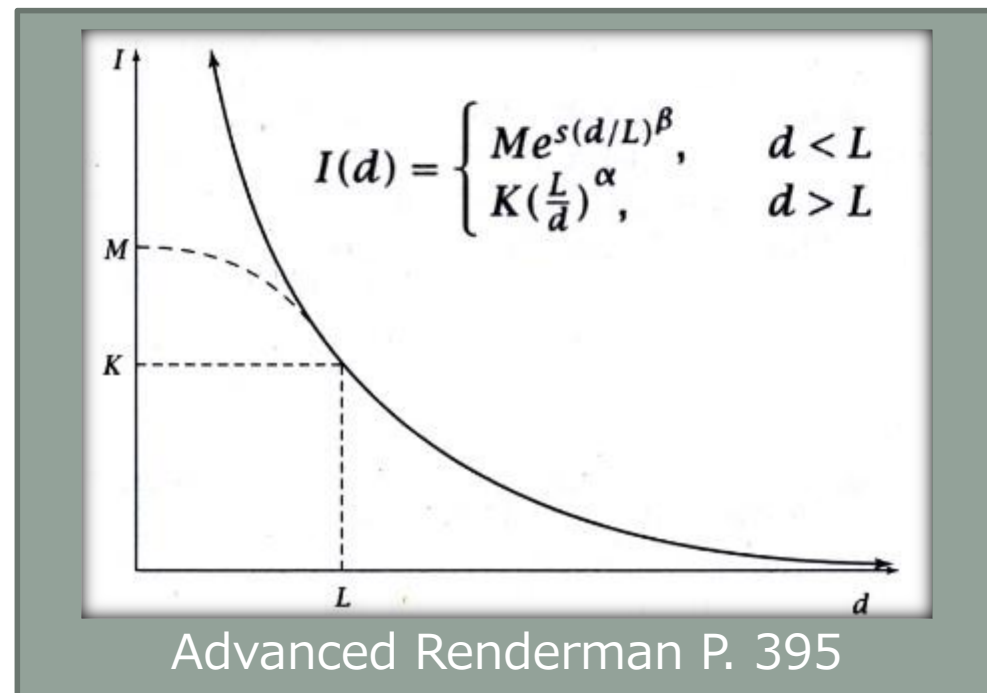
f_c : $r = r_c$ における減衰率

s_e : $r > r_c$ のときの減衰率の指数（どれだけ急に減衰するか）

f_{max} : $r \leq r_c$ で光源に近づいたときの減衰率の最大値

$k_0 = \log(f_c/f_{max})$

$k_1 = s_e/k_0$



計算コストが高い

スポットライトの場合

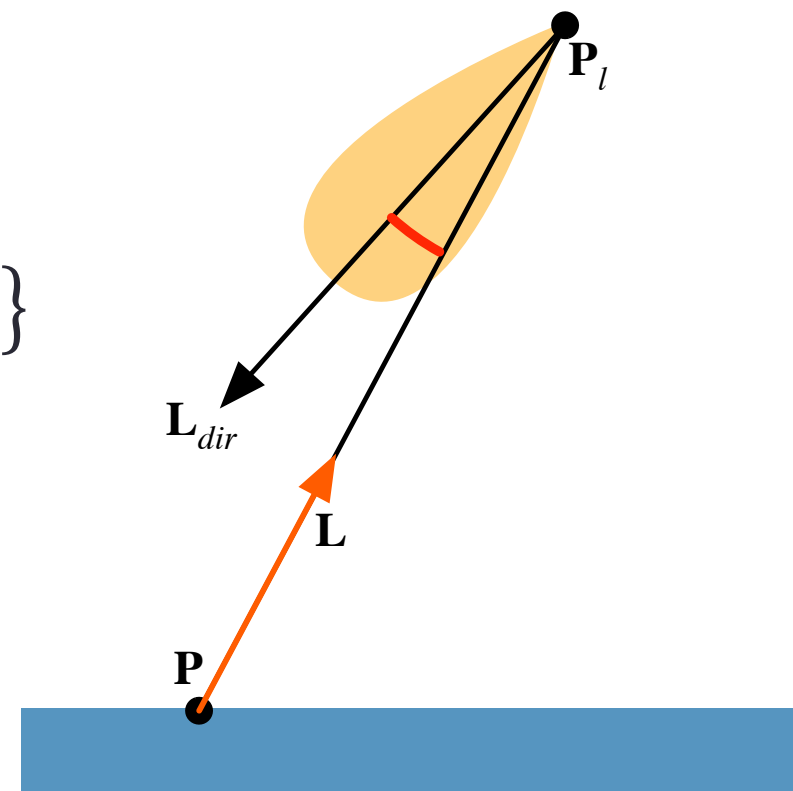
- スポットライトの配光分布

$$C_{spot} = \max(-\mathbf{L} \cdot \mathbf{L}_{dir}, 0)^{s_{exp}}$$

- 照明方程式

$$I_{tot} = C_{spot} \{ I_{amb} + d(I_{diff} + I_{spec}) \}$$

スポットライトの
広がり係数



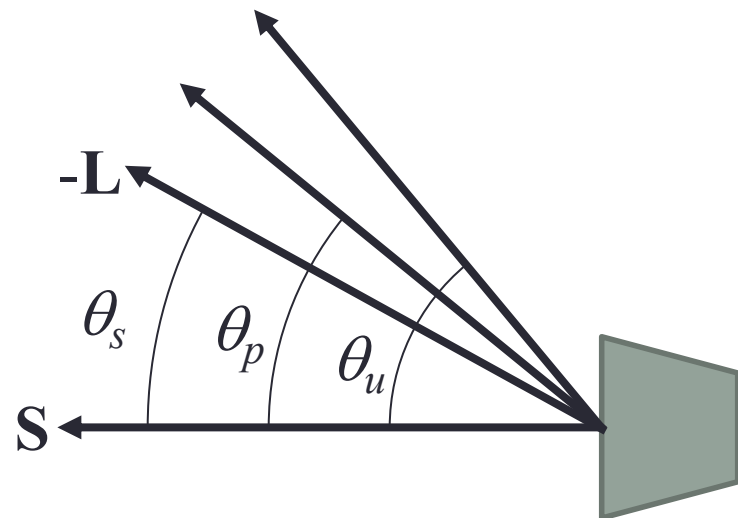
スポットライト (spotlights)

- OpenGL の固定機能パイプライン

$$C_{spot} = \begin{cases} \cos^{s_{exp}} \theta_s & (\theta_s \leq \theta_u) \\ 0 & (\theta_s > \theta_u) \end{cases}$$

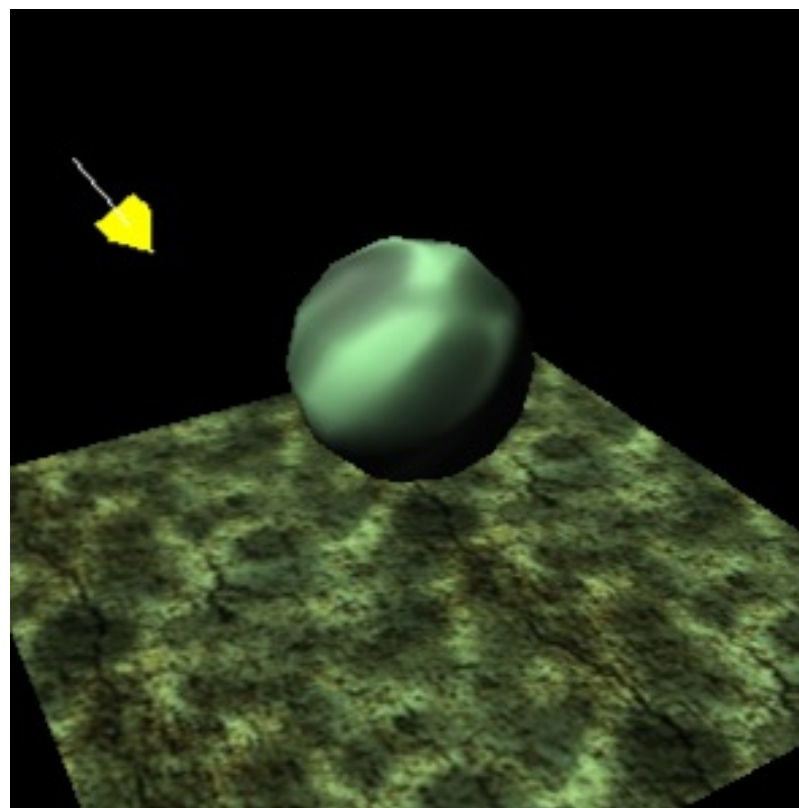
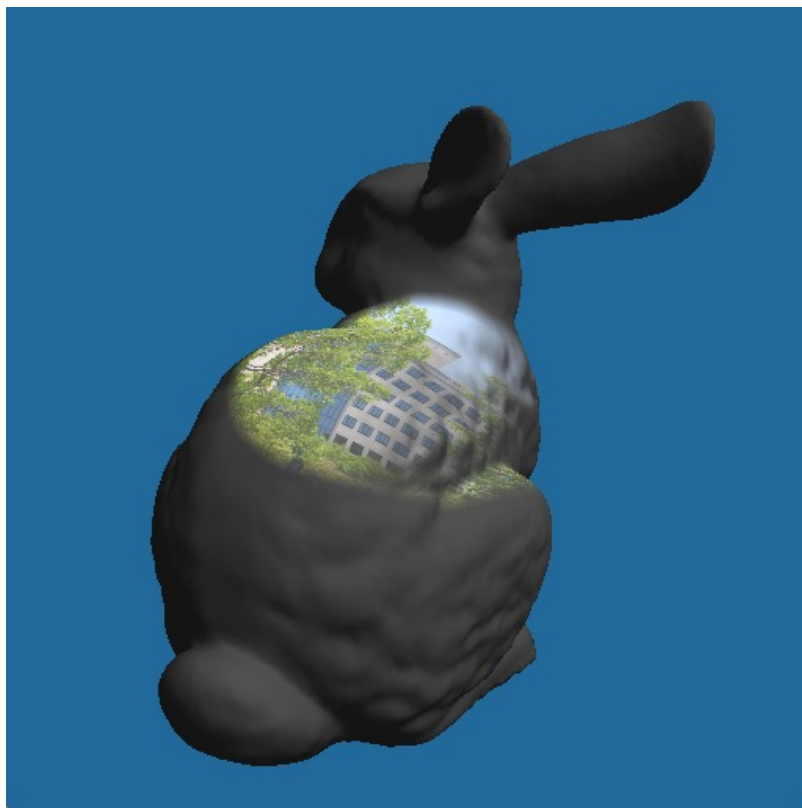
- DirectX の固定機能パイプライン

$$C_{spot} = \begin{cases} 1 & (\cos \theta_s \geq \cos \theta_p) \\ \cos^{s_{exp}} \theta_s & (\cos \theta_u < \cos \theta_s < \cos \theta_p) \\ 0 & (\cos \theta_s \leq \cos \theta_u) \end{cases}$$



テクスチャによる配光分布の制御

- 投影テクスチャ (Projective Texture)
 - スライドプロジェクタやスポットライトのような効果



<https://github.com/tokoik/projective>

大域環境光と自己発光

- 光源に依存しない環境光（背景光）

$$L_{glob}$$

- 自己発光強度

$$L_{emi}$$

- 照明方程式

$$I_{tot} = K_{amb} \otimes L_{glob} + L_{emi} + C_{spot} \{ I_{amb} + d(I_{diff} + I_{spec}) \}$$

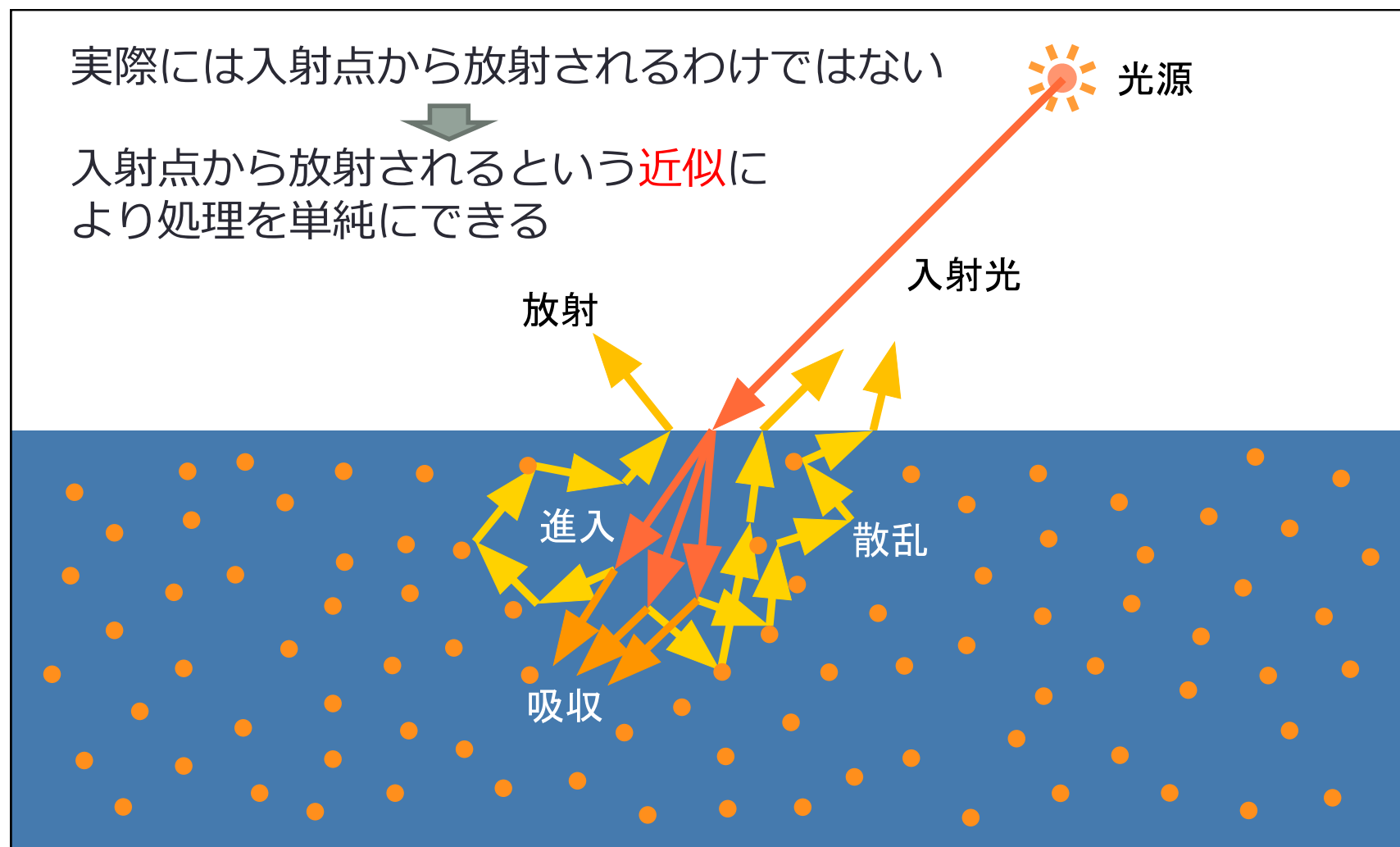
- 光源が n 個ある場合の照明方程式

$$I_{tot} = K_{amb} \otimes L_{glob} + L_{emi} + \sum_{k=1}^n C_{spot}^k \{ I_{amb}^k + d^k (I_{diff}^k + I_{spec}^k) \}$$

このモデルで表せないこと

近似の問題

表面下散乱



表面下散乱の有無

表面下散乱なし

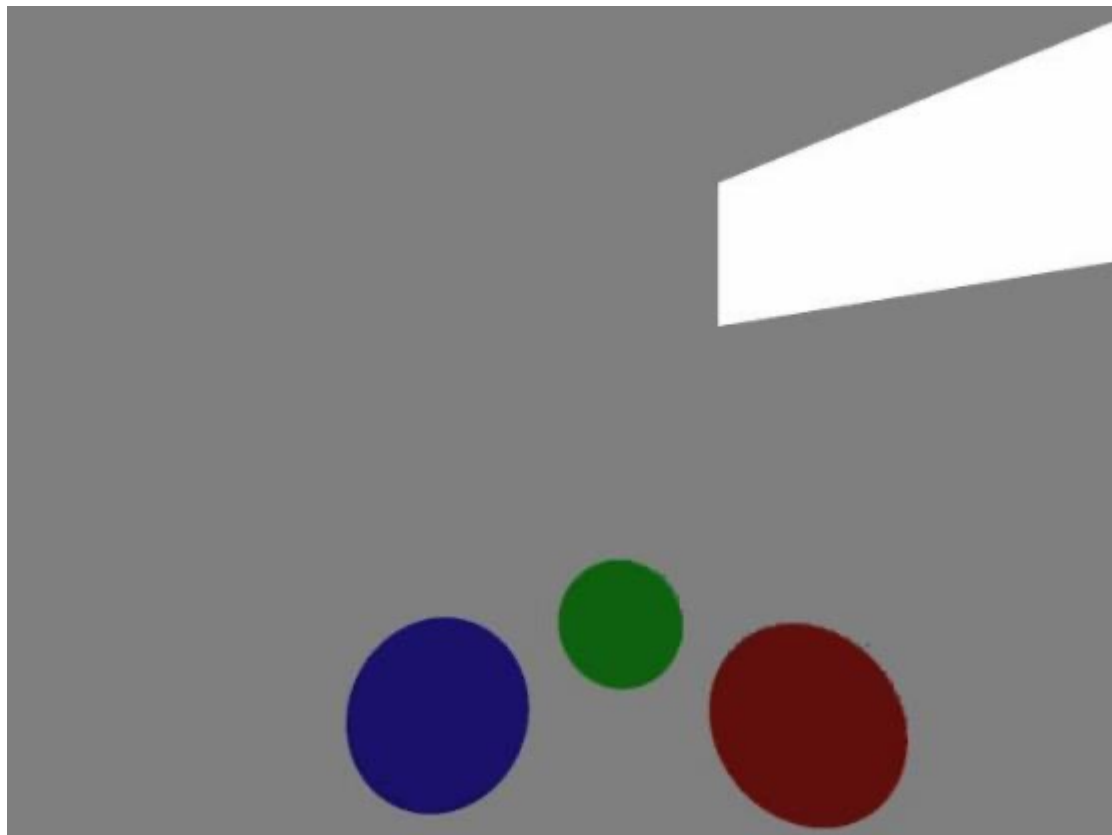


表面下散乱あり



直接光が存在しない場合

環境光が定数だと



間接光を計算すれば



テクスチャによる間接光の表現

- 拡散反射光強度は視点の位置に依存しない
 - 照明が動かないなら視点が動いても陰影は変わらない
- 拡散反射光強度だけを事前に計算しテクスチャを作る
 - レンダリングの際に「明るさ」のテクスチャを材質のテクスチャに乗じる
- 照明が異なるシーンでも材質のテクスチャを共有できる
 - 「明るさ」のテクスチャを切り替えれば良い
 - 「明るさ」のテクスチャの解像度は高くなくても良い
- Light Mapping

Light Mapping

Light Mapping 無



Light Mapping 有



面の陰影付け

陰影の補間

面全体の陰影

陰影付けモデルは物体表面上の一点に関するもの



面全体の陰影を求める必要がある



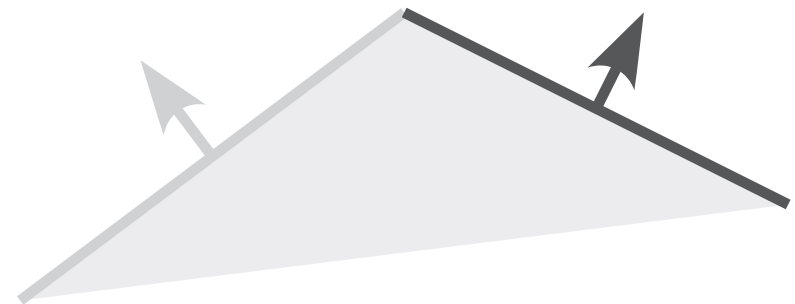
補間した頂点属性を使って面内の陰影を求める

面内の陰影

- **面ごと**に陰影を計算して面内の陰影は一定にする
 - ランバート (**Lambert**) シェーディング, フラットシェーディング
 - 陰影計算を行う必要がない (頂点属性をそのまま使う) 場合など
 - flat 修飾子が付けられた変数は三角形の最後の頂点のものしか使わない
- **頂点ごと**に陰影を計算して面内の陰影は線形補間で求める
 - グーロー (**Gouraud**) シェーディング
 - バーテックスシェーダで計算した陰影を頂点色として次のステージに送る
 - フラグメントシェーダは補間された頂点色を使って画素の色を決定する
- **画素ごと**に陰影を求める
 - フォン (**Phong**) シェーディングなど
 - バーテックスシェーダは法線ベクトル他の頂点属性を次のステージに送る
 - フラグメントシェーダは補間された頂点属性を使って陰影を計算する

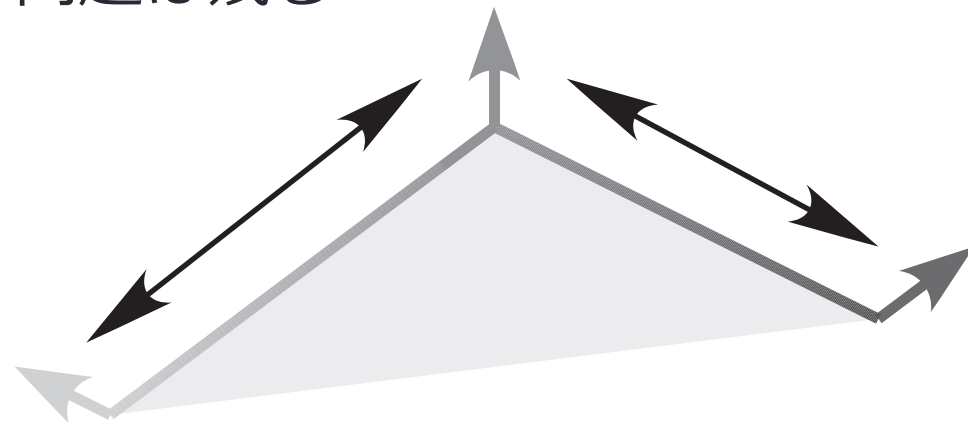
ランバート (Lambert) シェーディング

- 陰影計算を三角形単位に行う
 - 三角形上のすべての画素は同じ色になる (フラットシェーディング)
 - OpenGL では三角形の三頂点のうち最後に指定された頂点色を用いる
- 高速である
- 最も実装が簡単である
- 滑らかな曲面を表現するには多くの三角形が必要である



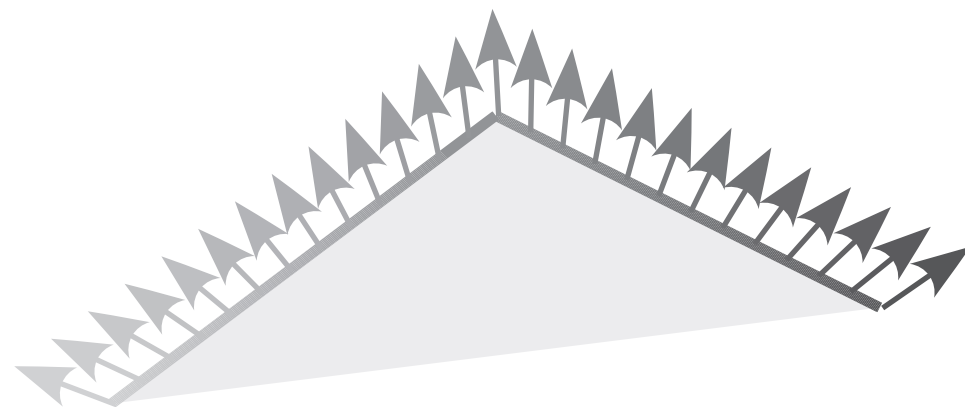
グーロー (Gouraud) シェーディング

- 陰影計算を頂点単位に行う
 - 三角形上の各画素の色は補間により求める
- 多くのグラフィックスハードウェアに実装されている
 - 陰影は頂点で計算されるためランバートシェーディング並に高速
- 少ない三角形で滑らかな曲面を表現することができる
 - ハイライトの消失やマッハバンドの発生などの問題は残る

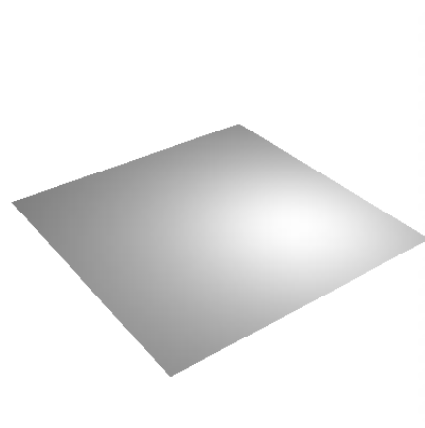
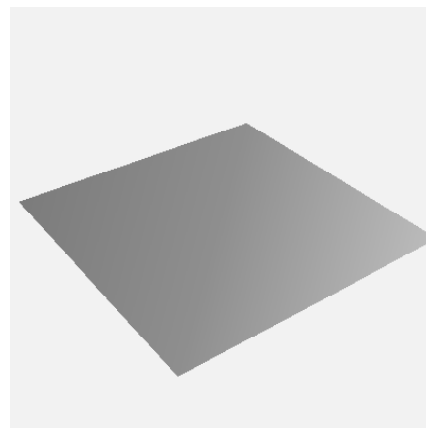
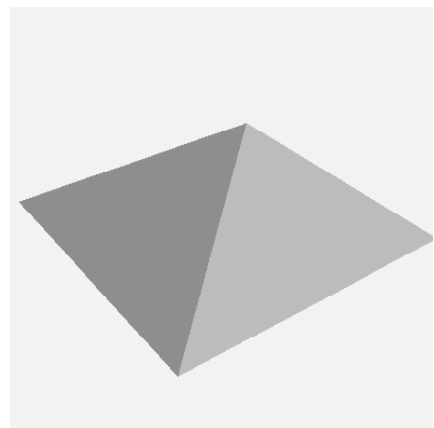
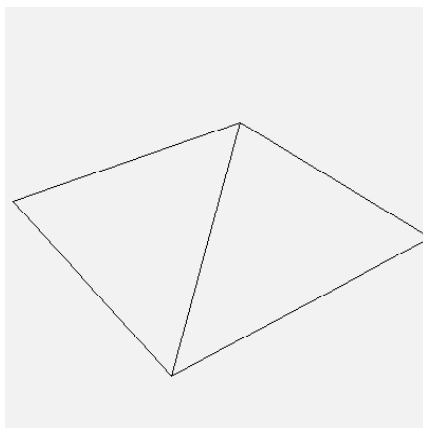


フォン (Phong) シェーディング

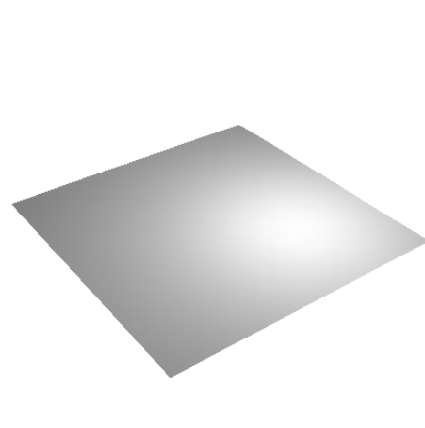
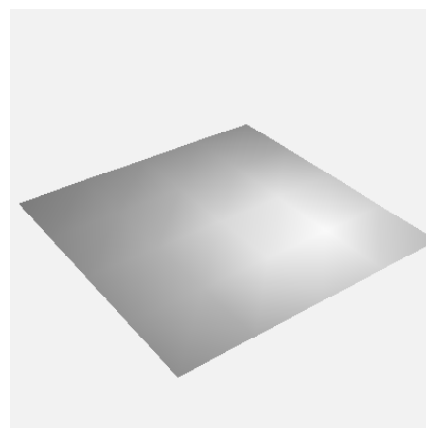
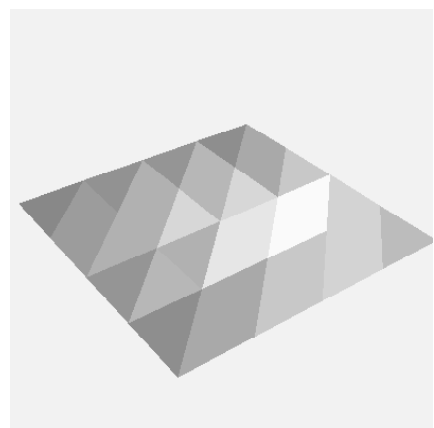
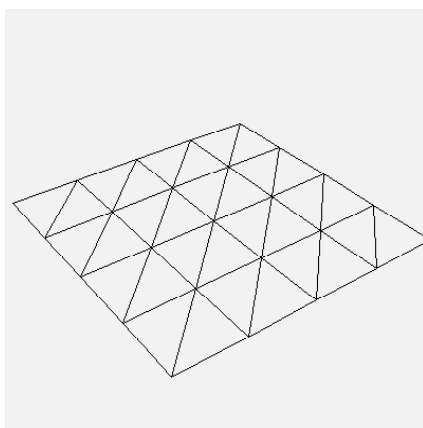
- 陰影計算を画素単位に行う
 - 頂点の法線ベクトルや頂点のワールド座標を補間して陰影計算を行う
 - 頂点単位の陰影計算に比べて計算コストが高い
- フラグメントシェーダを使って実装できる
 - テクスチャを使ってシミュレーションすることもできる



ランバート・グーロー・フォンの違い



ランバートシェーディング グーローシェーディング フォンシェーディング

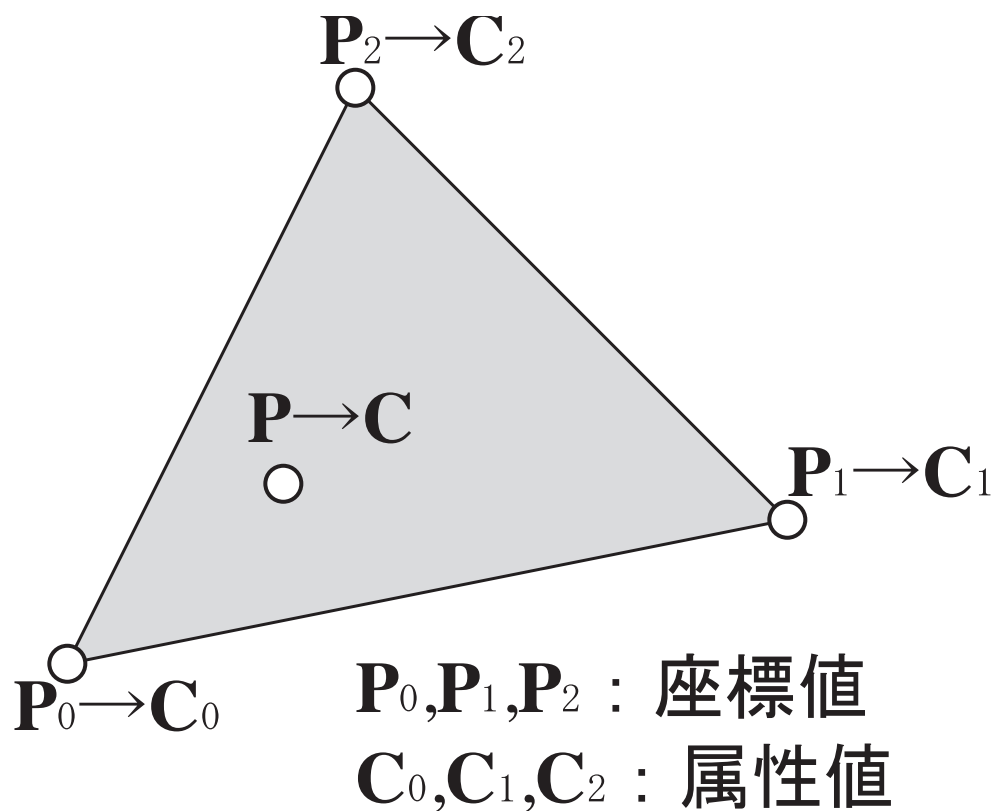


ランバートシェーディング グーローシェーディング フォンシェーディング

頂点属性の線形補間

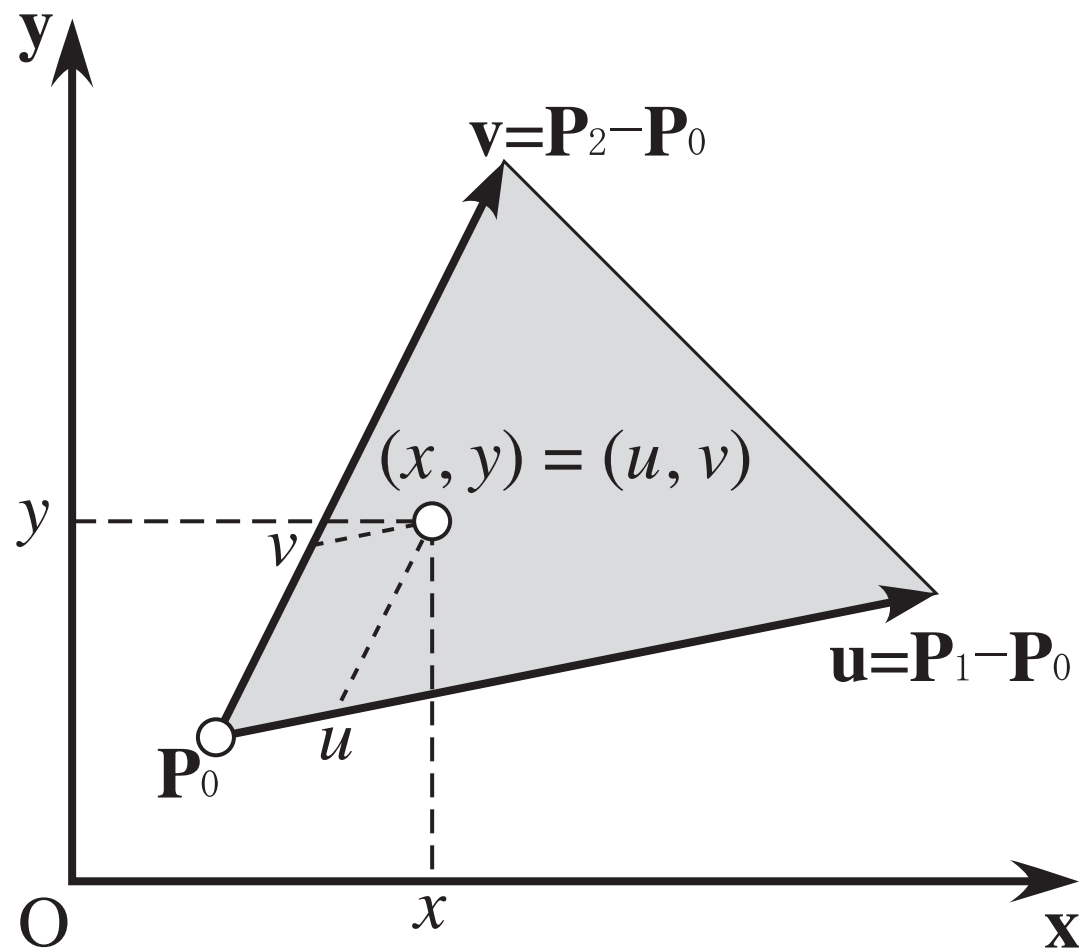
透視変換の影響

頂点属性の線形補間



- 三角形の頂点の座標値 (P_0, P_1, P_2 , `gl_Position` に出力するもの) に属性値 (C_0, C_1, C_2 , 色など) を対応づける
- 三角形の内部の点 P における属性値 C を線形補間により求める

三角形の内部のパラメータ座標



- 内部の点 $\mathbf{P} = (x, y)$ を $u = \mathbf{P}_1 - \mathbf{P}_0$, $v = \mathbf{P}_2 - \mathbf{P}_0$ を軸とする座標 (u, v) で表す

$$x\mathbf{x} + y\mathbf{y} + \mathbf{O} = u\mathbf{u} + v\mathbf{v} + \mathbf{P}_0$$

$$\mathbf{x} = (1, 0)$$

$$\mathbf{y} = (0, 1)$$

$$\mathbf{u} = (x_u, y_u)$$

$$\mathbf{v} = (x_v, y_v)$$

$$\mathbf{P}_0 = (x_0, y_0)$$

$$\mathbf{P}_1 = (x_1, y_1)$$

$$\mathbf{P}_2 = (x_2, y_2)$$

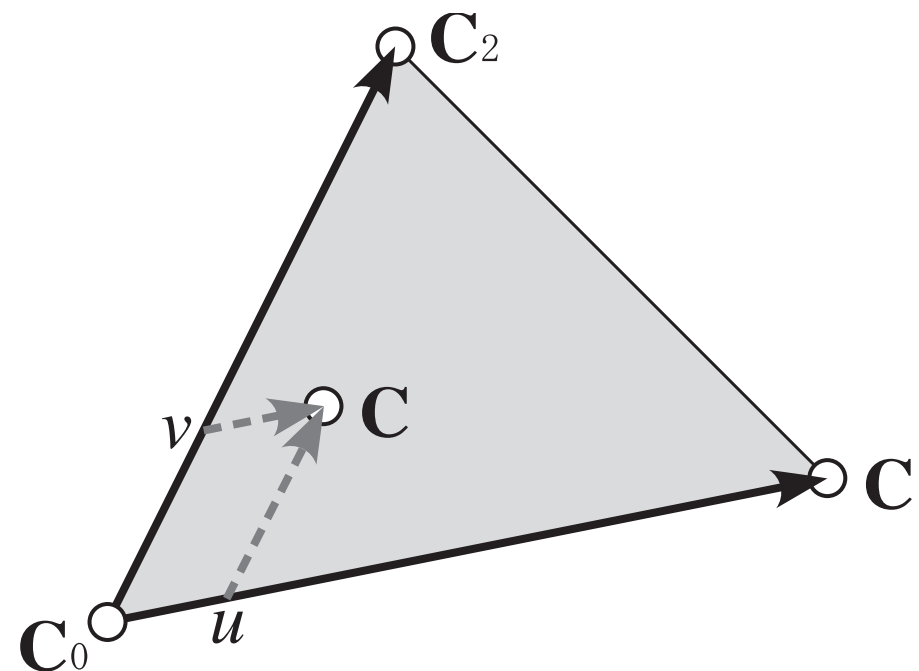
パラメータ座標による補間

- 連立方程式で表す

$$\begin{cases} x = ux_u + vx_v + x_0 \\ y = uy_u + vy_v + y_0 \end{cases}$$

- u, v について解く

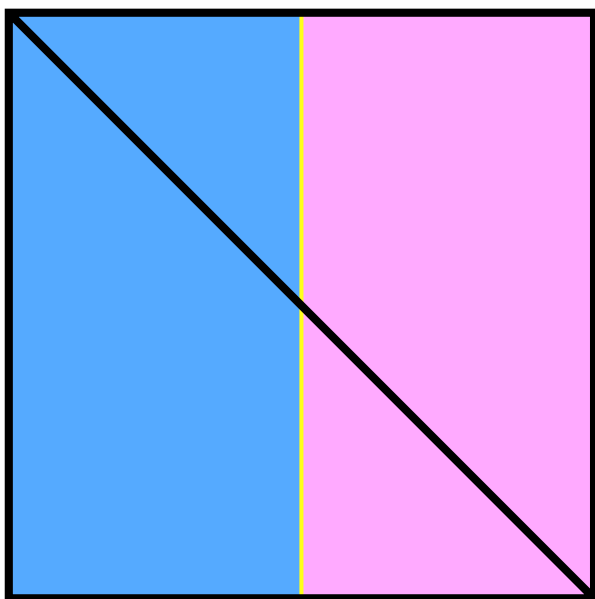
$$\begin{cases} u = \frac{(x - x_0)y_v - (y - y_0)x_v}{x_u y_v - x_v y_u} \\ v = \frac{(y - y_0)x_u - (x - x_0)y_u}{x_u y_v - x_v y_u} \end{cases}$$



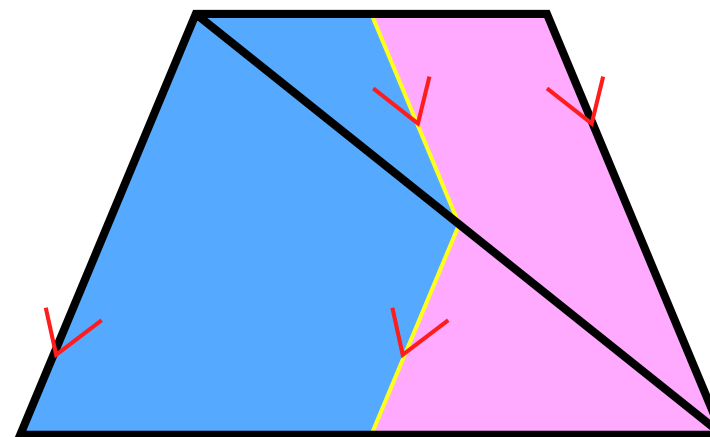
$$\begin{aligned} \mathbf{C} &= u(\mathbf{C}_1 - \mathbf{C}_0) + v(\mathbf{C}_2 - \mathbf{C}_0) + \mathbf{C}_0 \\ &= (1 - u - v)\mathbf{C}_0 + u\mathbf{C}_1 + v\mathbf{C}_2 \end{aligned}$$

透視投影の影響

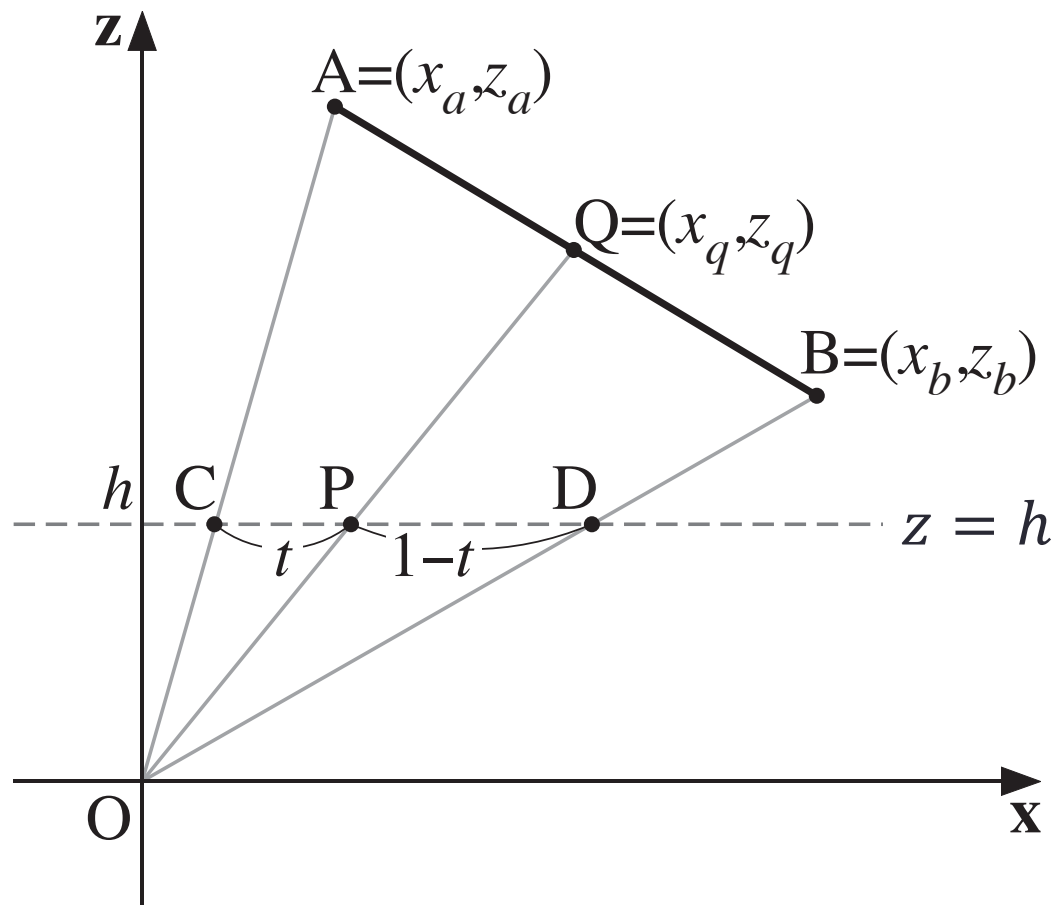
正面から見る



浅い角度から見る



線形補間の逆透視投影



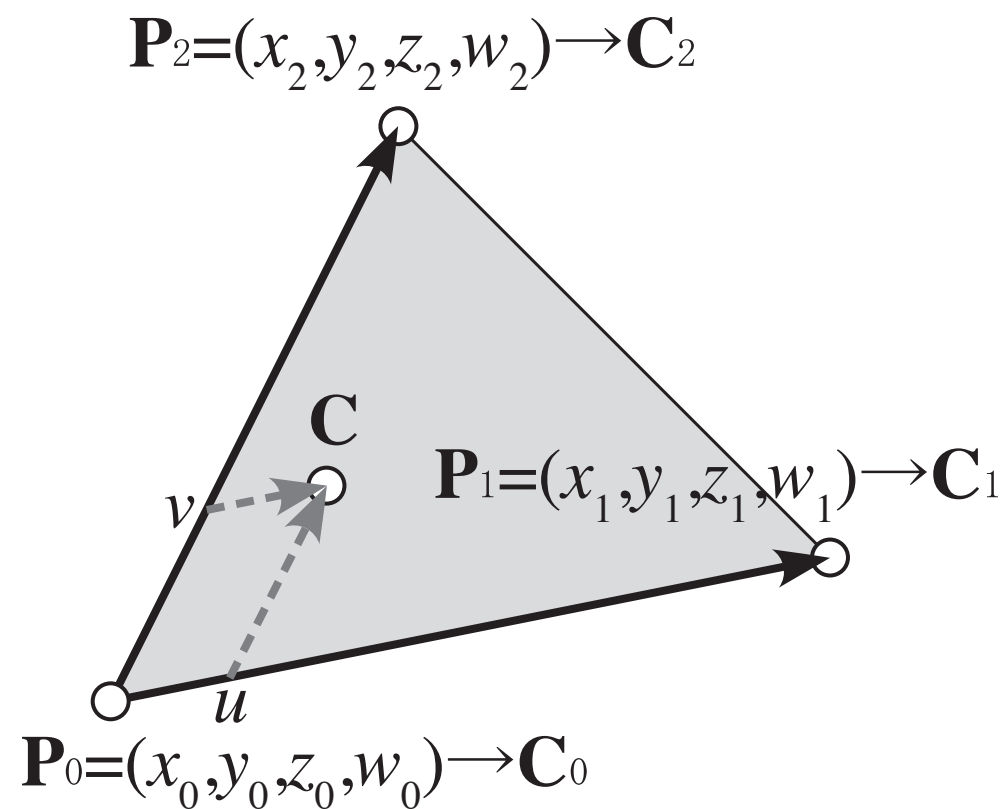
- P を AB 上に投影した点 Q

$$x_q = \frac{\frac{x_a}{z_a} (1 - t) + \frac{x_b}{z_b} t}{\frac{1}{z_a} (1 - t) + \frac{1}{z_b} t}$$

$$z_q = \frac{1}{\frac{1}{z_a} (1 - t) + \frac{1}{z_b} t}$$

1. 属性値を座標値の w (透視投影により z が格納されている) で割ったもの (スクリーン上の位置) を線形補間
2. これを w の逆数を線形補間したもので割る

透視投影を考慮した補間



$$\mathbf{C} = w \left\{ (1 - u - v) \frac{\mathbf{C}_0}{w_0} + u \frac{\mathbf{C}_1}{w_1} + v \frac{\mathbf{C}_2}{w_2} \right\}$$

$$w = \frac{1}{(1 - u - v) \frac{1}{w_0} + u \frac{1}{w_1} + v \frac{1}{w_2}}$$

頂点に色を付ける

バーテックスシェーダからフラグメントシェーダにデータを渡す

in/out 変数のインデックスを指定する

```
// プログラムオブジェクトの作成
const GLuint program(glCreateProgram());

... (ソースプログラムの読み込み, コンパイル, 取り付け等)

// プログラムオブジェクトのリンク
glBindAttribLocation(program, 0, "pv");
glBindFragDataLocation(program, 0, "fc");
glLinkProgram(program);

...

// uniform 変数 mc のインデックスの検索 (見つからなければ -1)
const GLint mcLoc(glGetUniformLocation(program, "mc"));
```

pv のインデックス

fc のカラー番号

頂点配列オブジェクトを作成する

```
// 頂点配列オブジェクト  
GLuint vao;  
glGenVertexArrays(1, &vao);  
glBindVertexArray(vao);
```

頂点バッファオブジェクトを作成する

```
// 頂点バッファオブジェクト  
GLuint vbo[1];  
glGenBuffers(1, vbo);
```

頂点バッファオブジェクトに位置のデータを転送する

// 頂点の位置

```
static const GLfloat pv[][3] =  
{  
    { -1.0f, -0.8660254f, 0.0f },  
    {  1.0f, -0.8660254f, 0.0f },  
    {  0.0f,  0.8660254f, 0.0f },  
};
```

三角形の
3頂点の
位置

配列の要素数を求める: `sizeof pv / sizeof pv[0]`

// 頂点の数

```
constexpr int points(std::size(pv));
```

// 一つ目の頂点バッファオブジェクトに頂点の位置のデータを転送する

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
```

```
glBufferData(GL_ARRAY_BUFFER, sizeof pv, pv, GL_STATIC_DRAW);
```

// この頂点バッファオブジェクトを index == 0 の in 変数から取得する

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

```
glEnableVertexAttribArray(0);
```

頂点配列オブジェクトの内容を描く

```
// シェーダプログラムの選択
glUseProgram(program);

// uniform 変数 mc (モデルビュー・投影変換行列) を設定する
glUniformMatrix4fv(mcLoc, 1, GL_FALSE, mc);

// 描画に使う頂点配列オブジェクトの指定
glBindVertexArray(vao);

// 図形の描画
glDrawArrays(GL_TRIANGLES, 0, points);
```


このときのシェーダプログラム

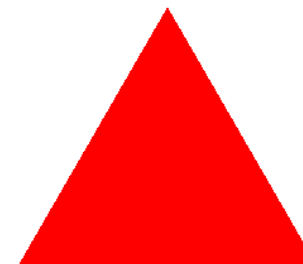
バーテックスシェーダ

```
#version 410
in vec4 pv;
uniform mat4 mc;
void main(void)
{
    gl_Position = mc * pv;
}
```

フラグメントシェーダ

```
#version 410
out vec4 fc;
void main(void)
{
    fc = vec4(1.0, 0.0, 0.0, 1.0);
}
```

定数 (赤)



追加する in (attribute) 変数 cv のインデックスを得る

```
// プログラムオブジェクトの作成
const GLuint program(glCreateProgram());

... (ソースプログラムの読み込み, コンパイル, リンク等)

glLinkProgram(program);

...

// in (attribute) 変数 cv のインデックスの検索 (見つからなければ -1)
const GLint cvLoc(glGetAttribLocation(program, "cv"));

// uniform 変数 mc のインデックスの検索 (見つからなければ -1)
const GLint mcLoc(glGetUniformLocation(program, "mc"));
```

頂点配列オブジェクトを作成する

```
// 頂点配列オブジェクト  
GLuint vao;  
glGenVertexArrays(1, &vao);  
glBindVertexArray(vao);
```

さっき
と同じ

頂点バッファオブジェクトを作成する

```
// 頂点バッファオブジェクト  
GLuint vbo[2];  
glGenBuffers(2, vbo);
```

頂点バッファオブジェクトを2個作る

頂点の位置の頂点バッファオブジェクト

```
// 頂点の位置
static const GLfloat pv[][3] =
{
    { -1.0f, -0.8660254f, 0.0f },
    {  1.0f, -0.8660254f, 0.0f },
    {  0.0f,  0.8660254f, 0.0f },
};

// 頂点の数
constexpr int points(std::size(pv));

// 一つ目の頂点バッファオブジェクトに頂点の位置のデータを転送する
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof pv, pv, GL_STATIC_DRAW);

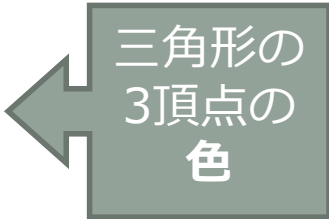
// この頂点バッファオブジェクトを index == 0 の in 変数から取得する
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
```

さっき
と同じ

頂点の色の頂点バッファオブジェクト

// 頂点の色

```
static const GLfloat cv[][3] =  
{  
    { 1.0f, 0.0f, 0.0f }, // 赤  
    { 0.0f, 1.0f, 0.0f }, // 緑  
    { 0.0f, 0.0f, 1.0f }, // 青  
};
```



三角形の
3頂点の
色

// 二つ目の頂点バッファオブジェクトに色のデータを転送する

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);  
glBufferData(GL_ARRAY_BUFFER, sizeof cv, cv, GL_STATIC_DRAW);
```

// この頂点バッファオブジェクトを index が cvLoc の in 変数から取得する

```
glVertexAttribPointer(cvLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);  
glEnableVertexAttribArray(cvLoc);
```

頂点配列オブジェクトの内容を描く

```
// シェーダプログラムの選択
glUseProgram(program);

// uniform 変数 mc (モデルビュー・投影変換行列) を設定する
glUniformMatrix4fv(mcLoc, 1, GL_FALSE, mc);

// 描画に使う頂点配列オブジェクトの指定
glBindVertexArray(vao);

// 図形の描画
glDrawArrays(GL_TRIANGLES, 0, points);
```

さっき
と同じ

シェーダプログラム

バーテックスシェーダ

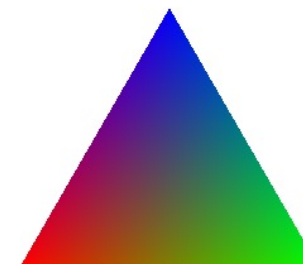
```
#version 410
in vec4 pv;
in vec4 cv;
uniform mat4 mc;
out vec4 vc;
void main(void)
{
    vc = cv;
    gl_Position = mc * pv;
}
```

頂点の色

補間される

フラグメントシェーダ

```
#version 410
in vec4 vc;
out vec4 fc;
void main(void)
{
    fc = vc;
}
```



補間方法の選択

バーテックスシェーダ

フラグメントシェーダ

```
#version 410
```

透視投影の影響を考慮した線形補間

```
out vec4 c1;  
smooth out vec4 c2;
```

```
#version 410
```

```
in vec4 c1;  
smooth in vec4 c2;
```

透視投影の影響を考慮しない線形補間

```
noperspective out vec4 c3;
```

```
noperspective in vec4 c3;
```

補間しない

最後の頂点属性が使われる

```
flat out vec4 c4;
```

```
flat in vec4 c4;
```

小テスト－陰影付け

Moodle の小テストに解答してください

宿題

- グローシェーディングによる陰影付けを行ってください
 - 次のプログラムは attribute に頂点の位置と色を指定して三角形を描きます。頂点の色は補間されて面内を塗りつぶします
 - <https://github.com/tokoik/ggsample06>
 - この色 c の代わりに頂点における法線ベクトル n を与えてバーテックスシェーダで頂点の陰影を計算し, out 変数 vc に代入するようにしてください
 - 陰影計算にはスライド 31 枚目の式, 鏡面反射光のモデルにはスライド 21 枚目の Blinn のモデルを使用してください
 - 必要な定数は ggsample06.vert 内で定義しています
- ggsample06.vert をアップロードしてください

main.cpp の変更点

```
// 頂点の色
static const GLfloat cv[][3] =
{
    { 1.0f, 0.0f, 0.0f }, // 赤
    { 0.0f, 1.0f, 0.0f }, // 緑
    { 0.0f, 0.0f, 1.0f }, // 青
};

// 【宿題】 頂点の法線ベクトルを頂点の色の代わりに使う
static const GLfloat nv[][3] =
{
    { -0.086172748f, -0.049751860f, 0.99503719f },
    {  0.086172748f, -0.049751860f, 0.99503719f },
    {  0.0f,          0.099503719f, 0.99503719f }
};

// 頂点の色 cv 用のバッファオブジェクト
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);

// 【宿題】 cv の代わりに nv を使うように変更する
glBufferData(GL_ARRAY_BUFFER, sizeof cv, cv, GL_STATIC_DRAW);
```

この cv を nv
に置き換える

GLSL の補足

- ベクトルの正規化

```
vec3 h = normalize(1 + v); // hは1とvの中間ベクトル
```

- ベクトル同士の積は要素ごとの積になる

```
vec3 v3 = v1 * v2; // ベクトルv1とv2の各要素をかける
```

- ベクトルとスカラーの積はベクトルになる

```
float f;  
vec3 v2 = v1 * f; // ベクトルv1の全要素をf倍する
```

- ベクトルとスカラーの和はベクトルになる

```
vec3 v2 = v1 * 2.0 + 1.0; // ベクトルv1の全要素を2倍して1を足す
```

- swizzling (ベクトルの要素の入れ替え)

```
vec4 v1 = vec4(4.0, 3.0, 2.0, 1.0);  
vec3 v2 = v1.wxy; // v2.x←v1.w, v2.y←v1.x, v2.z←v1.y
```

結果

このような画像が表示されれば、多分、正解です。
なお、これは動画です。

