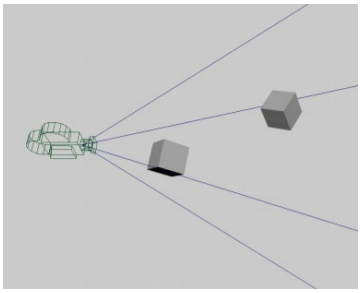
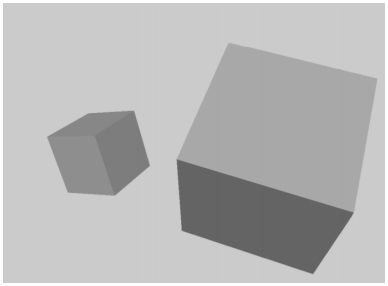
描画エンジン開発 EX

○概要

3D 空間に箱や球などの基本形状を描画します。
3D の描画には「カメラ」の概念があり、3D 空間のオブジェクトをカメラから見た光景として画面に表示されるまでの流れを理解しましょう。



3D 空間にカメラと
オブジェクトが
表示されている



カメラから見た世界が
画面に表示されている

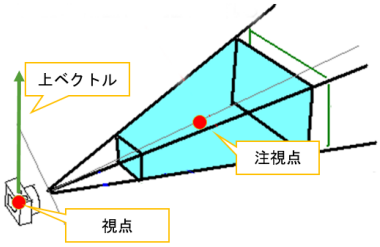
○カメラ

カメラはビュー行列とプロジェクション行列という二つの変換行列を組み合わせることにより、カメラから見た世界を画面に表示することができます。

ビュー行列とプロジェクション行列は以下のパラメータによって計算できます。

ビュー行列

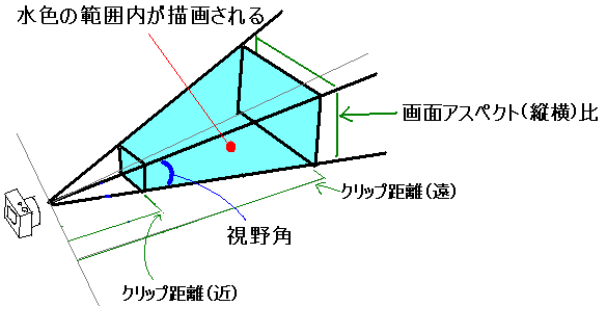
- 視点
- 注視点
- 上ベクトル



この３つのパラメータにより、カメラがどこにいて、どっちを向いているかを表現できます。

プロジェクション行列

- 視野角
- 画面比率
- クリップ距離（遠）
- クリップ距離（近）



このの４つのパラメータによりカメラが見ている世界の範囲を制限して画面に表示するかを表現できます。

カメラ関連の制御をするための Camera クラスを作成します。
Camera.cpp と Camera.h を作成しましょう。

Camera.h

```
#pragma once

#include <DirectXMath.h>

// カメラ
class Camera
{
public:
    // 指定方向を向く
    void SetLookAt(const DirectX::XMFLOAT3& eye, const DirectX::XMFLOAT3& focus, const DirectX::XMFLOAT3& up);

    // パースペクティブ設定
    void SetPerspectiveFov(float fovY, float aspect, float nearZ, float farZ);

    // ビュー行列取得
    const DirectX::XMFLOAT4X4& GetView() const { return view; }

    // プロジェクション行列取得
    const DirectX::XMFLOAT4X4& GetProjection() const { return projection; }

    // 視点取得
    const DirectX::XMFLOAT3& GetEye() const { return eye; }

    // 注視点取得
    const DirectX::XMFLOAT3& GetFocus() const { return focus; }

    // 上方向取得
    const DirectX::XMFLOAT3& GetUp() const { return up; }

    // 前方向取得
    const DirectX::XMFLOAT3& GetFront() const { return front; }

    // 右方向取得
    const DirectX::XMFLOAT3& GetRight() const { return right; }

private:
    DirectX::XMFLOAT4X4   view;
    DirectX::XMFLOAT4X4   projection;

    DirectX::XMFLOAT3     eye;
    DirectX::XMFLOAT3     focus;

    DirectX::XMFLOAT3     up;
    DirectX::XMFLOAT3     front;
    DirectX::XMFLOAT3     right;
};
```

Camera.cpp

```
#include "Camera.h"

// 指定方向を向く
void Camera::SetLookAt(const DirectX::XMFLOAT3& eye, const DirectX::XMFLOAT3& focus, const DirectX::XMFLOAT3& up)
{
    // 視点、注視点、上方向からビュー行列を作成
    DirectX::XMVECTOR Eye = DirectX::XMLoadFloat3(&eye);
    DirectX::XMVECTOR Focus = DirectX::XMLoadFloat3(&focus);
    DirectX::XMVECTOR Up = DirectX::XMLoadFloat3(&up);
    DirectX::XMMATRIX View = DirectX::XMMatrixLookAtLH(Eye, Focus, Up);
    DirectX::XMStoreFloat4x4(&view, View);

    // ビューを逆行列化し、ワールド行列に戻す
    DirectX::XMMATRIX World = DirectX::XMMatrixInverse(nullptr, View);
    DirectX::XMFLOAT4X4 world;
    DirectX::XMStoreFloat4x4(&world, World);

    // カメラの方向を取り出す
    this->right.x = world._11;
    this->right.y = world._12;
    this->right.z = world._13;

    this->up.x = world._21;
    this->up.y = world._22;
    this->up.z = world._23;

    this->front.x = world._31;
    this->front.y = world._32;
    this->front.z = world._33;

    // 視点、注視点を保存
    this->eye = eye;
    this->focus = focus;
}

// パースペクティブ設定
void Camera::SetPerspectiveFov(float fovY, float aspect, float nearZ, float farZ)
{
    // 画角、画面比率、クリップ距離からプロジェクション行列を作成
    DirectX::XMMATRIX Projection = DirectX::XMMatrixPerspectiveFovLH(fovY, aspect, nearZ, farZ);
    DirectX::XMStoreFloat4x4(&projection, Projection);
}
```

ビュー行列とはカメラのワールド行列を
逆行列化した行列。
逆にビュー行列を逆用列化すると
カメラのワールド行列に戻る。

カメラのワールド行列から
上、右、前方向の情報を取り出す。

$$\begin{bmatrix} \_11 & \_12 & \_13 & \_14 \\ \_21 & \_22 & \_23 & \_24 \\ \_31 & \_32 & \_33 & \_34 \\ \_41 & \_42 & \_43 & \_44 \end{bmatrix}$$

X軸ベクトル
Y軸ベクトル
Z軸ベクトル
位置

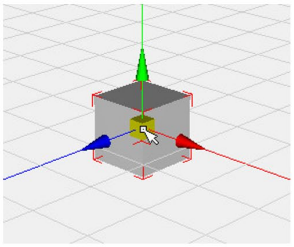
○ワールド行列

オブジェクトを世界に配置する際には「位置」「回転」「スケール」を元に行列を計算します。
この行列のことをワールド行列と呼び、ビュー行列とプロジェクション行列と掛け合わせることによりオブジェクトが画面上のどこに描画されるかを計算できます。

ワールド行列は「行」ごとに特性を持っており、１～３行目はＸＹＺ軸ベクトルとして表現され、４行目は位置として表現されています。

$$\begin{bmatrix} Xx & Xy & Xz & 0 \\ Yx & Yy & Yz & 0 \\ Zx & Zy & Zz & 0 \\ Px & Py & Pz & 1 \end{bmatrix}$$

X 軸
Y 軸
Z 軸
位置



```
DirectX::XMFLOAT4X4 transform =
{
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1
};
```

スケール&回転
位置

○ギズモ

ギズモとは直訳すると「仕掛け」や「装置」という意味ですが、ここではデバッグ用にオブジェクトの位置や衝突範囲などを可視化するための機能という意味合いで使用していきます。

今回は箱などの 3D 空間に配置するメッシュを作成し、任意の位置にメッシュを描画する Gizmos クラスを作成します。
Gizmos.cpp と Gizmos.h を作成しましょう。

Gizmos.h

```
#pragma once

#include <vector>
#include <wrl.h>
#include <d3d11.h>
#include <DirectXMath.h>

class Gizmos
{
public:
    Gizmos(ID3D11Device* device);
    ~Gizmos() {}

    // 箱描画
    void DrawBox(
        const DirectX::XMFLOAT3& position,
        const DirectX::XMFLOAT3& angle,
        const DirectX::XMFLOAT3& size,
        const DirectX::XMFLOAT4& color);

private:
    struct Mesh
    {
        Microsoft::WRL::ComPtr<ID3D11Buffer>    vertexBuffer;
        UINT                                    vertexCount;
    };

    struct Instance
    {
        Mesh*                   mesh;
        DirectX::XMFLOAT4X4     worldTransform;
```

```
        DirectX::XMFLOAT4      color;
    };

    // メッシュ生成
    void CreateMesh(ID3D11Device* device, const std::vector<DirectX::XMFLOAT3>& vertices, Mesh& mesh);

    // 箱メッシュ作成
    void CreateBoxMesh(ID3D11Device* device, float width, float height, float depth);

private:
    Mesh                                    boxMesh;
    std::vector<Instance>                   instances;
};
```

Gizmos.cpp

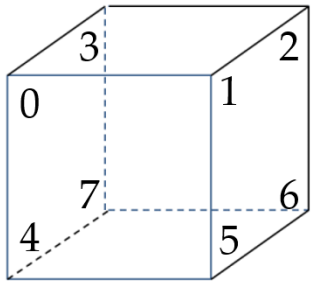
```
#include "Misc.h"
#include "GpuResourceUtils.h"
#include "Gizmos.h"

// コンストラクタ
Gizmos::Gizmos(ID3D11Device* device)
{
    // 箱メッシュ生成
    CreateBoxMesh(device, 0.5f, 0.5f, 0.5f);
}

// 箱描画
void Gizmos::DrawBox(
    const DirectX::XMFLOAT3& position,
    const DirectX::XMFLOAT3& angle,
    const DirectX::XMFLOAT3& size,
    const DirectX::XMFLOAT4& color)
{
    Instance& instance = instances.emplace_back();
    instance.mesh = &boxMesh;
    instance.color = color;

    DirectX::XMMATRIX S = DirectX::XMMatrixScaling(size.x, size.y, size.z);
    DirectX::XMMATRIX R = DirectX::XMMatrixRotationRollPitchYaw(angle.x, angle.y, angle.z);
    DirectX::XMMATRIX T = DirectX::XMMatrixTranslation(position.x, position.y, position.z);
    DirectX::XMStoreFloat4x4(&instance.worldTransform, S * R * T);
}

// メッシュ生成
void Gizmos::CreateMesh(ID3D11Device* device, const std::vector<DirectX::XMFLOAT3>& vertices, Mesh& mesh)
{
    D3D11_BUFFER_DESC desc = {};
    desc.ByteWidth = static_cast<UINT>(sizeof(DirectX::XMFLOAT3) * vertices.size());
    desc.Usage = D3D11_USAGE_IMMUTABLE;
    desc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    desc.CPUAccessFlags = 0;
    desc.MiscFlags = 0;
    desc.StructureByteStride = 0;
    D3D11_SUBRESOURCE_DATA subresourceData = {};
```

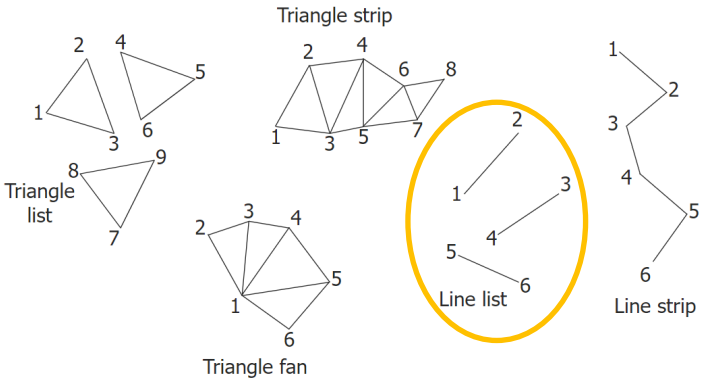
描画するインスタンスデータを追加登録

配置パラメータから
ワールド行列を作成

サブリソースデータに
頂点データを指定することで
頂点バッファ生成時に
頂点データを書き込むことができる

```
    subresourceData.pSysMem = vertices.data();
    subresourceData.SysMemPitch = 0;
    subresourceData.SysMemSlicePitch = 0;

    HRESULT hr = device->CreateBuffer(&desc, &subresourceData, mesh.vertexBuffer.GetAddressOf());
    _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));

    mesh.vertexCount = static_cast<UINT>(vertices.size());
}

// 箱メッシュ作成
void Gizmos::CreateBoxMesh(ID3D11Device* device, float width, float height, float depth)
{
    DirectX::XMFLOAT3 positions[8] =
    {
        // top
        { -width,  height, -depth},
        {  width,  height, -depth},
        {  width,  height,  depth},
        { -width,  height,  depth},
        // bottom
        { -width, -height, -depth},
        {  width, -height, -depth},
        {  width, -height,  depth},
        { -width, -height,  depth},
    };

    std::vector<DirectX::XMFLOAT3> vertices;
    vertices.resize(32);

    // top
    vertices.emplace_back(positions[0]);
    vertices.emplace_back(positions[1]);
    vertices.emplace_back(positions[1]);
    vertices.emplace_back(positions[2]);
    vertices.emplace_back(positions[2]);
    vertices.emplace_back(positions[3]);
    vertices.emplace_back(positions[3]);
    vertices.emplace_back(positions[0]);
    // bottom
    vertices.emplace_back(positions[4]);
    vertices.emplace_back(positions[5]);
    vertices.emplace_back(positions[5]);
    vertices.emplace_back(positions[6]);
    vertices.emplace_back(positions[6]);
    vertices.emplace_back(positions[7]);
    vertices.emplace_back(positions[7]);
    vertices.emplace_back(positions[4]);
    // side
    vertices.emplace_back(positions[0]);
    vertices.emplace_back(positions[4]);
    vertices.emplace_back(positions[1]);
    vertices.emplace_back(positions[5]);
    vertices.emplace_back(positions[2]);
    vertices.emplace_back(positions[6]);
    vertices.emplace_back(positions[3]);
    vertices.emplace_back(positions[7]);
```





Line list による描き方で
頂点バッファを作成する。
Line list は２つの頂点を
並べることで線を描ける。

```
    // メッシュ生成
    CreateMesh(device, vertices, boxMesh);
}
```

作成したメッシュを描画するシェーダーを作成します。
Gizmos.hlsli と GizmosVS.hlsl と GizmosPS.hlsl を作成しましょう。

Gizmos.hlsli

```
struct VS_OUT
{
    float4 position : SV_POSITION;
    float4 color    : COLOR;
};

cbuffer CbMesh : register(b0)
{
    row_major float4x4    worldViewProjection;
    float4                color;
};
```

定数バッファを介して
シェーダーにカメラなどの
データが渡される。

GizmosVS.hlsl

```
#include "Gizmos.hlsli"

VS_OUT main(float4 position : POSITION)
{
    VS_OUT vout;
    vout.position = mul(position, worldViewProjection);
    vout.color = color;

    return vout;
}
```

GizmosPS.hlsl

```
#include "Gizmos.hlsli"

float4 main(VS_OUT pin) : SV_TARGET
{
    return pin.color;
}
```

○定数バッファ

定数バッファとはプログラム側からシェーダーへデータの受け渡しをするバッファです。
ワールド行列、ビュー行列、プロジェクション行列などのパラメータを定数バッファに書き込み、シェーダー内で座標変換の計算ができます。

定数バッファの作成は今後よく使用するのでユーティリティ関数を作成しましょう。

GpuResourceUtils.h

```
---省略---

// GPUリソースユーティリティ
class GpuResourceUtils
{
public:
    ---省略---

    // 定数バッファ作成
    static HRESULT CreateConstantBuffer(
        ID3D11Device* device,
        UINT bufferSize,
        ID3D11Buffer** constantBuffer);
};
```

GpuResourceUtils.cpp

```
---省略---

// 定数バッファ作成
HRESULT GpuResourceUtils::CreateConstantBuffer(
    ID3D11Device* device,
    UINT bufferSize,
    ID3D11Buffer** constantBuffer)
{
    D3D11_BUFFER_DESC desc{};
    desc.Usage = D3D11_USAGE_DEFAULT;
    desc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    desc.CPUAccessFlags = 0;
    desc.MiscFlags = 0;
    desc.ByteWidth = bufferSize;
    desc.StructureByteStride = 0;

    HRESULT hr = device->CreateBuffer(&desc, 0, constantBuffer);
    _ASSERT_EXPR(SUCCEEDED(hr), HRTrace(hr));

    return hr;
}
```

○描画

カメラ、メッシュ、シェーダーの準備ができたので描画実行処理を実装します。
描画に必要な情報をまとめた RenderContext 構造体を定義します。
RenderContext.h を作成しましょう。

RenderContext.h

```
#pragma once
```

```
#include "Camera.h"
#include "RenderState.h"

struct RenderContext
{
    ID3D11DeviceContext*  deviceContext;
    const RenderState*    renderState;
    const Camera*         camera;
};
```

描画に必要な情報を
構造体で定義する

Gizmos クラスに描画処理を実装します。

Gizmos.h

```
---省略---
#include "RenderContext.h"

class Gizmos
{
public:
    ---省略---

    // 描画実行
    void Render(const RenderContext& rc);

private:
    ---省略---

    struct CbMesh
    {
        DirectX::XMFLOAT4X4   worldViewProjection;
        DirectX::XMFLOAT4     color;
    };

    ---省略---

private:
    ---省略---

    Microsoft::WRL::ComPtr<ID3D11VertexShader>  vertexShader;
    Microsoft::WRL::ComPtr<ID3D11PixelShader>   pixelShader;
    Microsoft::WRL::ComPtr<ID3D11InputLayout>   inputLayout;
    Microsoft::WRL::ComPtr<ID3D11Buffer>        constantBuffer;
};
```

```
cbuffer CbMesh : register(b0)
{
    row_major float4x4  worldViewProjection;
    float4              color;
};
```

HLSL で定義している内容と同じにする

Gizmos.cpp

```
---省略---

// コンストラクタ
Gizmos::Gizmos(ID3D11Device* device)
{
```

```
    // 入力レイアウト
    D3D11_INPUT_ELEMENT_DESC inputElementDesc[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT,
                                                        D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };

    // 頂点シェーダー
    GpuResourceUtils::LoadVertexShader(
        device,
        "Data/Shader/GizmosVS.cso",
        inputElementDesc,
        _countof(inputElementDesc),
        inputLayout.GetAddressOf(),
        vertexShader.GetAddressOf());

    // ピクセルシェーダー
    GpuResourceUtils::LoadPixelShader(
        device,
        "Data/Shader/GizmosPS.cso",
        pixelShader.GetAddressOf());

    // 定数バッファ
    GpuResourceUtils::CreateConstantBuffer(
        device,
        sizeof(CbMesh),
        constantBuffer.GetAddressOf());

    ---省略---
}

---省略---

// 描画実行
void Gizmos::Render(const RenderContext& rc)
{
    ID3D11DeviceContext* dc = rc.deviceContext;

    // シェーダー設定
    dc->VSSetShader(vertexShader.Get(), nullptr, 0);
    dc->PSSetShader(pixelShader.Get(), nullptr, 0);
    dc->IASetInputLayout(inputLayout.Get());

    // 定数バッファ設定
    dc->VSSetConstantBuffers(0, 1, constantBuffer.GetAddressOf());

    // レンダーステート設定
    const float blendFactor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    dc->OMSetBlendState(rc.renderState->GetBlendState(BlendState::Opaque), blendFactor, 0xFFFFFFFF);
    dc->OMSetDepthStencilState(rc.renderState->GetDepthStencilState(DepthState::TestAndWrite), 0);
    dc->RSSetState(rc.renderState->GetRasterizerState(RasterizerState::SolidCullNone));

    // ビュープロジェクション行列作成
    DirectX::XMMATRIX V = DirectX::XMLoadFloat4x4(&rc.camera->GetView());
    DirectX::XMMATRIX P = DirectX::XMLoadFloat4x4(&rc.camera->GetProjection());
    DirectX::XMMATRIX VP = V * P;
```

```
cbuffer CbMesh : register(b0)
{
    row_major float4x4  worldViewProjection;
    float4              color;
};
```

b0 のスロットに定数バッファを渡す

```
    // プリミティブ設定
    UINT stride = sizeof(DirectX::XMFLOAT3);
    UINT offset = 0;
    dc->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_LINELIST);

    for (const Instance& instance : instances)
    {
        // 頂点バッファ設定
        dc->IASetVertexBuffers(0, 1, instance.mesh->vertexBuffer.GetAddressOf(), &stride, &offset);

        // ワールドビュープロジェクション行列作成
        DirectX::XMMATRIX W = DirectX::XMLoadFloat4x4(&instance.worldTransform);
        DirectX::XMMATRIX WVP = W * VP;

        // 定数バッファ更新
        CbMesh cbMesh;
        DirectX::XMStoreFloat4x4(&cbMesh.worldViewProjection, WVP);
        cbMesh.color = instance.color;

        dc->UpdateSubresource(constantBuffer.Get(), 0, 0, &cbMesh, 0, 0);

        // 描画
        dc->Draw(instance.mesh->vertexCount, 0);
    }
    instances.clear();
}
```

LineList で描画

Graphics.h

```
---省略---
#include "Gizmos.h"

// グラフィックス
class Graphics
{
public:
    ---省略---

    // ギズモ取得
    Gizmos* GetGizmos() { return gizmos.get(); }

private:
    ---省略---
    std::unique_ptr<Gizmos>             gizmos;
};
```

Graphics.cpp

```
---省略---

// 初期化
void Graphics::Initialize(HWND hWnd)
{
    ---省略---
```

```
    // ギズモ生成
    gizmos = std::make_unique<Gizmos>(device.Get());
}
```

Scene.h

```
#include "Camera.h"
---省略---

// ギズモテストシーン
class GizmosTestScene : public Scene
{
public:
    GizmosTestScene();
    ~GizmosTestScene() override = default;

    // 描画処理
    void Render(float elapsedTime) override;

private:
    Camera   camera;
    float    rotation = 0;
};
```

Scene.cpp

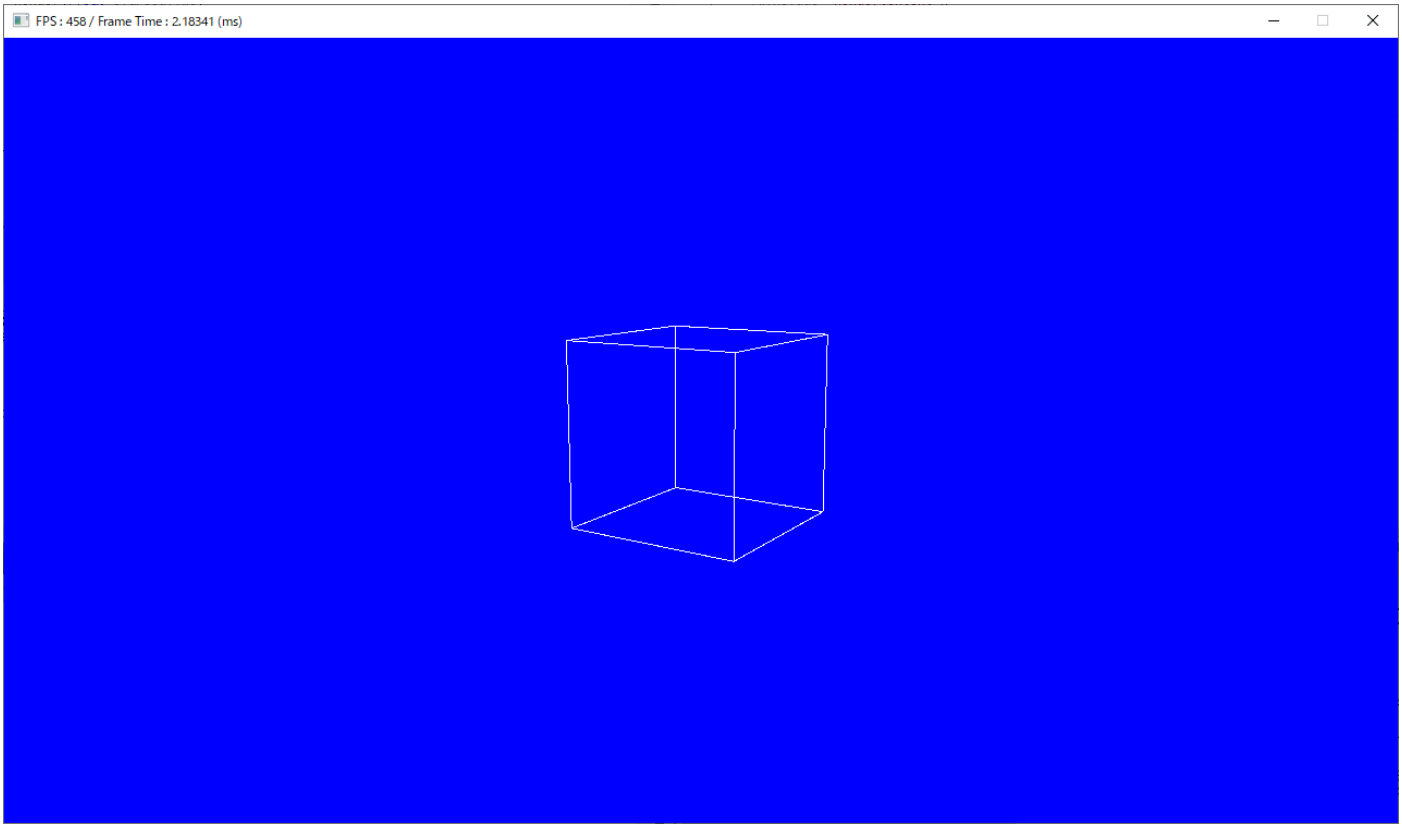
```
---省略---

// コンストラクタ
GizmosTestScene::GizmosTestScene()
{
    float screenWidth = Graphics::Instance().GetScreenWidth();
    float screenHeight = Graphics::Instance().GetScreenHeight();

    // カメラ設定
    camera.SetPerspectiveFov(
        DirectX::XMConvertToRadians(45),    // 画角
        screenWidth / screenHeight,         // 画面アスペクト比
        0.1f,                               // ニアクリップ
        1000.0f                             // ファークリップ
    );
    camera.SetLookAt(
        { 0, 1, -5 },// 視点
        { 0, 0, 0 }, // 注視点
        { 0, 1, 0 }  // 上ベクトル
    );
}

// 描画処理
void GizmosTestScene::Render(float elapsedTime)
{
    Gizmos* gizmos = Graphics::Instance().GetGizmos();
```

```
    // 回転処理
    rotation += 0.5f * elapsedTime;

    // 箱描画
    gizmos->DrawBox(
        { 0, 0, 0 },          // 位置
        { 0, rotation, 0 },   // 回転
        { 1, 1, 1 },          // サイズ
        { 1, 1, 1, 1 });      // 色

    // 描画コンテキスト設定
    RenderContext rc;
    rc.camera = &camera;
    rc.deviceContext = Graphics::Instance().GetDeviceContext();
    rc.renderState = Graphics::Instance().GetRenderState();

    // 描画実行
    gizmos->Render(rc);
}
```

Framework.cpp

```
---省略---

// コンストラクタ
Framework::Framework(HWND hWnd)
{
    ---省略---

    // シーン初期化
    scene = std::make_unique<RasterizeTestScene>();
    scene = std::make_unique<GizmosTestScene>();
}
```

実行確認してみましょう。
下図のようにラインによる立方体が表示されていれば OK です。



○球形状の追加

球形状のメッシュも描画できるようにしましょう。

Gizmos.h

```
---省略---

class Gizmos
{
public:
    ---省略---

    // 球描画
    void DrawSphere(
        const DirectX::XMFLOAT3& position,
        float radius,
        const DirectX::XMFLOAT4& color);

private:
    ---省略---

    // 球メッシュ作成
    void CreateSphereMesh(ID3D11Device* device, float radius, int subdivisions);

private:
    ---省略---
    Mesh                                    sphereMesh;
};
```

Gizmos.cpp

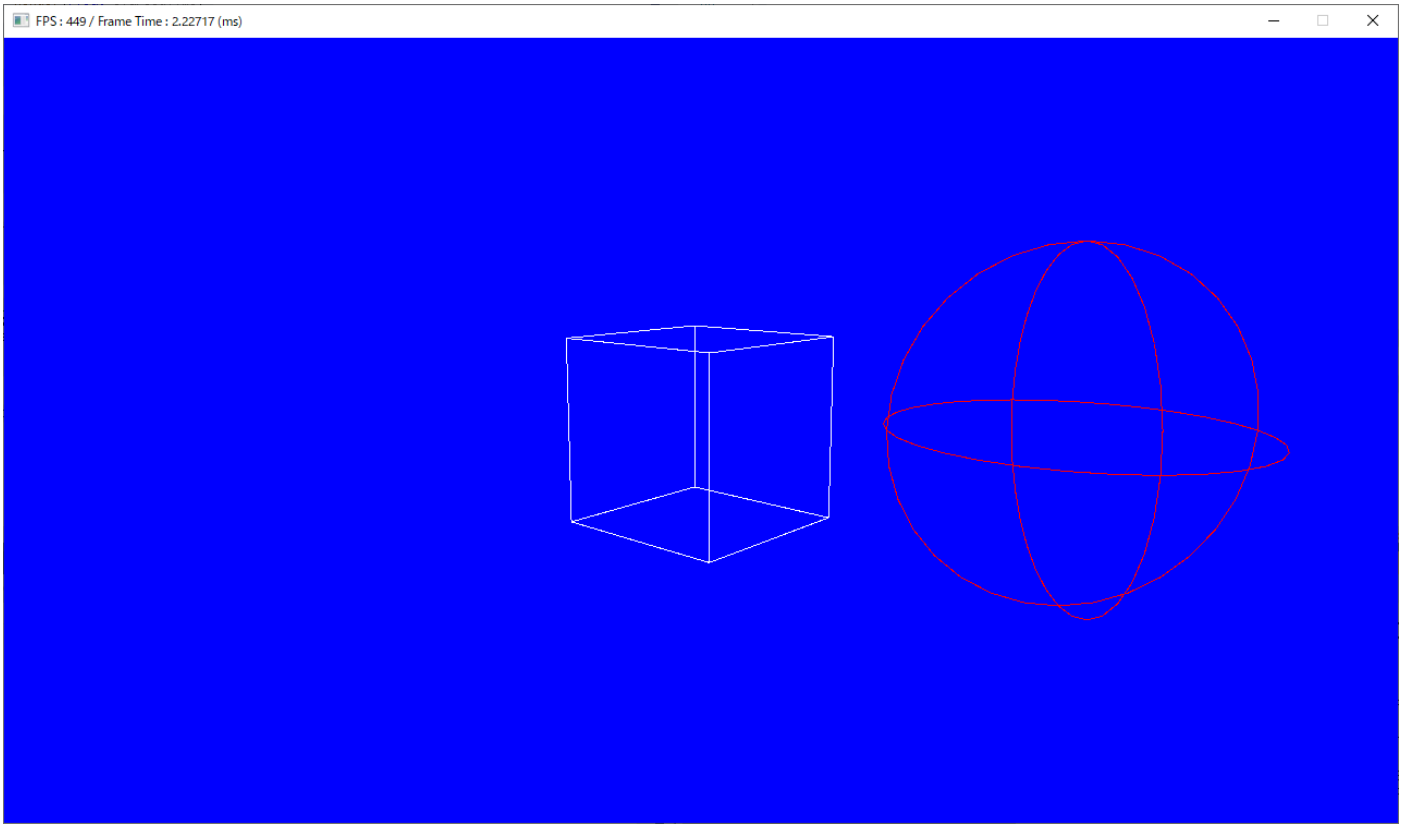
```
---省略---

// コンストラクタ
Gizmos::Gizmos(ID3D11Device* device)
{
    ---省略---

    // 球メッシュ生成
    CreateSphereMesh(device, 1.0f, 32);
}

---省略---

// 球描画
void Gizmos::DrawSphere(
    const DirectX::XMFLOAT3& position,
    float radius,
    const DirectX::XMFLOAT4& color)
{
    Instance& instance = instances.emplace_back();
    instance.mesh = &sphereMesh;
    instance.color = color;

    DirectX::XMMATRIX S = DirectX::XMMatrixScaling(radius, radius, radius);
    DirectX::XMMATRIX T = DirectX::XMMatrixTranslation(position.x, position.y, position.z);
    DirectX::XMStoreFloat4x4(&instance.worldTransform, S * T);
}

---省略---

// 球メッシュ作成
void Gizmos::CreateSphereMesh(ID3D11Device* device, float radius, int subdivisions)
{
    float step = DirectX::XM_2PI / subdivisions;

    std::vector<DirectX::XMFLOAT3> vertices;

    // XZ平面
    for (int i = 0; i < subdivisions; ++i)
    {
        for (int j = 0; j < 2; ++j)
        {
            float theta = step * ((i + j) % subdivisions);

            DirectX::XMFLOAT3& p = vertices.emplace_back();
            p.x = sinf(theta) * radius;
            p.y = 0.0f;
            p.z = cosf(theta) * radius;
        }
    }
    // XY平面
    for (int i = 0; i < subdivisions; ++i)
```

```
    {
        for (int j = 0; j < 2; ++j)
        {
            float theta = step * ((i + j) % subdivisions);

            DirectX::XMFLOAT3& p = vertices.emplace_back();
            p.x = sinf(theta) * radius;
            p.y = cosf(theta) * radius;
            p.z = 0.0f;
        }
    }
    // YZ平面
    for (int i = 0; i < subdivisions; ++i)
    {
        for (int j = 0; j < 2; ++j)
        {
            float theta = step * ((i + j) % subdivisions);

            DirectX::XMFLOAT3& p = vertices.emplace_back();
            p.x = 0.0f;
            p.y = sinf(theta) * radius;
            p.z = cosf(theta) * radius;
        }
    }

    // メッシュ生成
    CreateMesh(device, vertices, sphereMesh);
}
```

Scene.cpp

```
---省略---

// 描画処理
void GizmosTestScene::Render(float elapsedTime)
{
    ---省略---

    // 箱描画
    ---省略---

    // 球描画
    gizmos->DrawSphere(
        { 2, 0, 0 },          // 位置
        1.0f,                 // 半径
        { 1, 0, 0, 1 });      // 色

    ---省略---
}
```

実行確認してみましょう。
画面右側に球体が表示されていれば OK です。


FPS : 449 / Frame Time : 2.22717 (ms)