④魚眼レンズ画像の平面展開

床井浩平

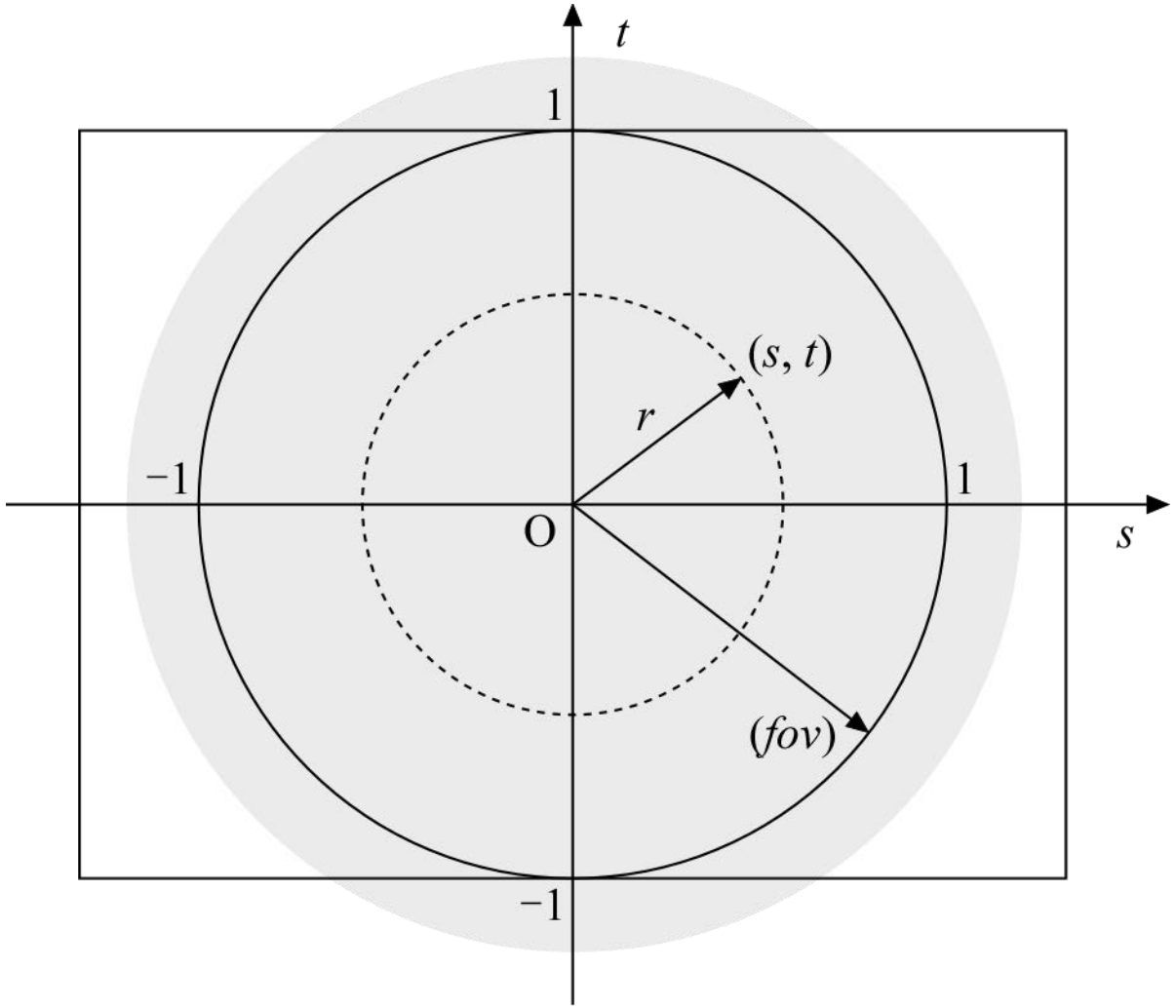
こんなことを話します

- 頂点属性を使わない全画面描画
- 魚眼レンズ画像の平面展開
- ArUco Marker の検出

魚眼レンズ

魚眼レンズ画像のイメージサークル

- 右図は撮像素子上の魚眼レンズの投影像のイメージサークルを表しています
- イメージサークルの中心は撮像素子の中心にあるとします
- fov は撮像素子の短辺と接する円の位置における魚眼レンズの画角です
- 魚眼レンズの上下と左右の画角は一致しているものとします



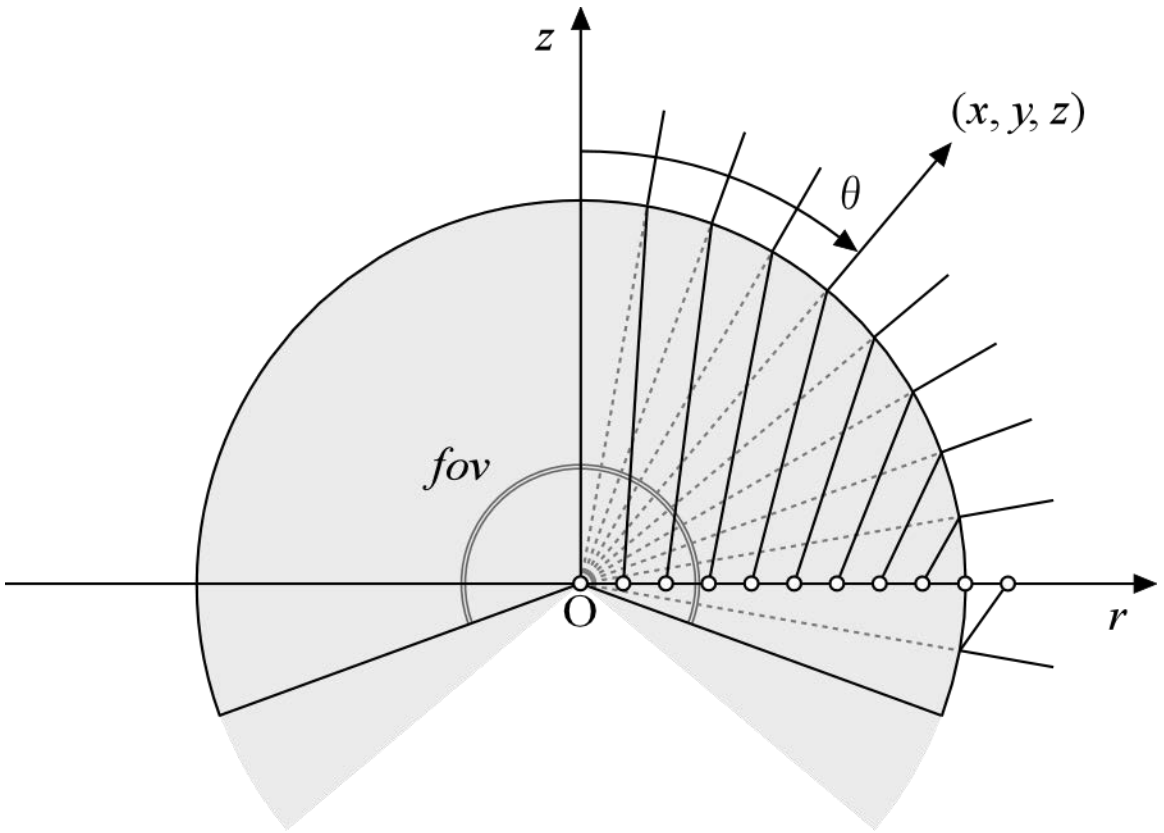
等距離射影方式の魚眼レンズの場合

- 右図は等距離射影方式の魚眼レンズの撮影範囲を横から見たものです
- 天頂からの角度 θ の投影面上の中心からの距離 r は次式で得られます

$$r = \frac{2\theta}{fov}$$

  - fov は魚眼レンズの画角
- レンズの主点から (x, y, z) 方向の点が投影される投影面上の位置 (s, t) は

$$(s, t) = \frac{2\cos^{-1} z}{fov} \frac{(x, y)}{\sqrt{x^2 + y^2}}$$



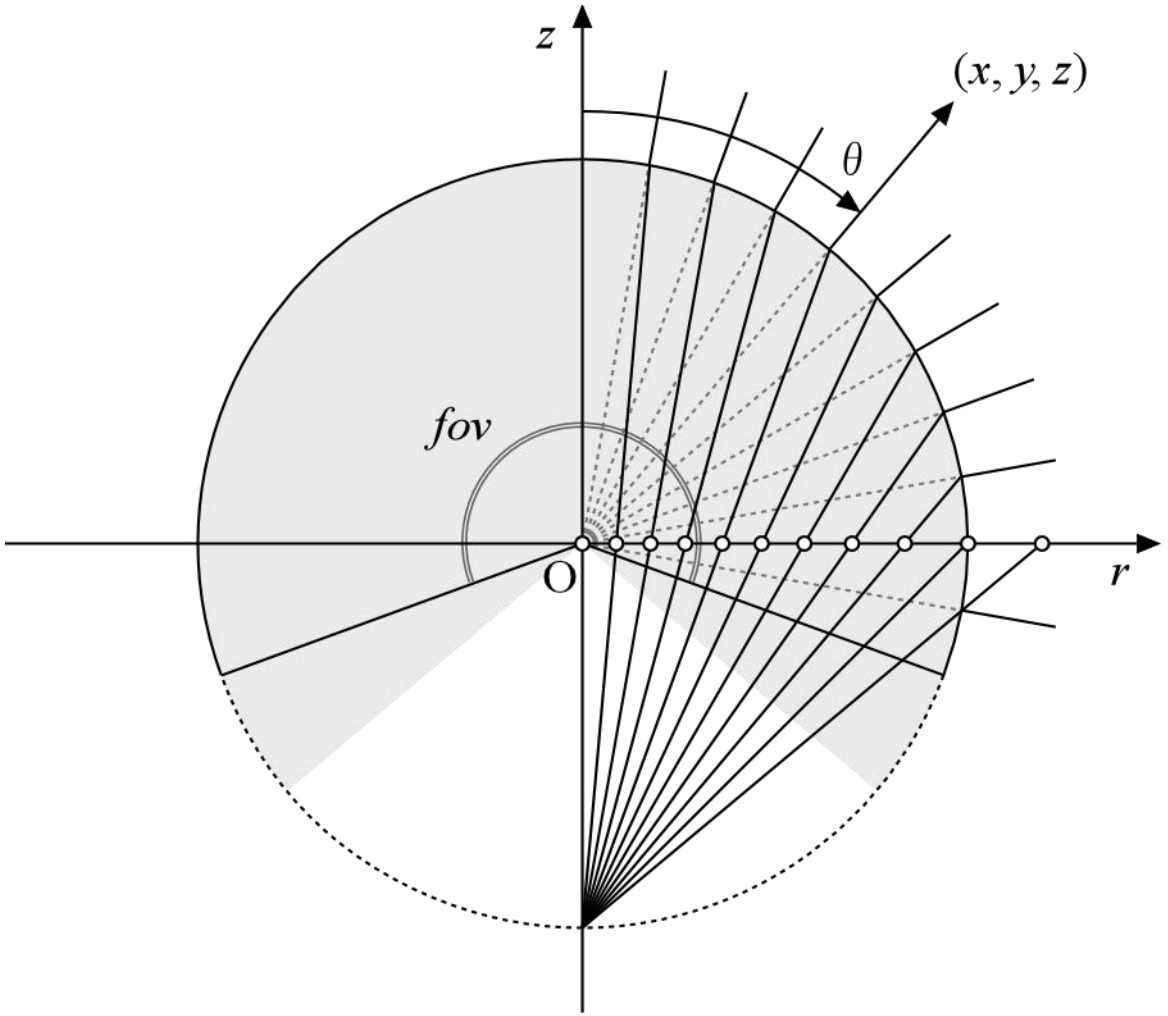
立体射影方式の魚眼レンズの場合

- 右図は立体射影方式の魚眼レンズの撮影範囲を横から見たものです
- 天頂からの角度 θ の投影面上の中心からの距離 r は次式で得られます

$$r = \frac{1 + \cos(fov/2)}{\sin(fov/2)} \frac{\sin\theta}{1 + \cos\theta}$$

  - fov は魚眼レンズの画角
- レンズの主点から (x, y, z) 方向の点が投影される投影面上の位置 (s, t) は

$$(s, t) = \frac{1}{\tan(fov/4)} \frac{(x, y)}{1 + z}$$



平面展開

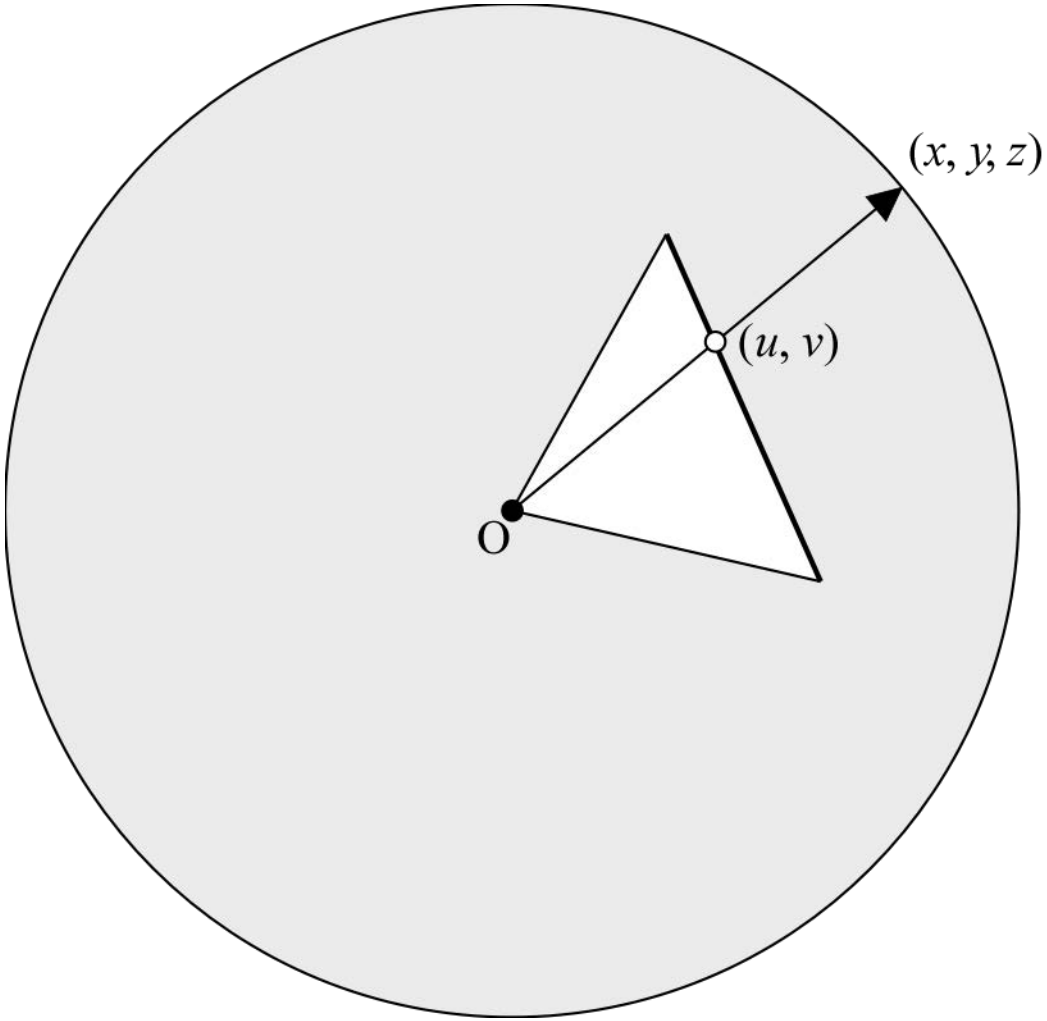
- 原点 O にある視点からスクリーン上の (u, v) に向かう方向を (x, y, z) とする
- (x, y, z) から魚眼レンズ画像上の位置 (s, t) を求める

$$(s,t) = \frac{2\cos^{-1} z}{fov} \frac{(x,y)}{\sqrt{x^2+y^2}}$$

または

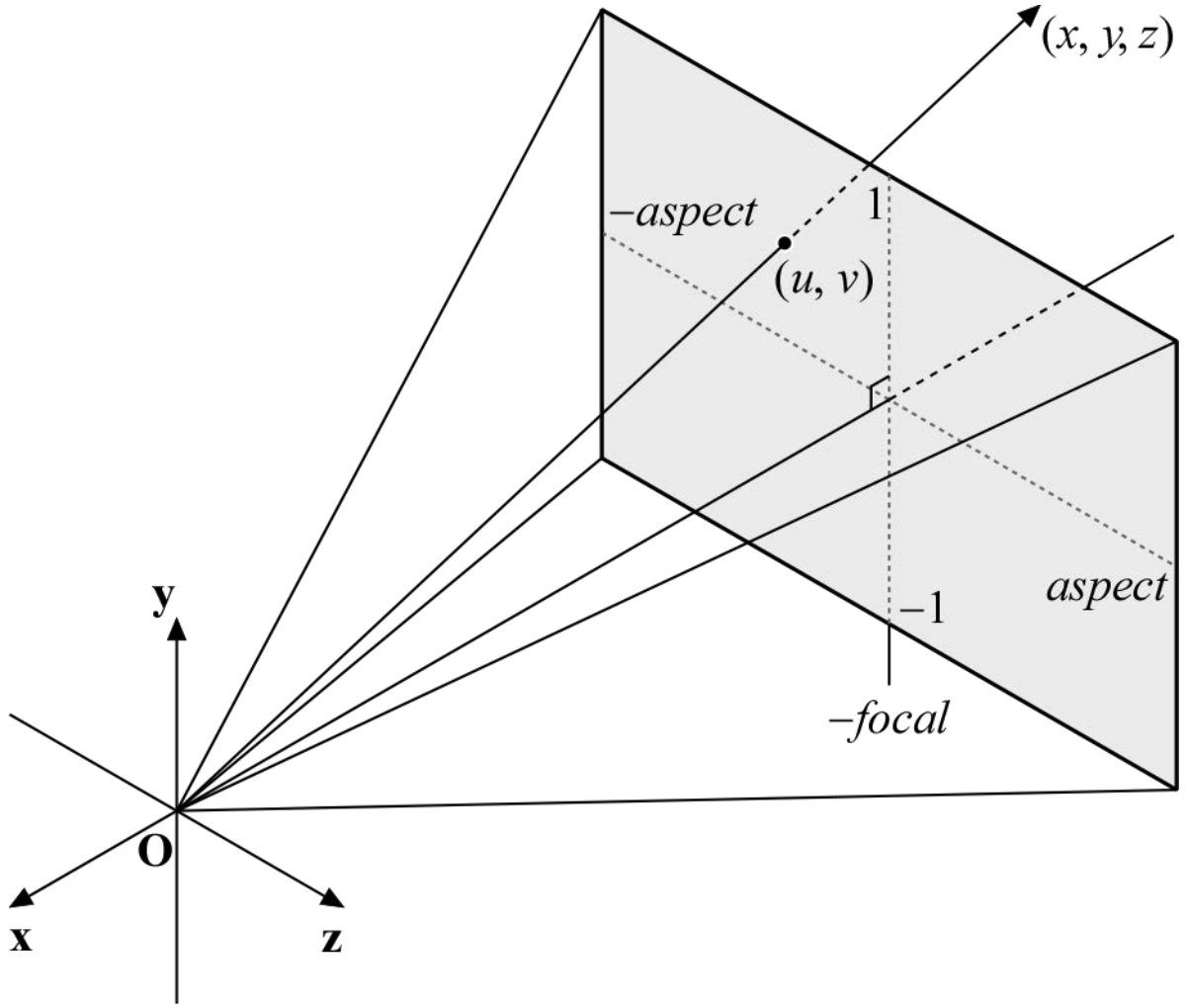
$$(s,t) = \frac{1+\cos(fov/2)}{\sin(fov/2)} \frac{(x,y)}{1+z}$$

- (s, t) の位置の画素の色をサンプリングして (u, v) の位置の点の色に用いる



スクリーン上の点 (u, v) と視線ベクトル (x, y, z)

- スクリーンの大きさは横 $\pm aspect$、縦 ± 1 とします
  - 視線ベクトル (x, y, z) を決めるためにアスペクト比 $aspect$ を用います
  - 通常アスペクト比には表示するウィンドウのものを用いますが、ここでは fbo のアスペクト比を用いています
  - fbo のサイズはキャプチャデバイスの解像度をもとに決定したので、このアスペクト比はキャプチャデバイスのアスペクト比と一致しています



uniform 変数の追加 (ortho.vert)

```
#version 410

// 頂点の位置
layout (location = 0) in vec4 position;

// ウィンドウのアスペクト比
uniform float aspect;

// スクリーンまでの焦点距離
uniform float focal = 1.0;

// スクリーンを回転する変換行列
uniform mat4 rotation;

// テクスチャ座標
out vec3 texcoord;
```

- アスペクト比と焦点距離の uniform 変数を追加します
- スクリーンを回転する変換行列は投影方向を変更するために使います
- テクスチャ座標 texcoord には頂点における視線ベクトル (x, y, z) を格納するので vec3 にします

頂点における視線ベクトルを求める (ortho.vert)

```
void main()
{
  // クリッピング空間いっぱいに描く
  gl_Position = position;

  // aspect × 1.0 のスクリーン上の点の位置
  vec2 uv = position.st * vec2(aspect, 1.0);

  // z = -focal の平面上の点に向かうベクトル
  vec3 xyz = vec3(uv, -focal);

  // 回転の変換行列をかけて視線ベクトルを求める
  texcoord = normalize(mat3(rotation) * xyz);
}
```

- ウィンドウ全体に描画するのでクリッピング空間いっぱいの矩形を描きます
- クリッピング空間の大きさは縦横 ± 1 なので u, v にアスペクト比をかけてスクリーンの大きさに合わせます
- 視線ベクトルは $(u, v, -focal)$ になります
- これに回転の変換行列をかけて正規化します

uniform 変数の追加 (ortho.frag)

```
#version 410

// 画素の色
layout (location = 0) out vec4 color;

// 魚眼レンズの画角
uniform vec2 fov = vec2(3.839724);

// テクスチャ座標に対するスケール
uniform vec2 scale = vec2(0.5, -0.5);

// テクスチャ座標の中心位置
uniform vec2 center = vec2(0.5);

// サンプラー
uniform sampler2D image;

// テクスチャ座標
in vec3 texcoord;
```

- 画角を設定する uniform 変数 fov を追加します（画角 $220° = 220\pi/180$）
- 視線ベクトル (x, y, z) から求めた座標 (s, t) の範囲は $[-1, 1]$ なので、0.5 倍して 0.5 を足して $[0, 1]$ の範囲のテクスチャ座標に直します
- さらにテクスチャの上下を反転します
- 投影像は中心がずれていたり歪んでいたりすることがあるので、これらを uniform 変数 scale と center で設定して実行時に調整可能にします

視線ベクトルでテクスチャをサンプリング (ortho.frag)

```
void main()
{
  // 補間された視線ベクトルを正規化する
  vec3 direction = normalize(texcoord);

  // テクスチャ座標を求める
  vec2 st = normalize(direction.xy)
    * acos(-direction.z) * 2.0 / fov;

  // テクスチャのアスペクト比の逆数
  vec2 size = vec2(textureSize(image, 0));

  // テクスチャのアスペクト比に合わせる
  color = texture(image,
    st * scale * size.y / size + center);
}
```

- テクスチャ座標は線形補間により頂点の位置以外では単位ベクトルで無くなっているため、再度正規化します
- 次式によりテクスチャ座標を求めます

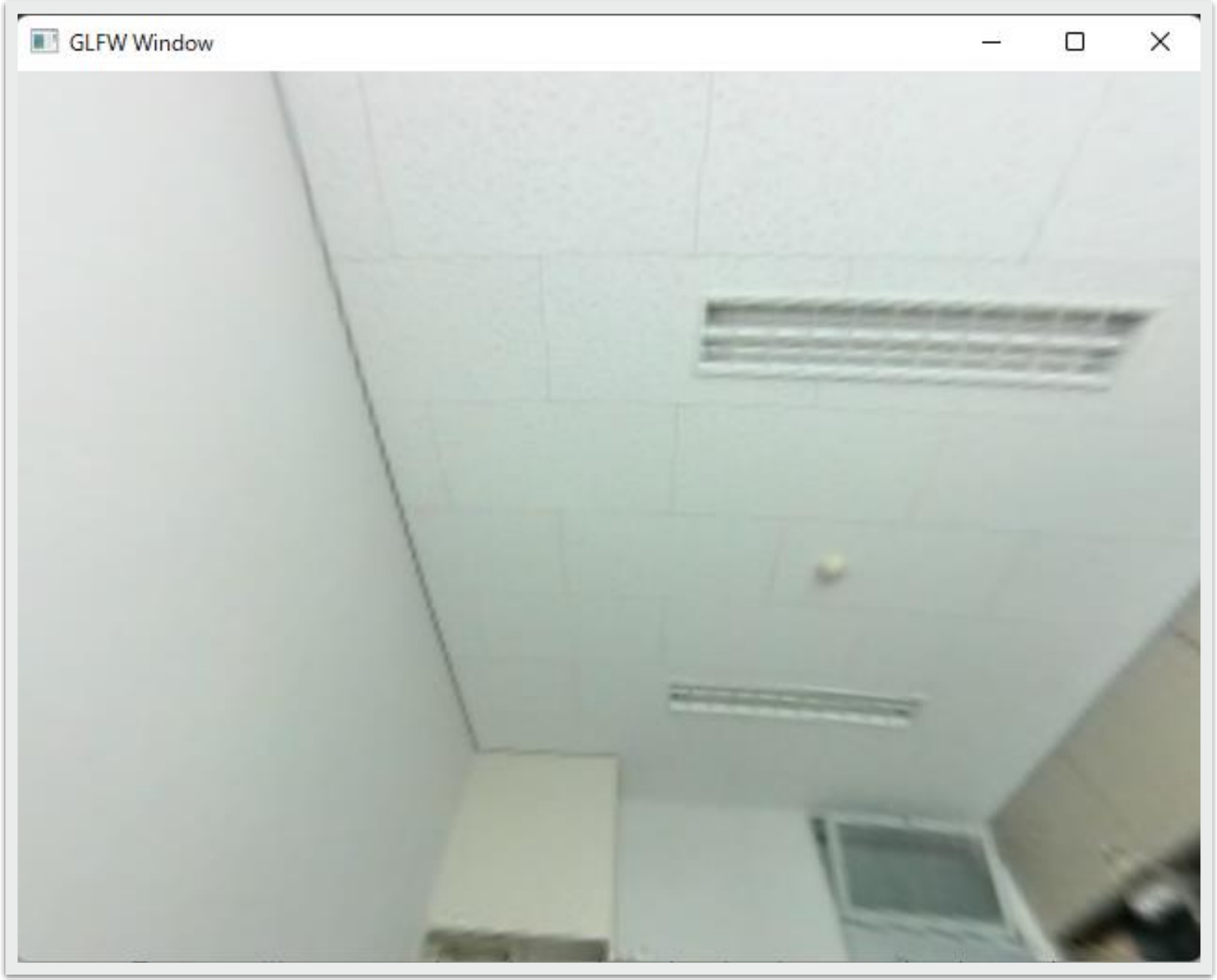
$$(s,t) = \frac{2\cos^{-1} z}{fov}\frac{(x,y)}{\sqrt{x^2+y^2}}$$

- テクスチャは size.x × size.y の大きさなのでテクスチャ座標の $[0,1]$ の範囲にスケーリングします

$$\left(\frac{size.y}{size.x},1\right) = \frac{size.y}{(size.x,size.y)}$$

平面展開した結果

レンズの中心が撮像素子の中心と一致していなかったり正確なレンズの画角が設定されていないなどの理由で歪みが残ります

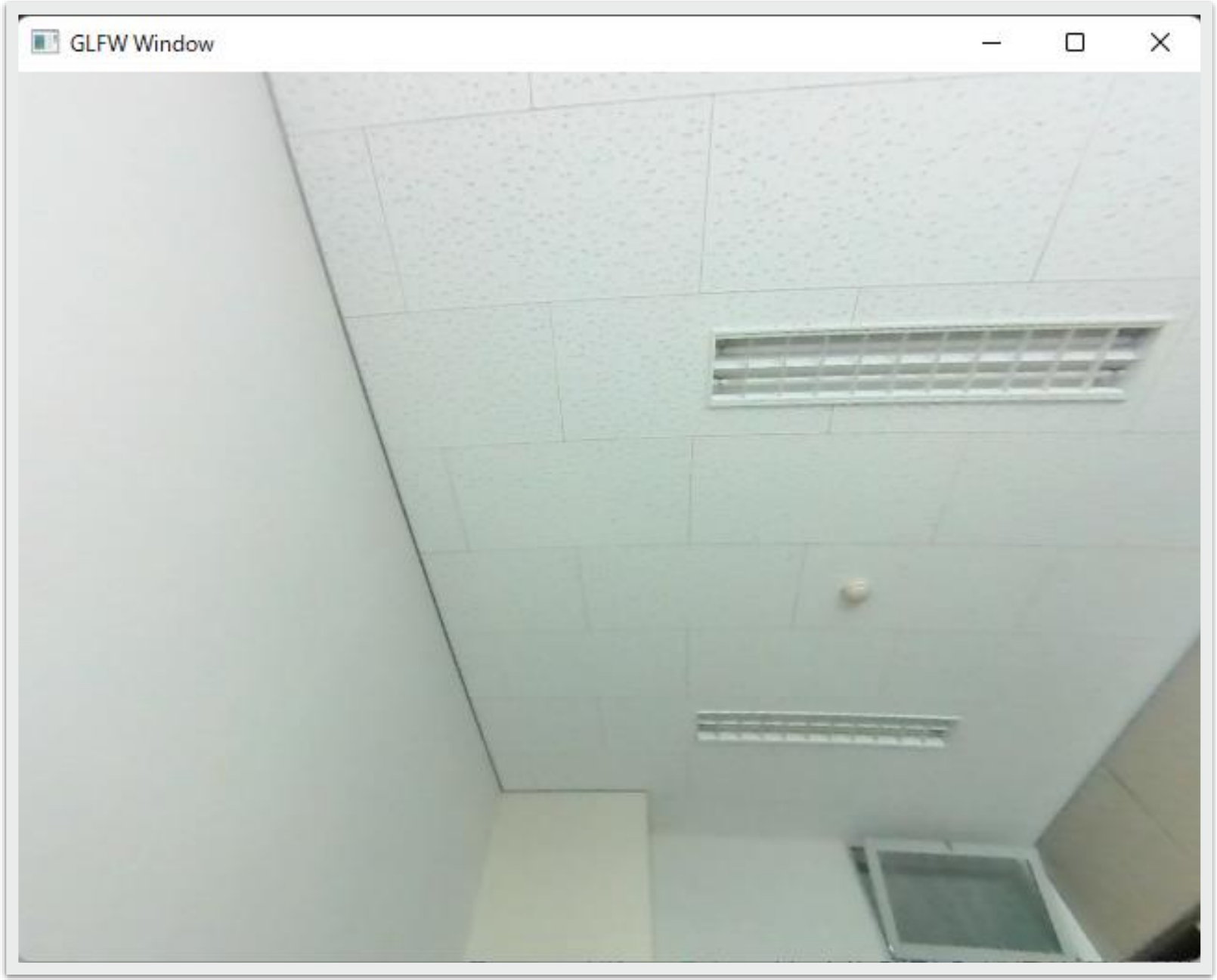


カメラの解像度を調整する (oglcv.cpp)

```
// キャプチャデバイスの準備
cv::VideoCapture camera;
if (!camera.open(0)) throw std::runtime_error("The capture device cannot be used.");
camera.set(cv::CAP_PROP_FRAME_WIDTH, 2592);    // 水平画素数
camera.set(cv::CAP_PROP_FRAME_HEIGHT, 1944);   // 垂直画素数
camera.set(cv::CAP_PROP_FPS, 30);              // フレームレート


// キャプチャデバイスから 1 フレーム取り込む
cv::Mat image;
if (!camera.read(image)) throw std::runtime_error("No frames has been grabbed.");
```

水平画素数・垂直画素数・フレームレートの設定値はカメラ（センサ）に依存します

カメラの解像度を上げた結果

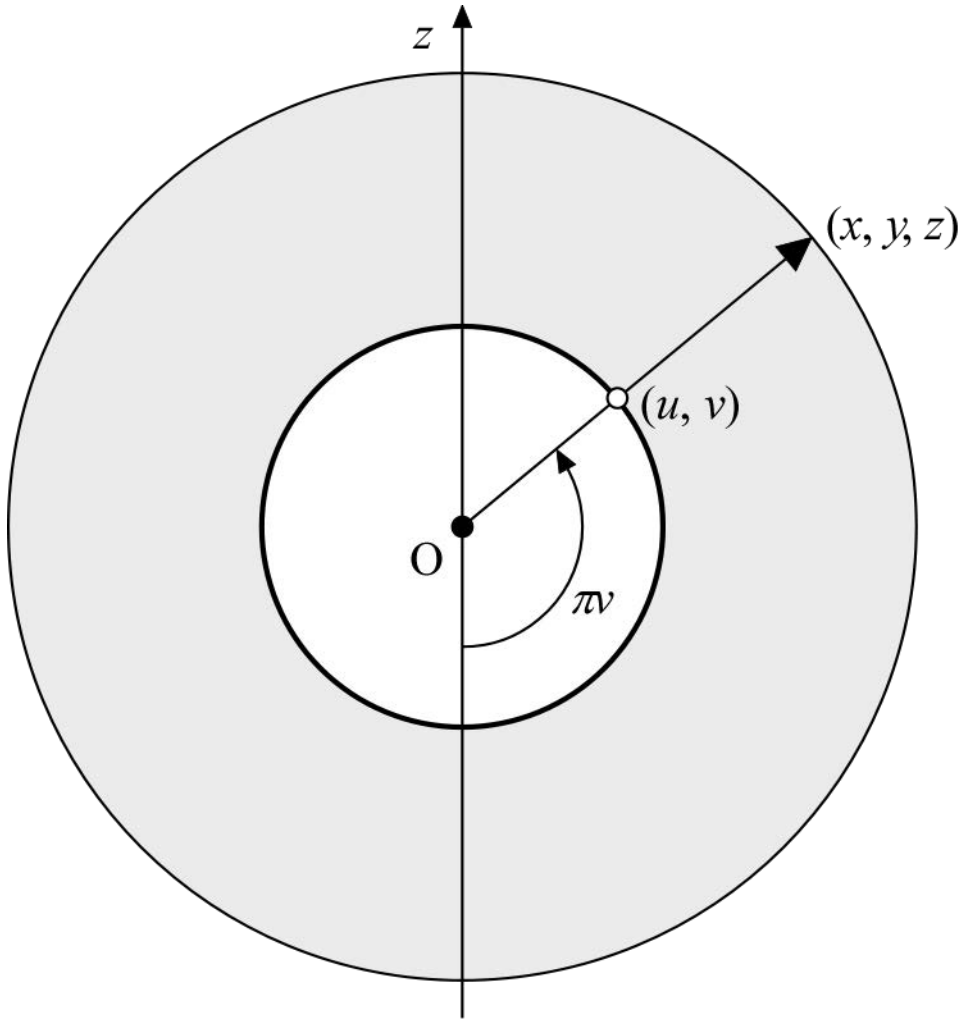


正距円筒図法展開

- 原点 O にある視点から円筒上の (u, v) に向かう方向を (x, y, z) とする

$$\begin{cases} x = \sin \pi v \cos 2\pi u \\ y = \sin \pi v \sin 2\pi u \\ z = -\cos \pi v \end{cases}$$

- この (x, y, z) から魚眼レンズ画像上の位置 (s, t) を求める
- (s, t) の位置の画素の色をサンプリングして (u, v) の位置の点の色に用いる



uniform 変数の追加 (ortho.vert)

```
#version 410

// 頂点の位置
layout (location = 0) in vec4 position;

// ウィンドウのアスペクト比
uniform float aspect;

// スクリーンまでの焦点距離
uniform float focal = 1.0;

// テクスチャ座標
out vec2 texcoord;
```

- アスペクト比と焦点距離の uniform 変数を追加します
- 視線ベクトル (x, y, z) はフラグメントシェーダで求めるのでテクスチャ座標 texcoord は vec2 にします

頂点における視線ベクトルを求める (ortho.vert)

```
void main()
{
 // クリッピング空間いっぱいに描く
  gl_Position = position;

  // 大きさが 2π × π のスクリーン上の点の位置
  texcoord = (position.st * 0.5 + 0.5)
     * vec2(6.283185, 3.141593);
}
```

- ウィンドウ全体に描画するのでクリッピング空間いっぱいの矩形を描きます
- クリッピング空間の大きさは縦横 ± 1 なので u, v を 0.5 倍して 0.5 を足して $[0, 1]$ の範囲にした後、$(2\pi, \pi)$ をかけます

uniform 変数の追加 (ortho.frag)

```
#version 410

// 画素の色
layout (location = 0) out vec4 color;

// 魚眼レンズの画角
uniform vec2 fov = vec2(3.839724);

// テクスチャ座標に対するスケール
uniform vec2 scale = vec2(0.5, -0.5);

// テクスチャ座標の中心位置
uniform vec2 center = vec2(0.5);

// スクリーンを回転する変換行列
uniform mat4 rotation;

// サンプラー
uniform sampler2D image;
```

- 画角を設定する uniform 変数 fov を追加します（画角 $220° = 220\pi/180$）
- 視線ベクトル (x, y, z) から求めた座標 (s, t) の範囲は $[-1, 1]$ なので、0.5 倍して 0.5 を足して $[0, 1]$ の範囲のテクスチャ座標に直します
- さらにテクスチャの上下を反転します
- 投影像は中心がずれていたり歪んでいたりすることがあるので、これらを uniform 変数 scale と center で設定して実行時に調整可能にします

正距円筒図法の視線ベクトル (ortho.frag)

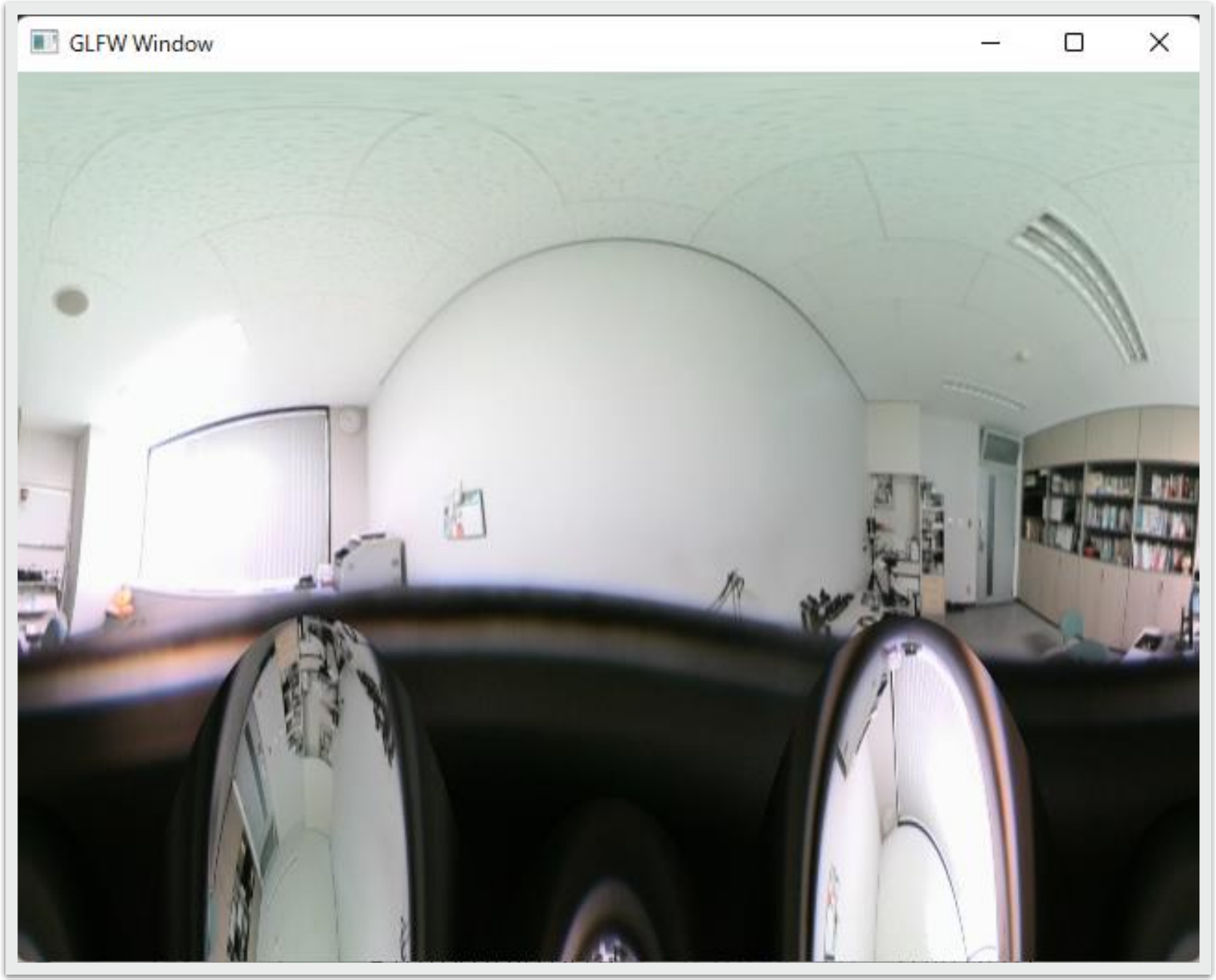
```
// テクスチャ座標
in vec2 texcoord;

void main()
{
  // テクスチャ座標から視線ベクトルを求める
  vec3 direction = mat3(rotation) * vec3(
    sin(texcoord.y) * cos(texcoord.x),
    sin(texcoord.y) * sin(texcoord.x),
    cos(texcoord.y)
  );

  // 以下は平面の場合と同じです
  ...
}
```

- 補間されたテクスチャ座標に対して極座標変換を行って視線ベクトルを求めます

$$\begin{cases} x = \sin \pi v \cos 2\pi u \\ y = \sin \pi v \sin 2\pi u \\ z = -\cos \pi v \end{cases}$$

正距円筒図法展開

波打つ曲線はイメージサークルの外周で直線に近づけるにはテクスチャ座標の中心位置＝画像上のイメージサークルの中心の位置を調整します

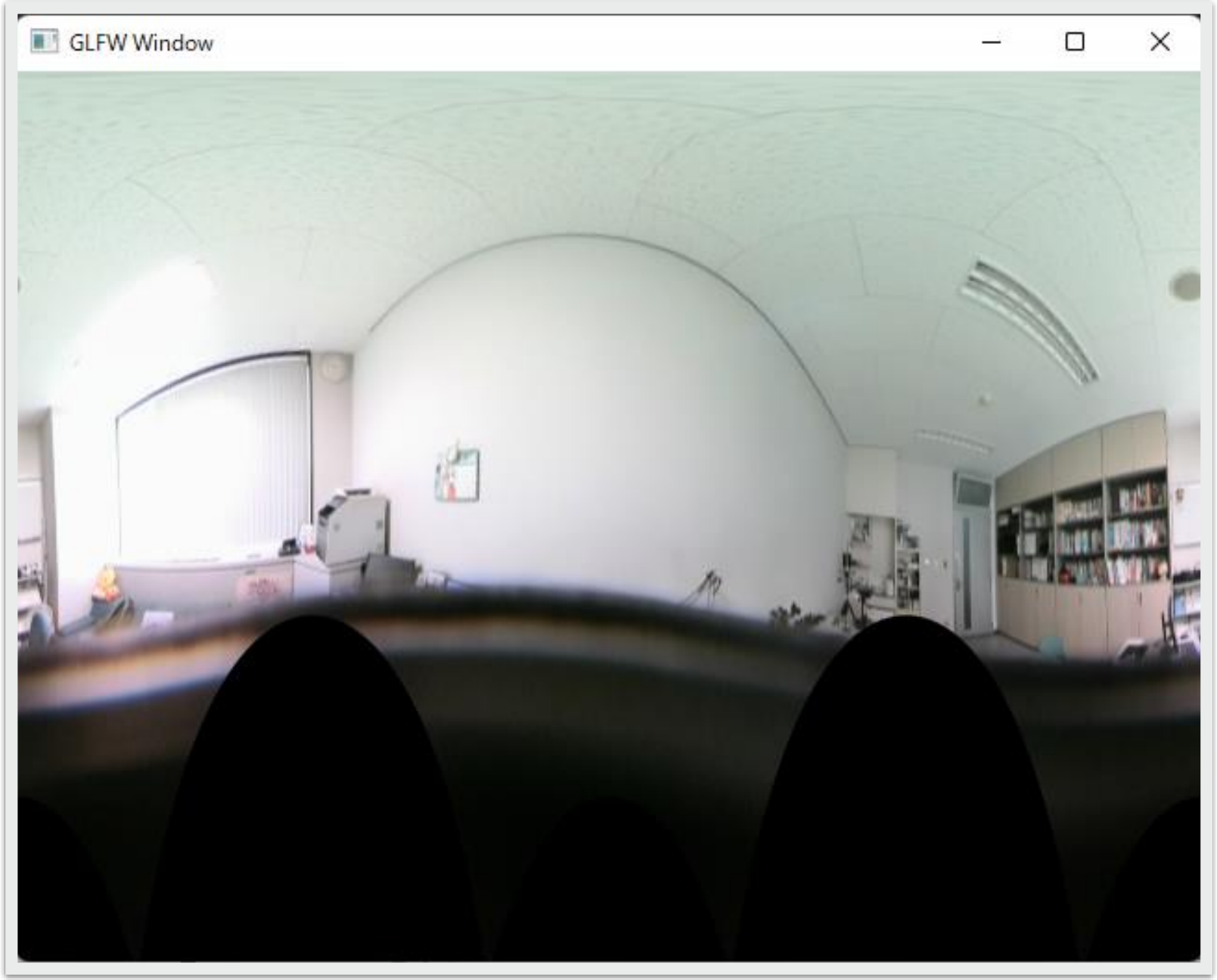


テクスチャ空間の外は境界色を拡張する (oglcv.cpp)

```
// テクスチャの準備
std::array<GLuint, 2> textures;
glGenTextures(textures.size(), textures.data());
for (const auto texture : textures)
{
  glBindTexture(GL_TEXTURE_2D, texture);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image.cols, image.rows, 0,
    GL_BGR, GL_UNSIGNED_BYTE, image.data);

  // テクスチャをサンプリングする方法の指定
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

  // テクスチャ空間を外れた場合は境界色を拡張する
  constexpr GLfloat borderColor[] = { 0.0f, 0.0f, 0.0f, 0.0f };
  glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
}
```

境界色を拡張した結果

テクスチャ空間の外は境界色になります

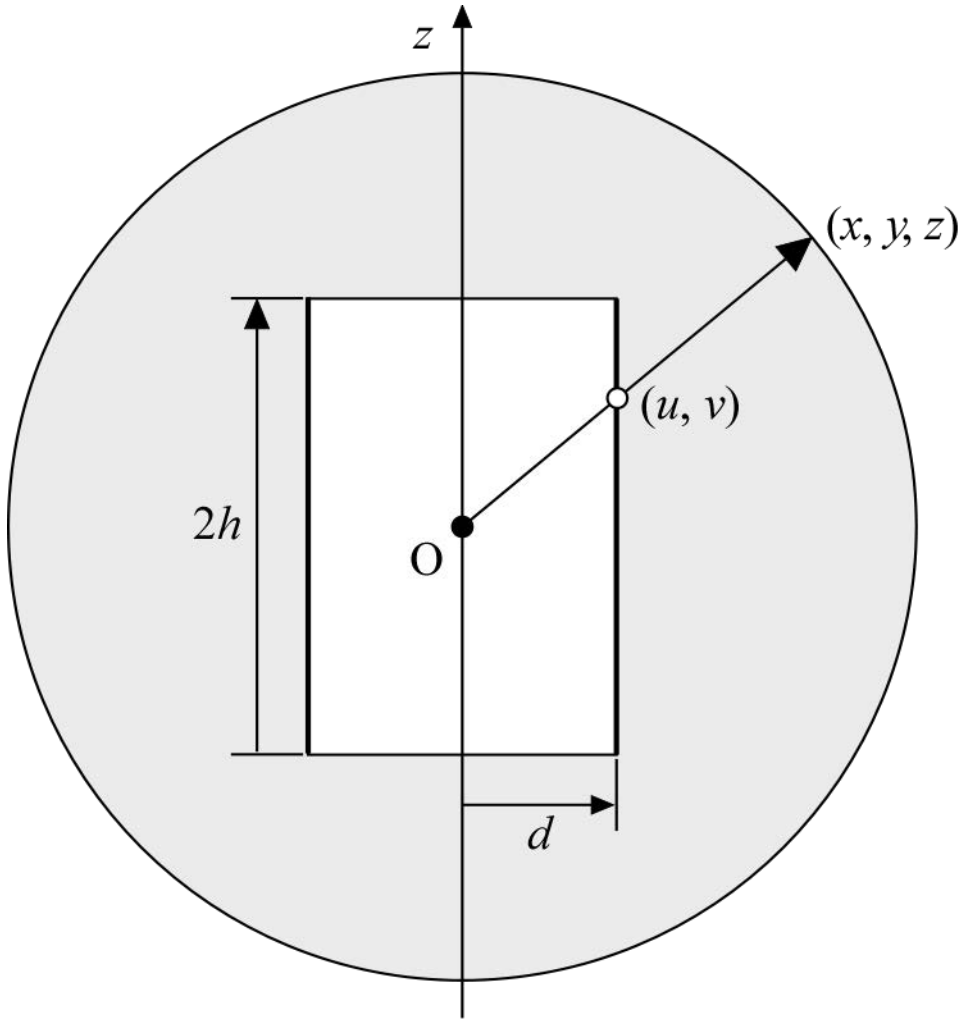


メルカトル図法展開

- 原点 O にある視点から円筒上の (u, v) に向かう方向を (x, y, z) とする

$$\begin{cases} x = d\cos 2\pi u \\ y = d\sin 2\pi u \\ z = (2v - 1)h \end{cases}$$

- この (x, y, z) から魚眼レンズ画像上の位置 (s, t) を求める
- (s, t) の位置の画素の色をサンプリングして (u, v) の位置の点の色に用いる



uniform 変数の追加 (ortho.vert)

```
#version 410

// 頂点の位置
layout (location = 0) in vec4 position;

// ウィンドウのアスペクト比
uniform float aspect;

// スクリーンまでの焦点距離
uniform float focal = 1.0;

// テクスチャ座標
out vec2 texcoord;
```

- アスペクト比と焦点距離の uniform 変数を追加します
- 視線ベクトル (x, y, z) はフラグメントシェーダで求めるのでテクスチャ座標 texcoord は vec2 にします

頂点における視線ベクトルを求める (ortho.vert)

```
void main()
{
 // クリッピング空間いっぱいに描く
  gl_Position = position;

  // 大きさが 2π × 1 のスクリーン上の点の位置
  texcoord = (position.st * 0.5 + 0.5)
     * vec2(6.283185, 1.0);
}
```

- ウィンドウ全体に描画するのでクリッピング空間いっぱいの矩形を描きます
- クリッピング空間の大きさは縦横 ± 1 なので u, v を 0.5 倍して 0.5 を足して $[0, 1]$ の範囲にした後、$(2\pi, 1)$ をかけます

uniform 変数の追加 (ortho.frag)

```
#version 410

// 画素の色
layout (location = 0) out vec4 color;

// 魚眼レンズの画角
uniform vec2 fov = vec2(3.839724);

// テクスチャ座標に対するスケール
uniform vec2 scale = vec2(0.5, -0.5);

// テクスチャ座標の中心位置
uniform vec2 center = vec2(0.5);

// スクリーンを回転する変換行列
uniform mat4 rotation;

// サンプラー
uniform sampler2D image;
```

- 画角を設定する uniform 変数 fov を追加します（画角 $220° = 220\pi/180$）
- 視線ベクトル (x, y, z) から求めた座標 (s, t) の範囲は $[-1, 1]$ なので、0.5 倍して 0.5 を足して $[0, 1]$ の範囲のテクスチャ座標に直します
- さらにテクスチャの上下を反転します
- 投影像は中心がずれていたり歪んでいたりすることがあるので、これらを uniform 変数 scale と center で設定して実行時に調整可能にします

メルカトル図法の視線ベクトル (ortho.frag)

```
// テクスチャ座標
in vec2 texcoord;

void main()
{
  // テクスチャ座標から視線ベクトルを求める
  vec3 direction = mat3(rotation) * vec3(
    cos(texcoord.x),          // d = 1
    sin(texcoord.x),          // d = 1
    1.0 - texcoord.y * 2.0  // h = -1
  );

  // 以下は平面の場合と同じです
  ...
}
```

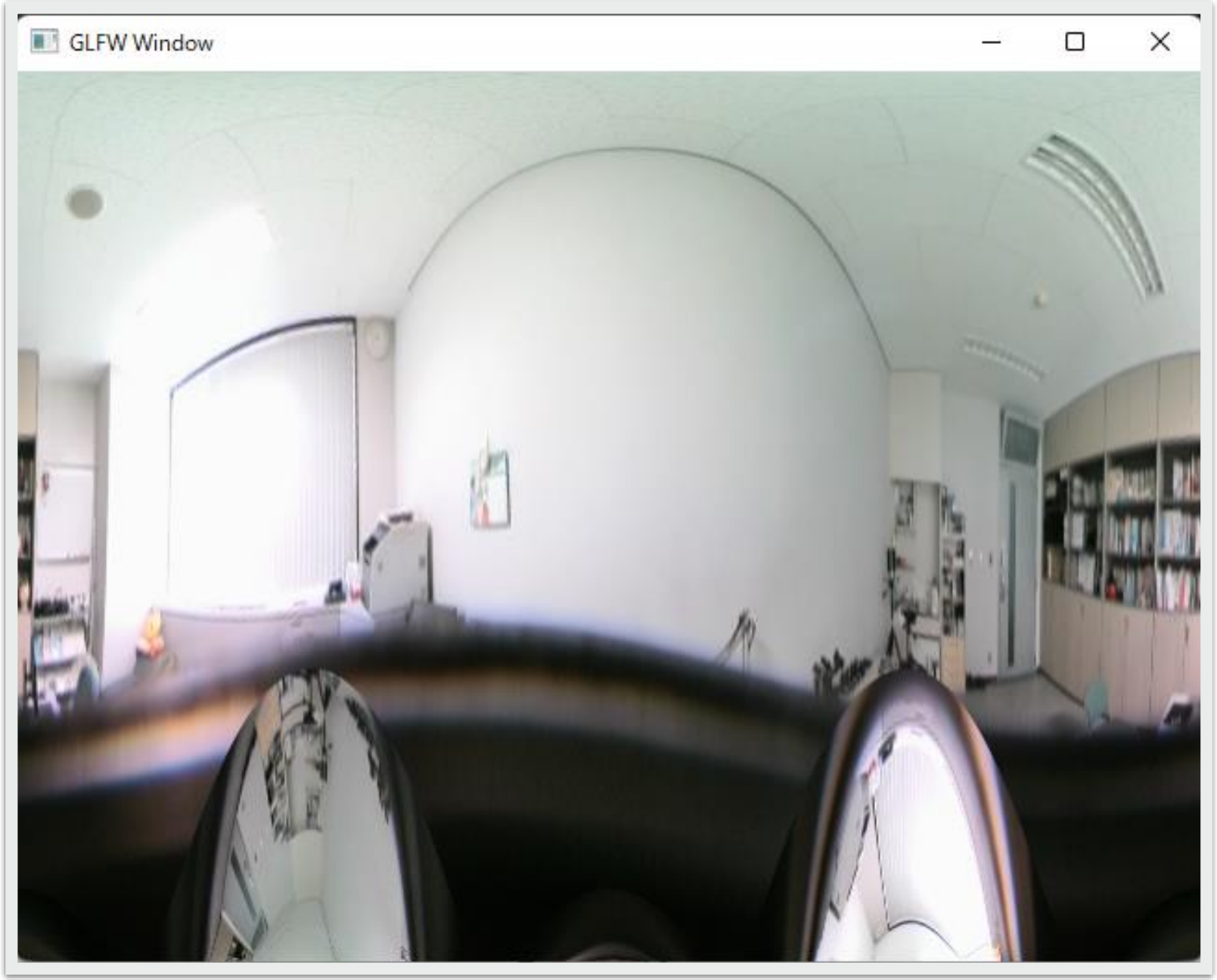
- 補間されたテクスチャ座標に対して極座標変換を行って視線ベクトルを求めます

$$\begin{cases} x = d\cos 2\pi u \\ y = d\sin 2\pi u \\ z = (2v-1)h \end{cases}$$

- ここでは $d = 1$ としテクスチャ座標の上下を反転するために $h = -1$ としています

メルカトル図法展開

真上は写りません



頂点番号 gl_VertexID から座標値を求める (ortho.vert)

```
#version 410

// 頂点の位置
layout (location = 0) in vec4 position;

... （中略）

void main()
{
  // 頂点番号から座標値を求める
  int s = gl_VertexID & ~1;
  int t = (~gl_VertexID & 1) << 1;
  vec2 position = vec2(s, t) - 1.0;

  // クリッピング空間いっぱいに描く
  gl_Position = vec4(position, 0.0, 1.0);

  // 大きさがaspect×1.0のスクリーン上の点の位置
  vec2 uv = position * vec2(aspect, 1.0);
```

- gl_VertexID に格納されている頂点番号を使って頂点の座標値を決定します
- したがって in 変数 (attribute 変数) の position は不要になります
- 頂点の位置 position は次表の値になるようにします

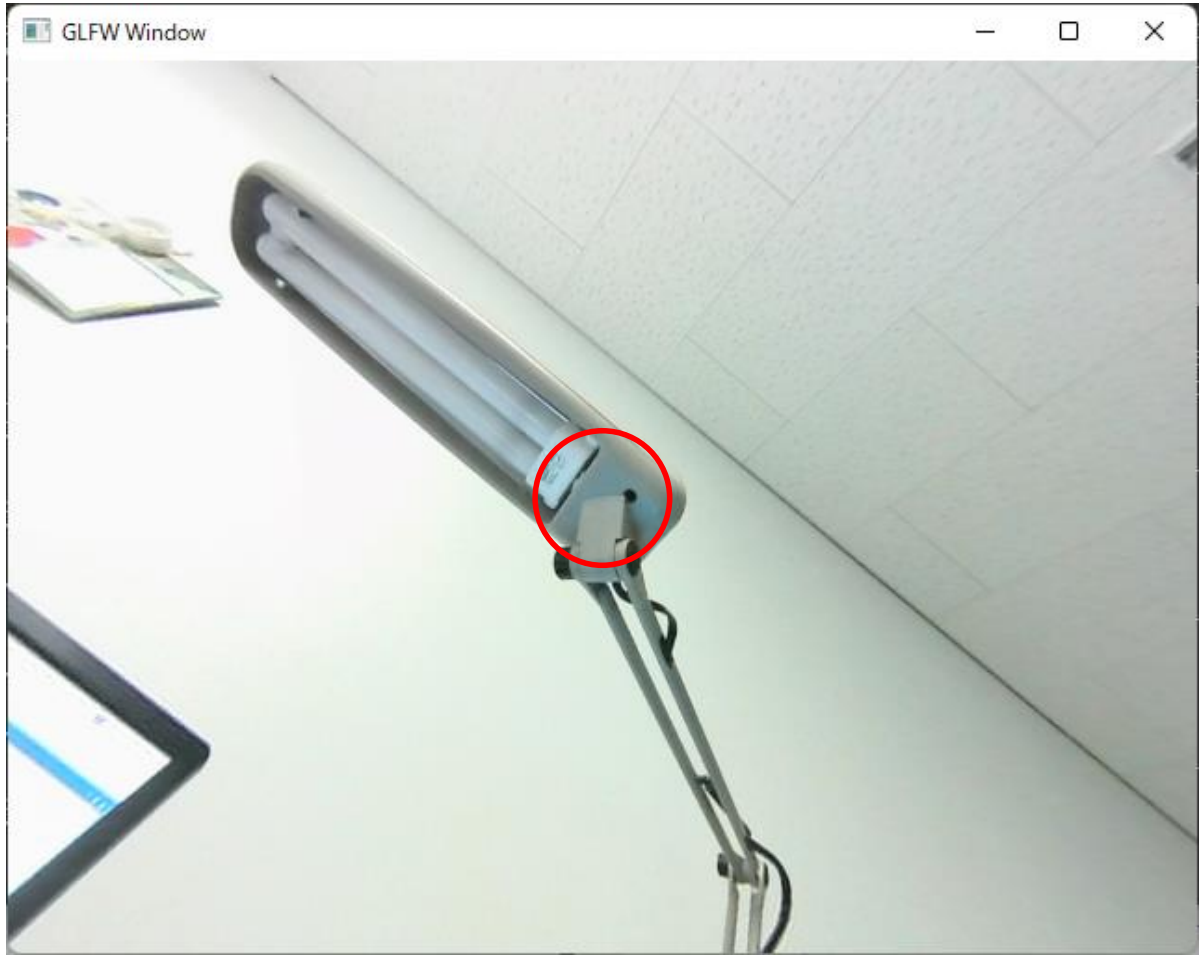
gl_VertexID	0	1	2	3
position.s	-1	-1	1	1
position.t	1	-1	1	-1

頂点バッファオブジェクトも不要 (oglcv.cpp)

```
  // 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
  GLuint vbo;
  glGenBuffers(1, &vbo);
  glBindBuffer(GL_ARRAY_BUFFER, vbo);

... （中略）

  // 頂点配列オブジェクト完成
  glBindVertexArray(0);

  // プログラムオブジェクトの作成
  const GLuint program{ ggLoadShader("ortho.vert", "ortho.frag") };
```

VAO を作るだけで済みます

等距離射影方式のレンズでは中央部が歪むことがある



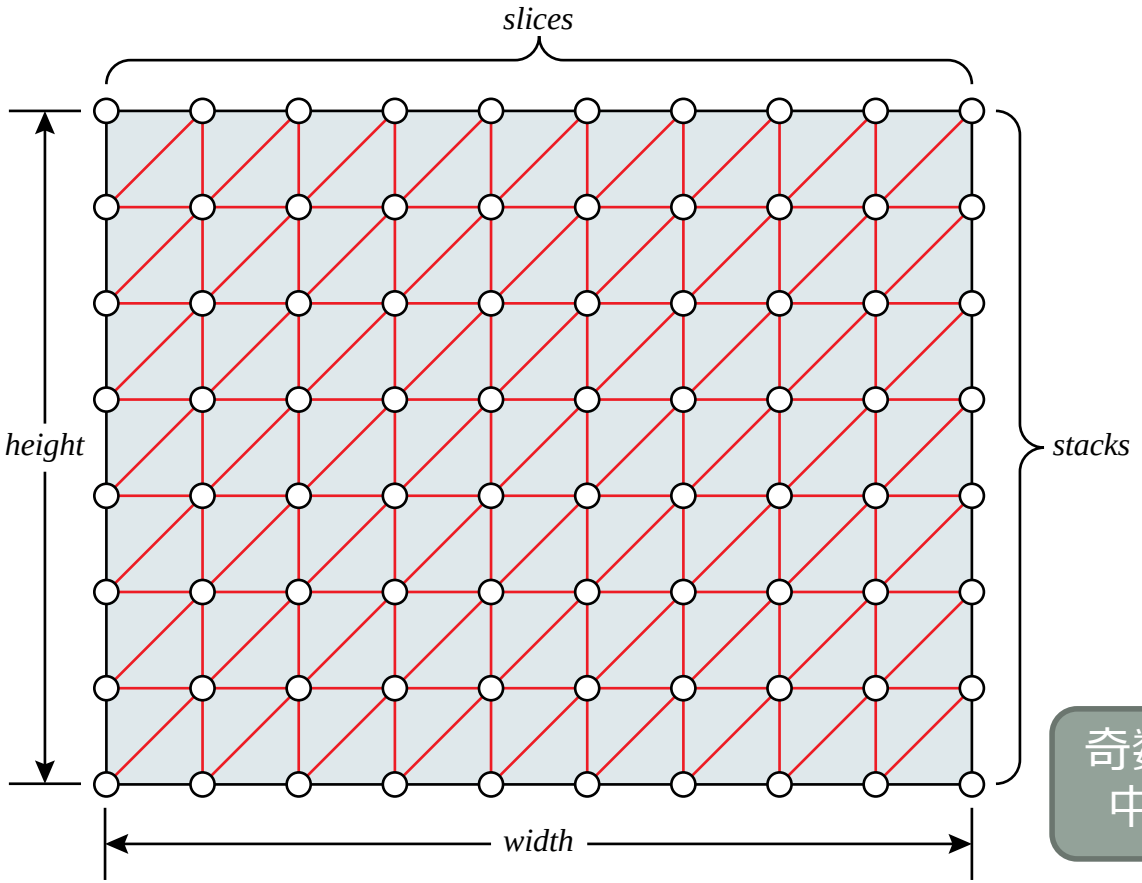
- スクリーンの中心からスクリーン上の各点 (x, y) に向かうベクトルを正規化しています

$$(s, t) = \frac{2\cos^{-1} z}{fov} \frac{(x, y)}{\sqrt{x^2 + y^2}}$$

- スクリーンの中心付近ではベクトルの長さが 0 に近くなり計算誤差が出ます
  - GPU やドライバに依存します
- また、この計算をディスプレイ上（フレームバッファオブジェクト）の全画素に対して実行するのは高コスト

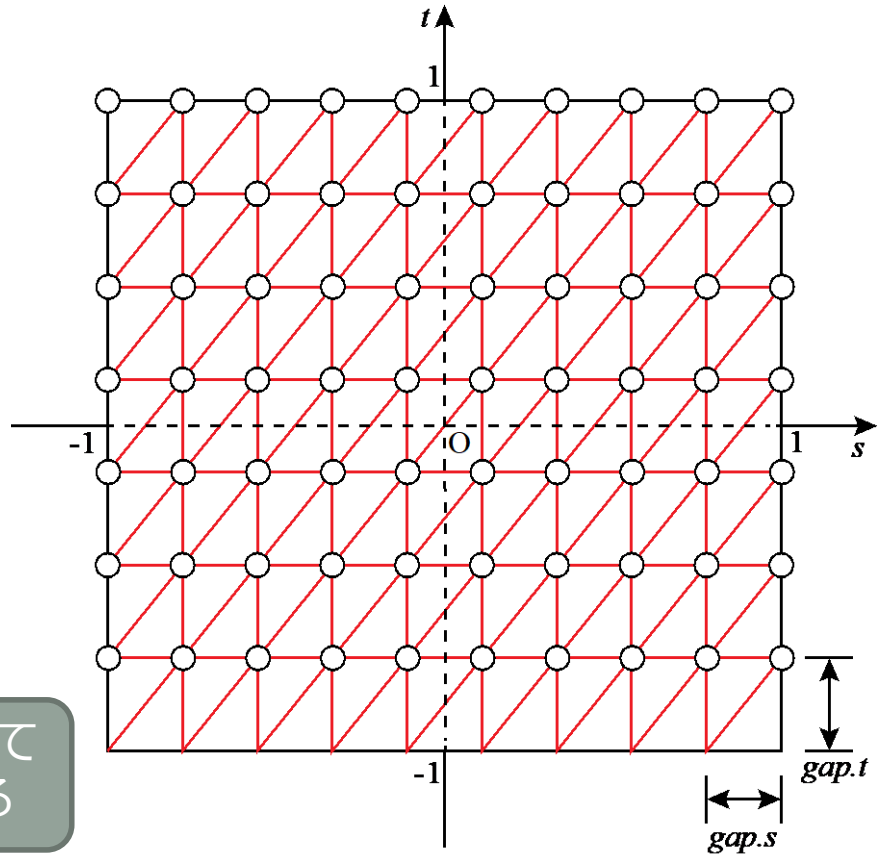
ウィンドウいっぱいに描画する矩形を分割する

ウィンドウ



奇数に分割して
中央を避ける

クリッピング空間



分割した矩形の頂点の座標値を求める (ortho.vert)

```
// スクリーンを回転する変換行列
uniform mat4 rotation;

// 分割数
uniform int slices = 41, stacks = 31;

// テクスチャ座標
out vec3 texcoord;

void main()
{
  // 頂点番号から座標値を求める
  int s = gl_VertexID & ~1;
  int t = (~gl_VertexID & 1) + gl_InstanceID << 1;
  vec2 position = vec2(s, t) / vec2(slices, stacks) - 1.0;

  // クリッピング空間いっぱいに描く
  gl_Position = vec4(position, 0.0, 1.0);
```

後で CPU 側から値を設定できるように uniform 変数にして初期値を与えています

分割した矩形をレンダリングする (oglcv.cpp)

```
// 頂点配列オブジェクトの指定
glBindVertexArray(vao);

// レンダーターゲットの指定
glDrawBuffers(std::size(bufs), bufs);

// 図形の描画
glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, 41 * 2 + 2, 31);

// 標準のフレームバッファへの転送
glBindFramebuffer(GL_READ_FRAMEBUFFER, fbo);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT);
GLsizei w{ window.getWidth() };
GLsizei h{ window.getHeight() };
GLsizei x{ w / 2 };
GLsizei y{ h / 2 };
float t{ h * aspect };
```

slices

stacks

この段階でも描画はできますが
問題は解決していません

フラグメントシェーダの座標計算を削除 (ortho.frag)

```
#version 410

// 画素の色
layout (location = 0) out vec4 color;

// 魚眼レンズの画角
uniform vec2 fov = vec2(3.839724);

// テクスチャ座標に対するスケール
uniform vec2 scale = vec2(0.5, -0.5);

// テクスチャ座標の中心位置
uniform vec2 center = vec2(0.5);

// サンプラー
uniform sampler2D image;
```

バーテックスシェーダに移動

```
// テクスチャ座標
in vec2 texcoord;

void main()
{
  // 補間された視線ベクトルを正規化する
  vec3 direction = normalize(texcoord);

  // テクスチャ座標を求める
  vec2 st = normalize(direction.xy) * acos(-direction.z) * 2.0 / fov;

  // テクスチャのアスペクト比の逆数
  vec2 size = vec2(textureSize(image, 0));

  // 補間されたテクスチャ座標でサンプリングする
  color = texture(image, texcoord);
}
```

バーテックスシェーダに移動

バーテックスシェーダに座標計算を追加 (ortho.vert)

```
#version 410

// 魚眼レンズの画角
uniform vec2 fov = vec2(3.839724);

// テクスチャ座標に対するスケール
uniform vec2 scale = vec2(0.5, -0.5);

// テクスチャ座標の中心位置
uniform vec2 center = vec2(0.5);

// サンプラー
uniform sampler2D image;

// ウィンドウのアスペクト比
uniform float aspect;

... （中略）
```

フラグメントシェーダから移動

```
  // 大きさが aspect × 1.0 のスクリーン上の点の位置
  vec2 uv = position * vec2(aspect, 1.0);

  // 右手系で原点から z = -focal の位置にあるその点に向かうベクトル
  vec3 xyz = vec3(uv, -focal);

  // 原点から z = -focal の位置にある点に向かうベクトル
  vec3 xyz = vec3(uv, -focal);

  // 回転の変換行列をかけて視線ベクトルを求める
  vec3 direction = normalize(mat3(rotation) * xyz);

  // テクスチャ座標を求める
  vec2 st = normalize(direction.xy) * acos(-direction.z) * 2.0 / fov;

  // テクスチャのアスペクト比の逆数
  vec2 size = vec2(textureSize(image, 0));

  // テクスチャ座標を求める
  texcoord = st * scale * size.y / size + center;
}
```

フラグメントシェーダから移動