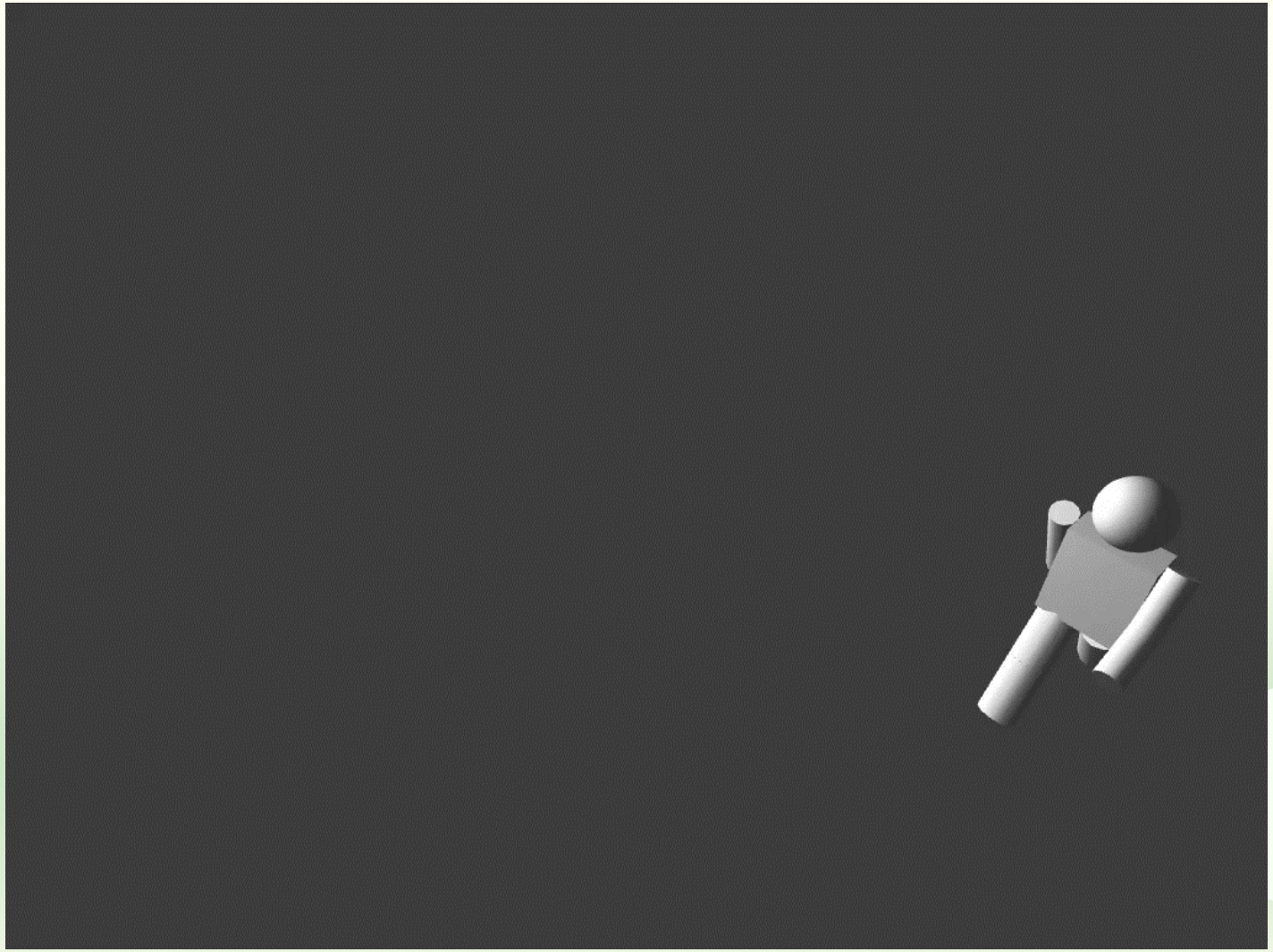
メディアプログラミング演習

第3回

本日はロボットの歩行アニメーションの作成

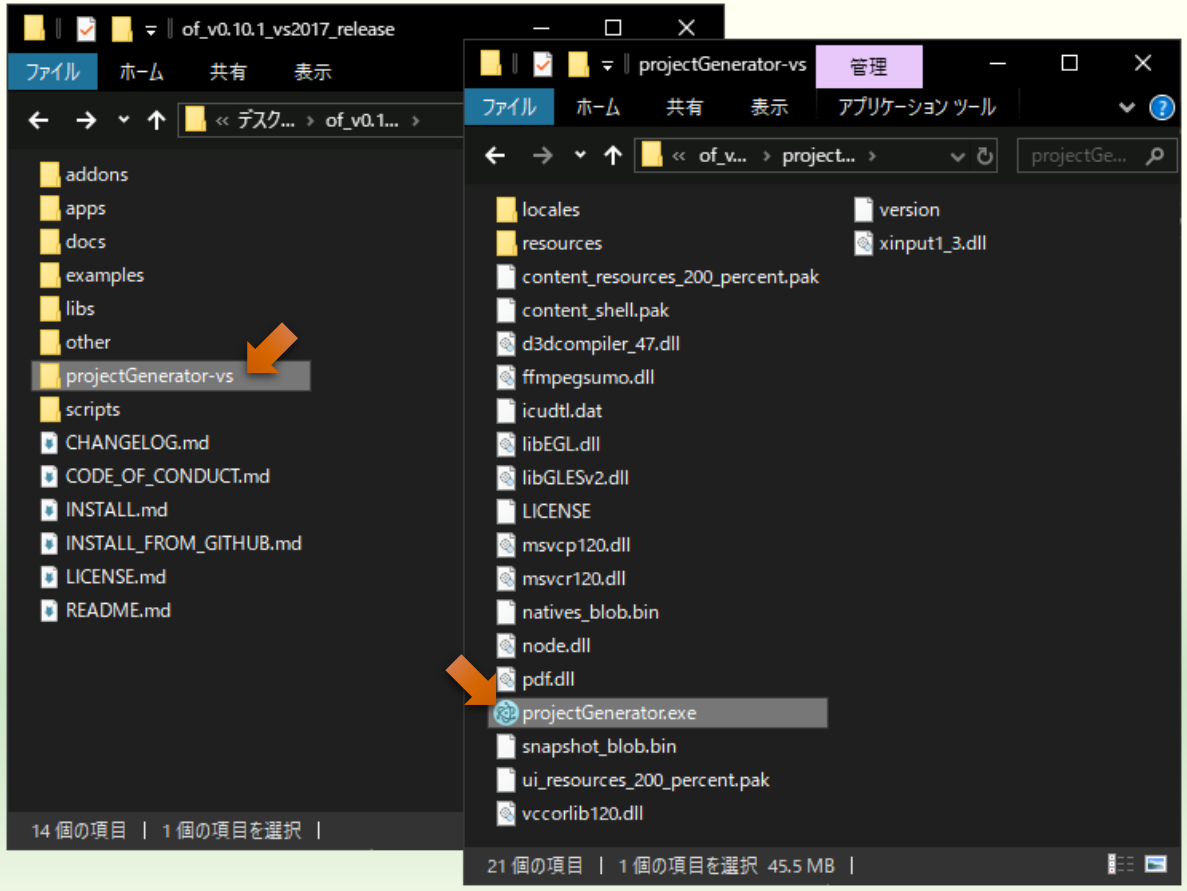


準備

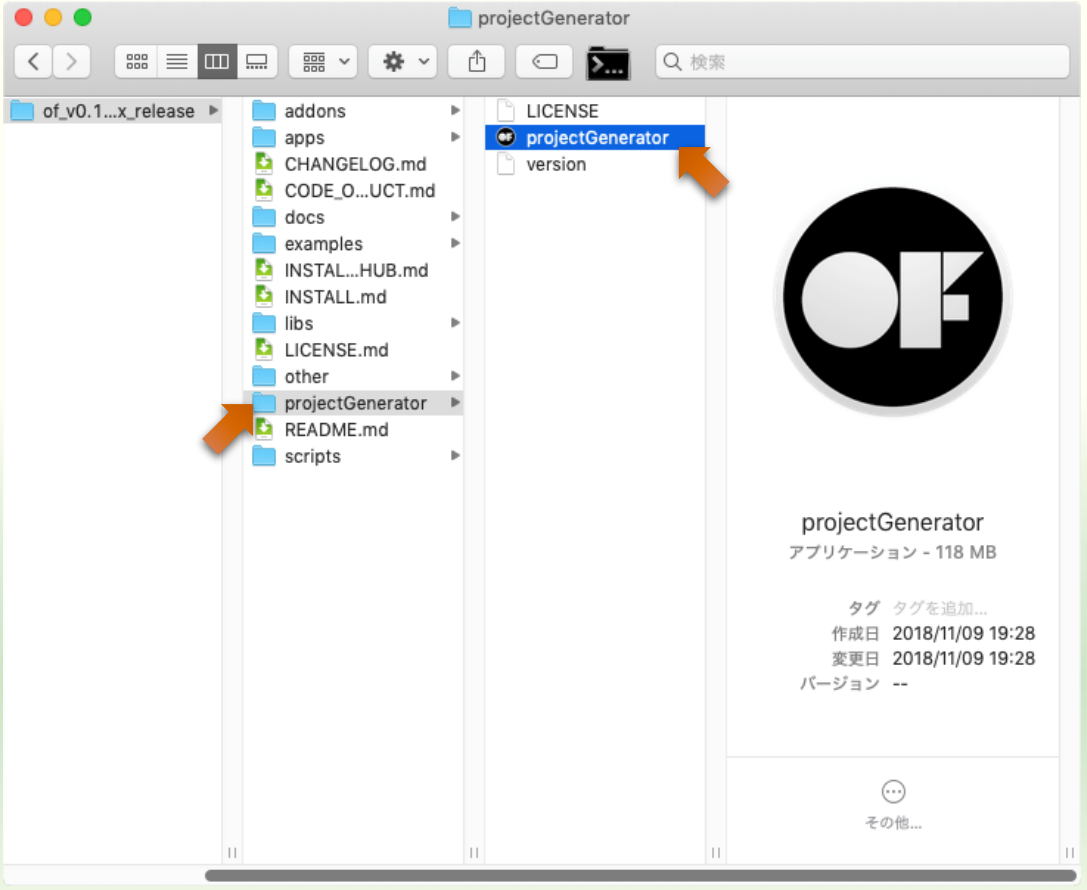
プロジェクトの作成

projectGenerator を起動する

windows 版のパッケージ



macOS 版のパッケージ



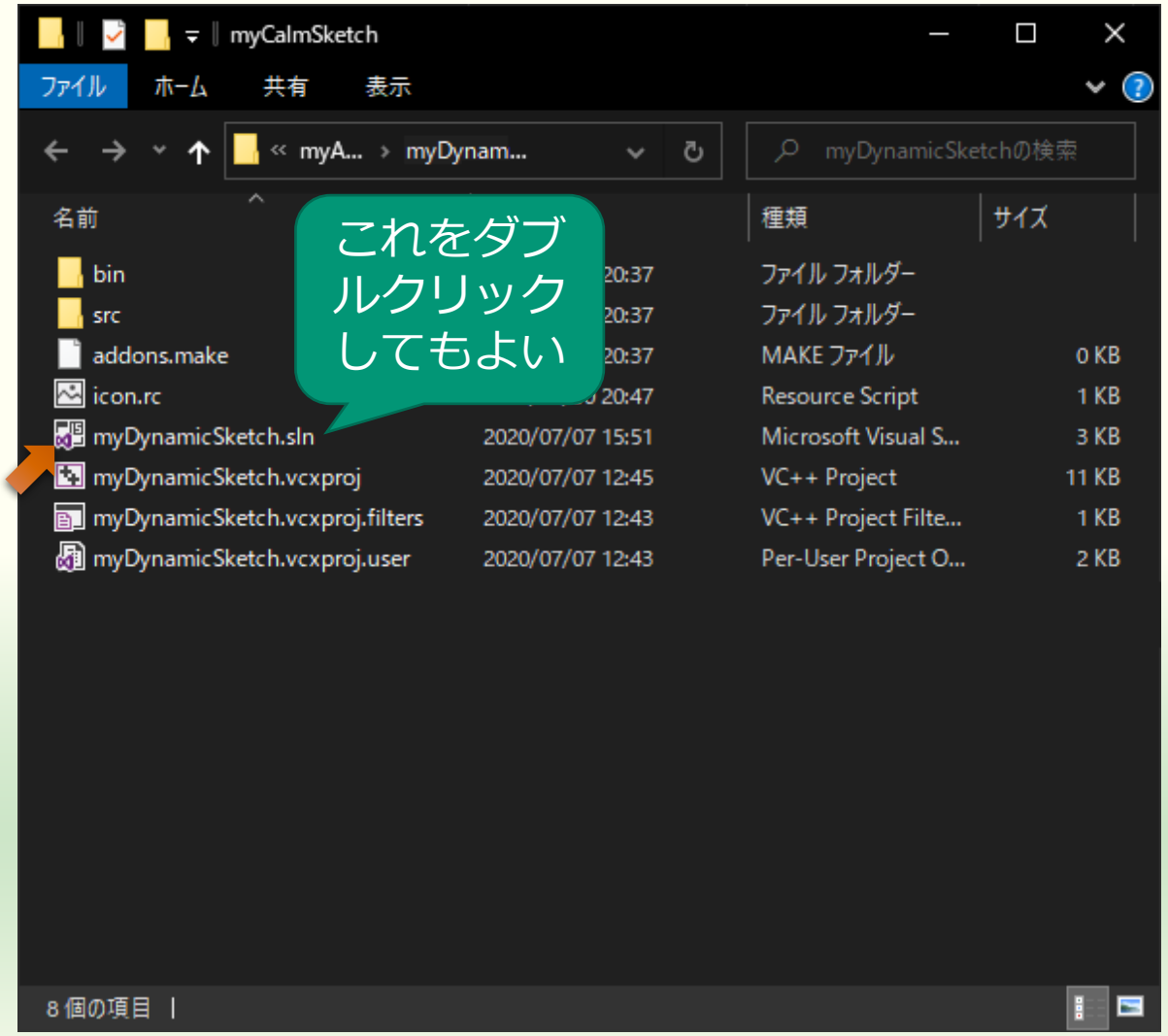
空のプロジェクトの作成

- Project name:
  - 作成するプロジェクト（プログラム）の名前
- Project path:
  - 作成するプロジェクトのファイルを置く場所
  - openFrameworks のパッケージを展開した場所の中の apps¥myApps

プロジェクトの作成成功

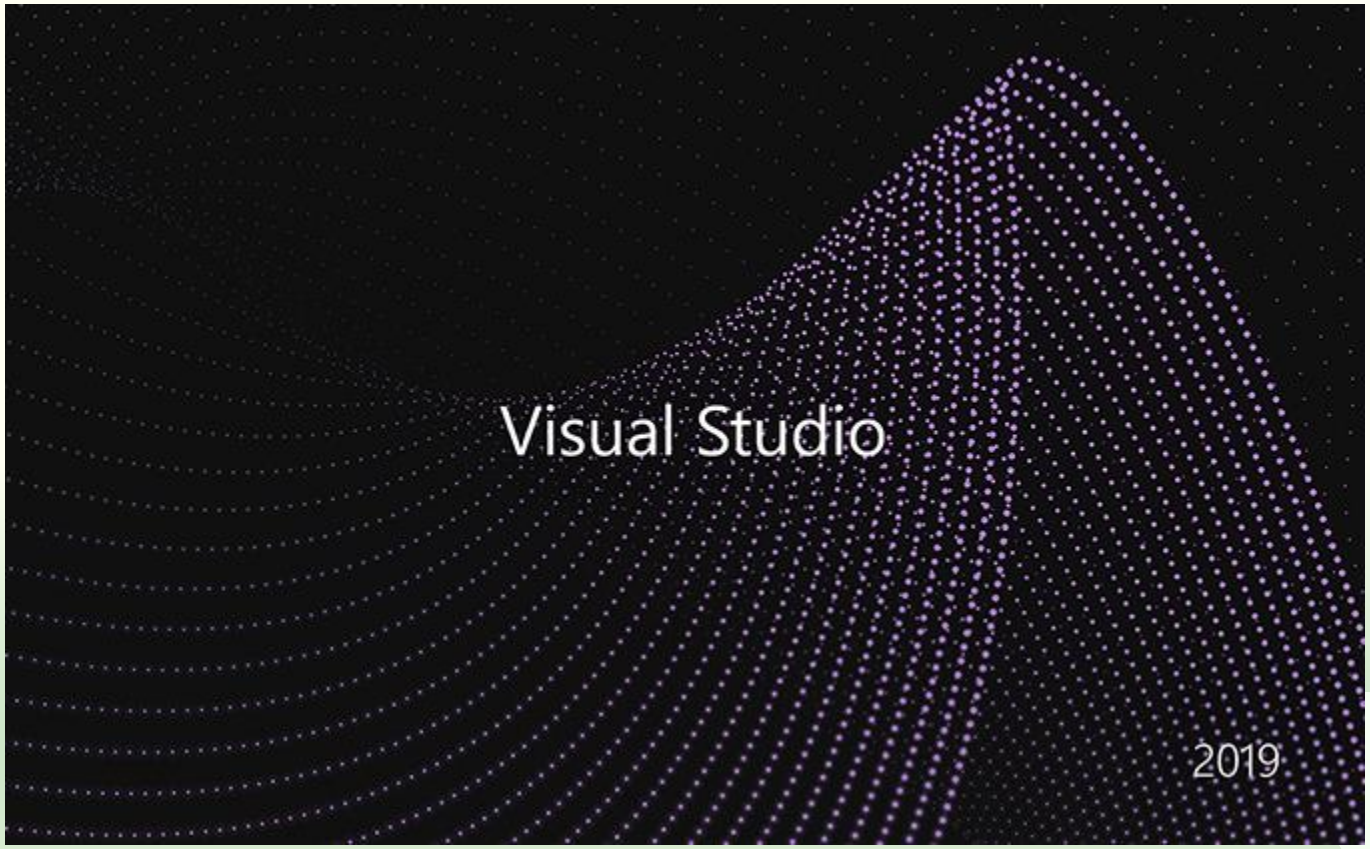
クリックすると開く

IDE で開く

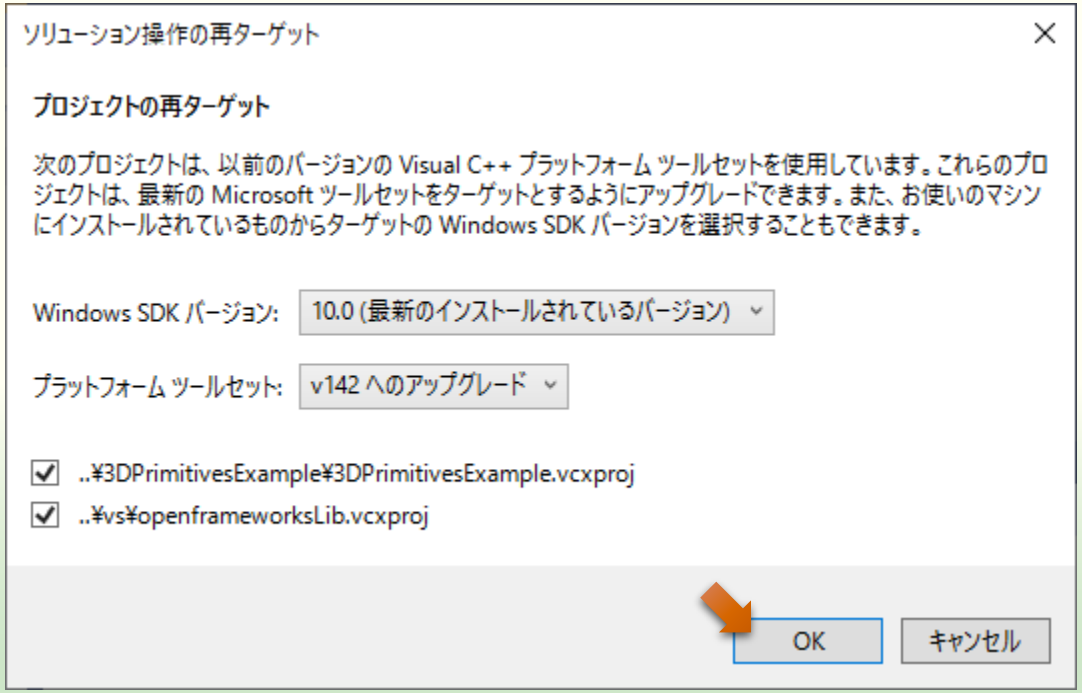


これをダブルクリックしてもよい

Visual Studio 2019 が起動する

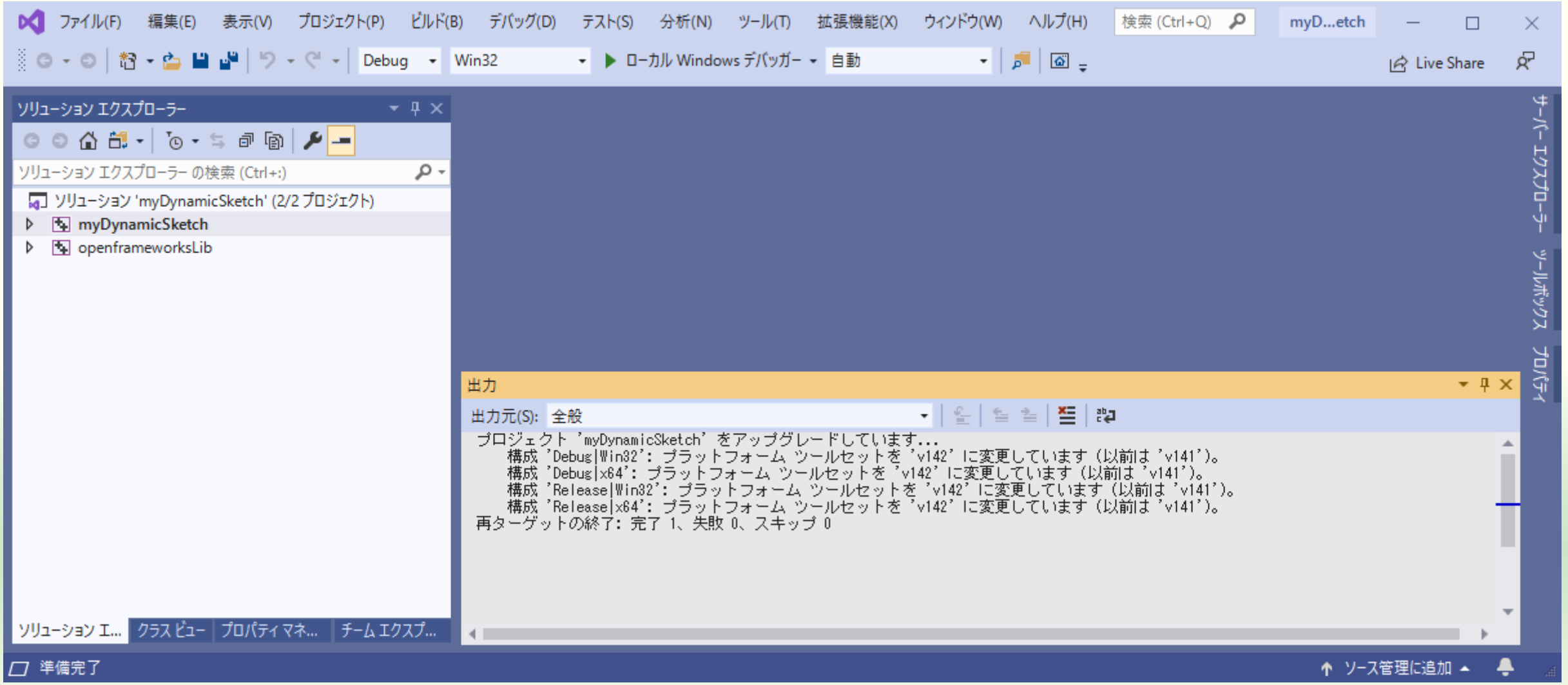


ソリューションの再ターゲット



Visual Studio は頻繁に更新しているので皆さんがお使いの Visual Studio SDK のバージョンと合わない場合がある

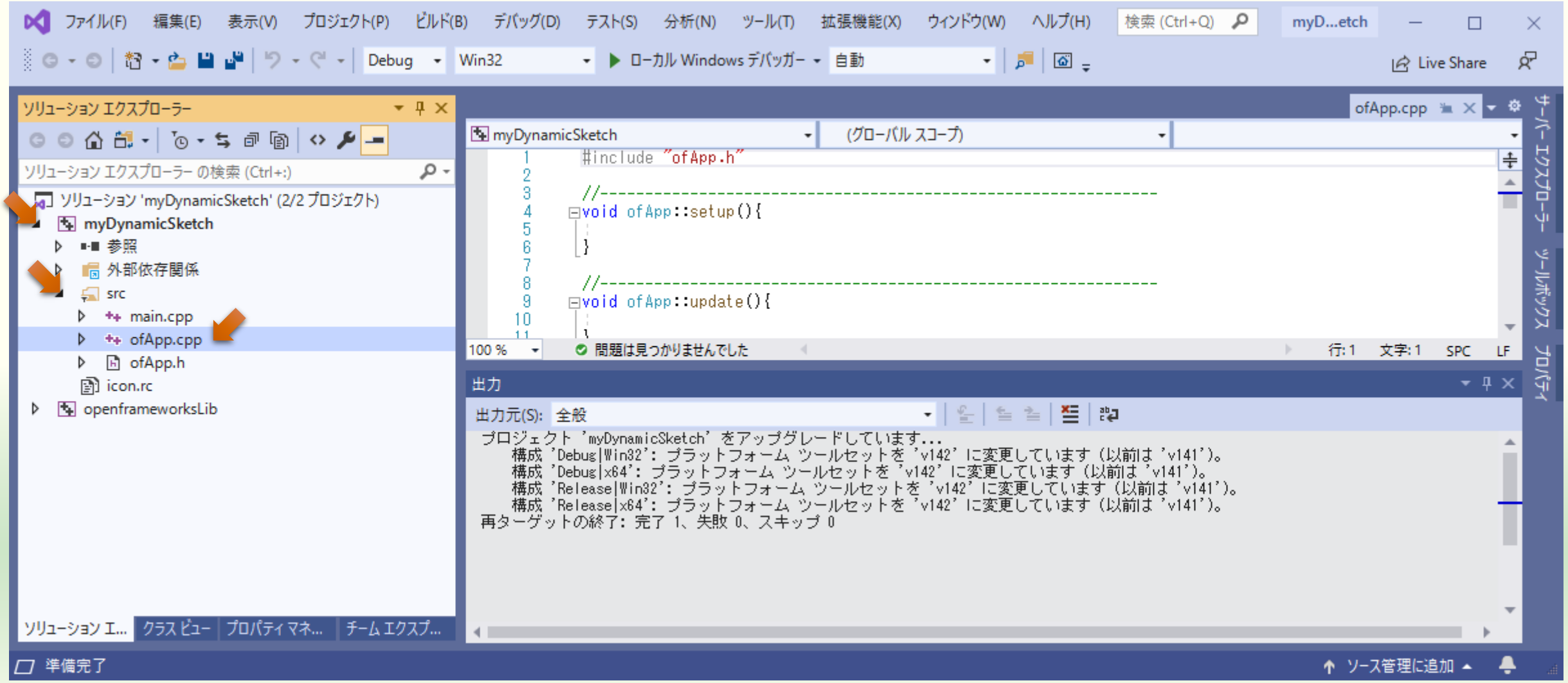
Visual Studio 起動



３次元コンピュータグラフィックス

カメラと照明を配置する

ofApp.cpp を開く



ofDrawSphere() で球を描く

```
//--------------------------------------------
void ofApp::setup(){

}

//--------------------------------------------
void ofApp::update(){

}

//--------------------------------------------
void ofApp::draw(){
  ofDrawSphere(0.0f, 0.0f, 30.0f);
}

//--------------------------------------------
void ofApp::keyPressed(int key){

}
```

中心位置　半径

- (0, 0) を中心に半径 30 の**球**を ofDrawSphere() 関数で描く

ofDrawSphere() の宣言

- **void ofDrawSphere(float x, float y, float radius)**

ofDrawSphere() 関数には戻り値がない

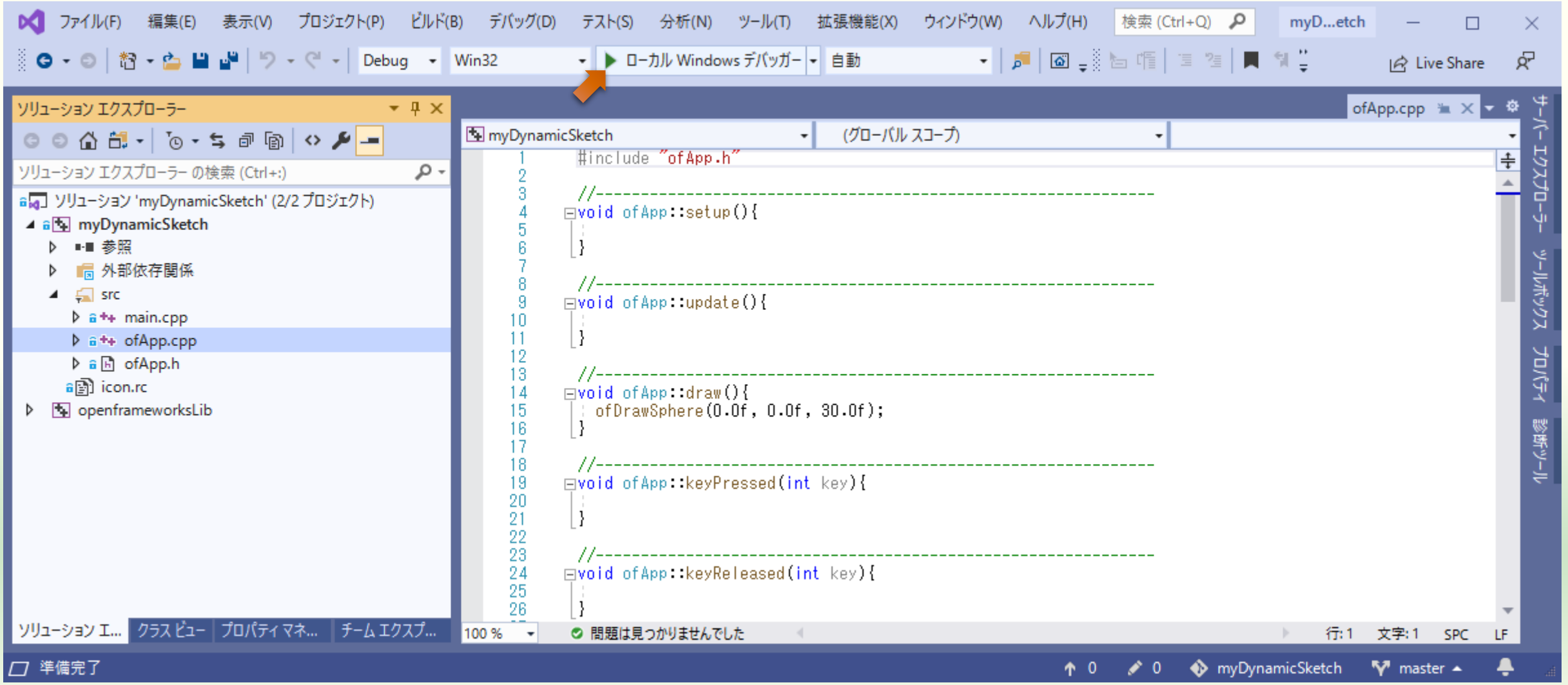
第 1 引数 x は float 型である

第 2 引数 y は float 型である

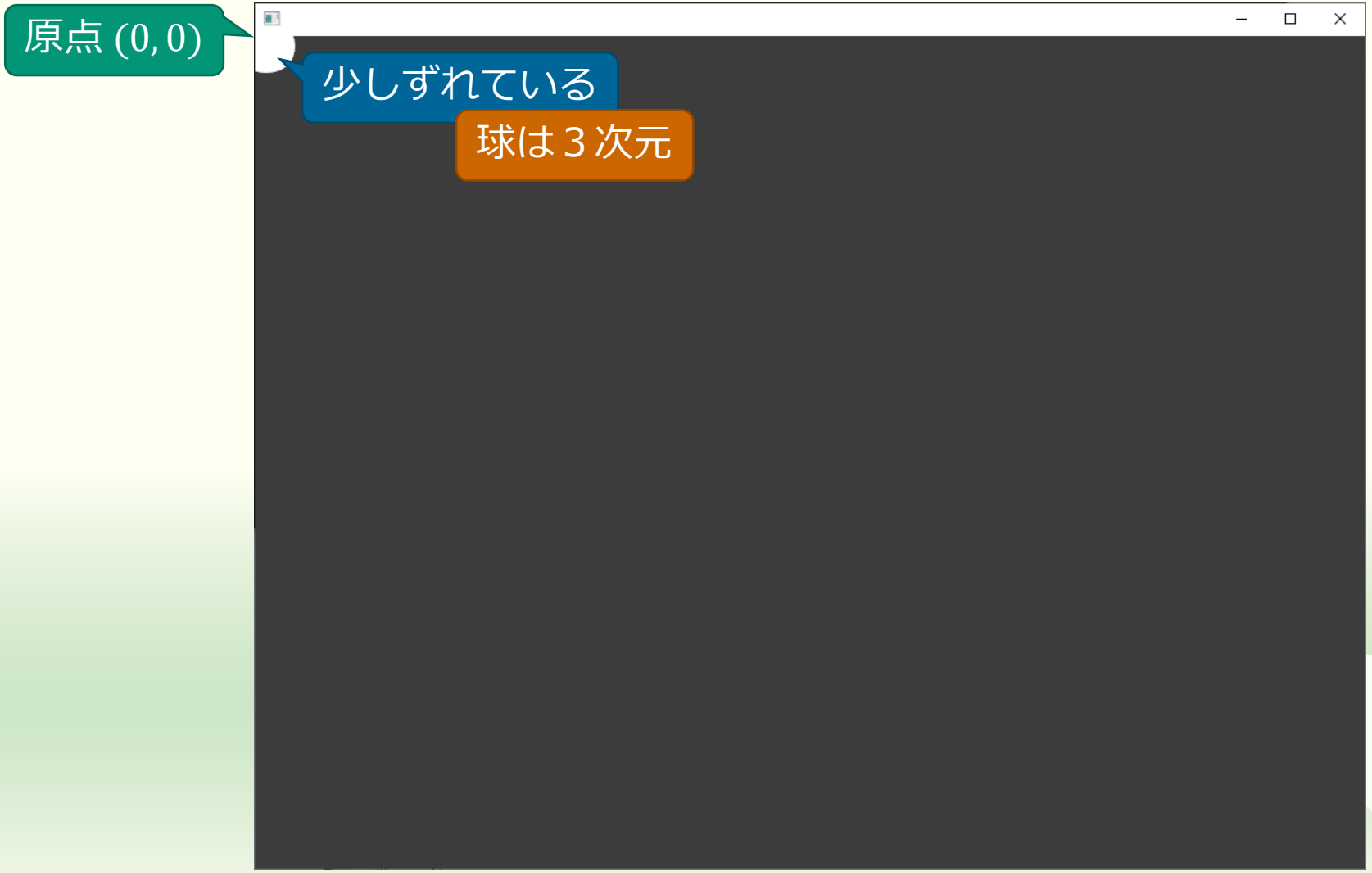
第 3 引数 radius は float 型である

- 仮引数 x, y, radius が何を表すのかは示されていない
  - マニュアルやコメントで説明されている（はず）
  - 宣言では引数名 x, y, radius が省略されている場合がある

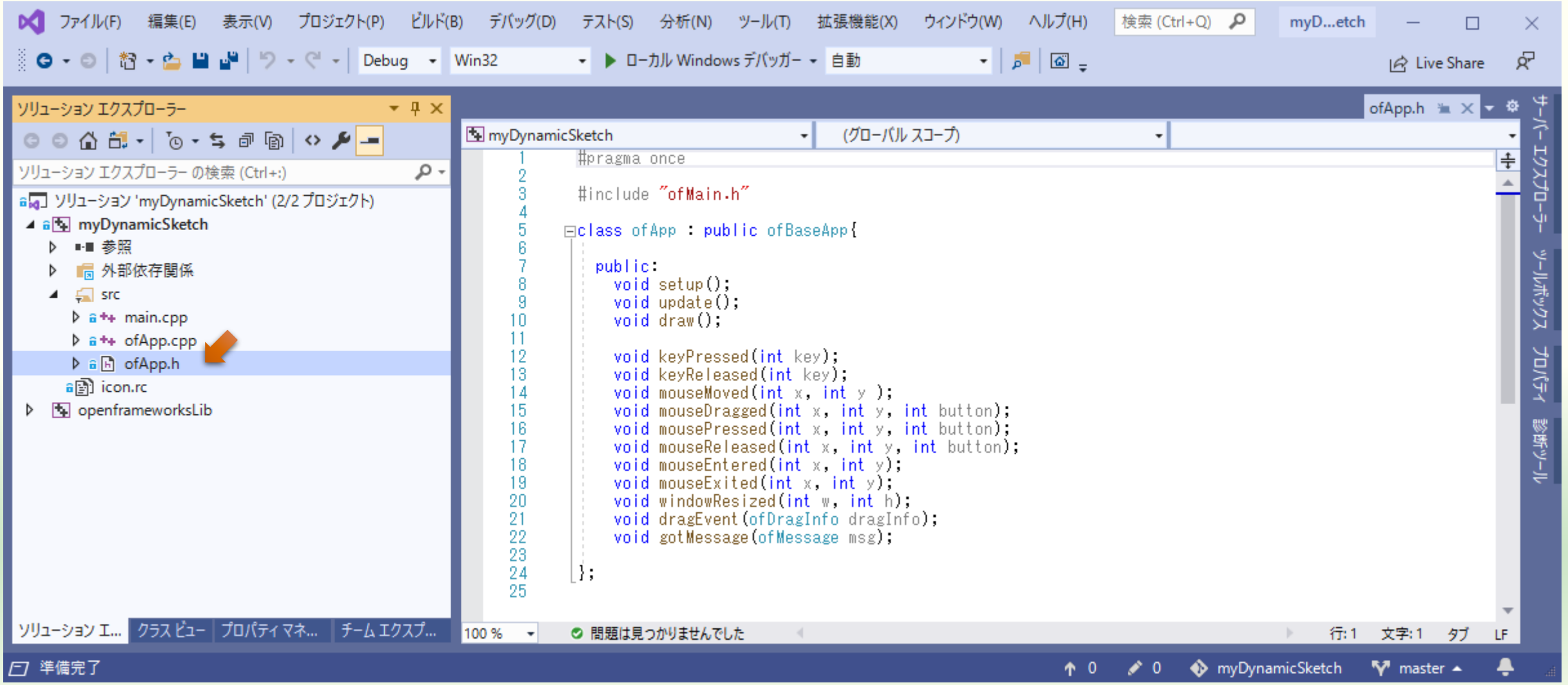
ビルドと実行


ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) テスト(S) 分析(N) ツール(T) 拡張機能(X) ウィンドウ(W) ヘルプ(H)
検索 (Ctrl+Q)
Debug
Win32
ローカル Windows デバッガー
自動
Live Share
ソリューション エクスプローラー
ソリューション エクスプローラー の検索 (Ctrl+;)
ソリューション 'myDynamicSketch' (2/2 プロジェクト)
myDynamicSketch
参照
外部依存関係
src
main.cpp
ofApp.cpp
ofApp.h
icon.rc
openframeworksLib
ofApp.cpp
myDynamicSketch
(グローバル スコープ)
#include "ofApp.h"
//--
void ofApp::setup(){
}
//--
void ofApp::update(){
}
//--
void ofApp::draw(){
ofDrawSphere(0.0f, 0.0f, 30.0f);
}
//--
void ofApp::keyPressed(int key){
}
//--
void ofApp::keyReleased(int key){
}
100 %
問題は見つかりませんでした
行: 1 文字: 1 SPC LF
ソリューション エ... クラス ビュー プロパティ マネ... チーム エクスプ...
サーバー エクスプローラー ツールボックス プロパティ 診断ツール
準備完了
0 0 myDynamicSketch master

左上の原点を中心に球が描かれる



そこで ofApp.h を開く



ofApp クラスにカメラのメンバ変数を追加する

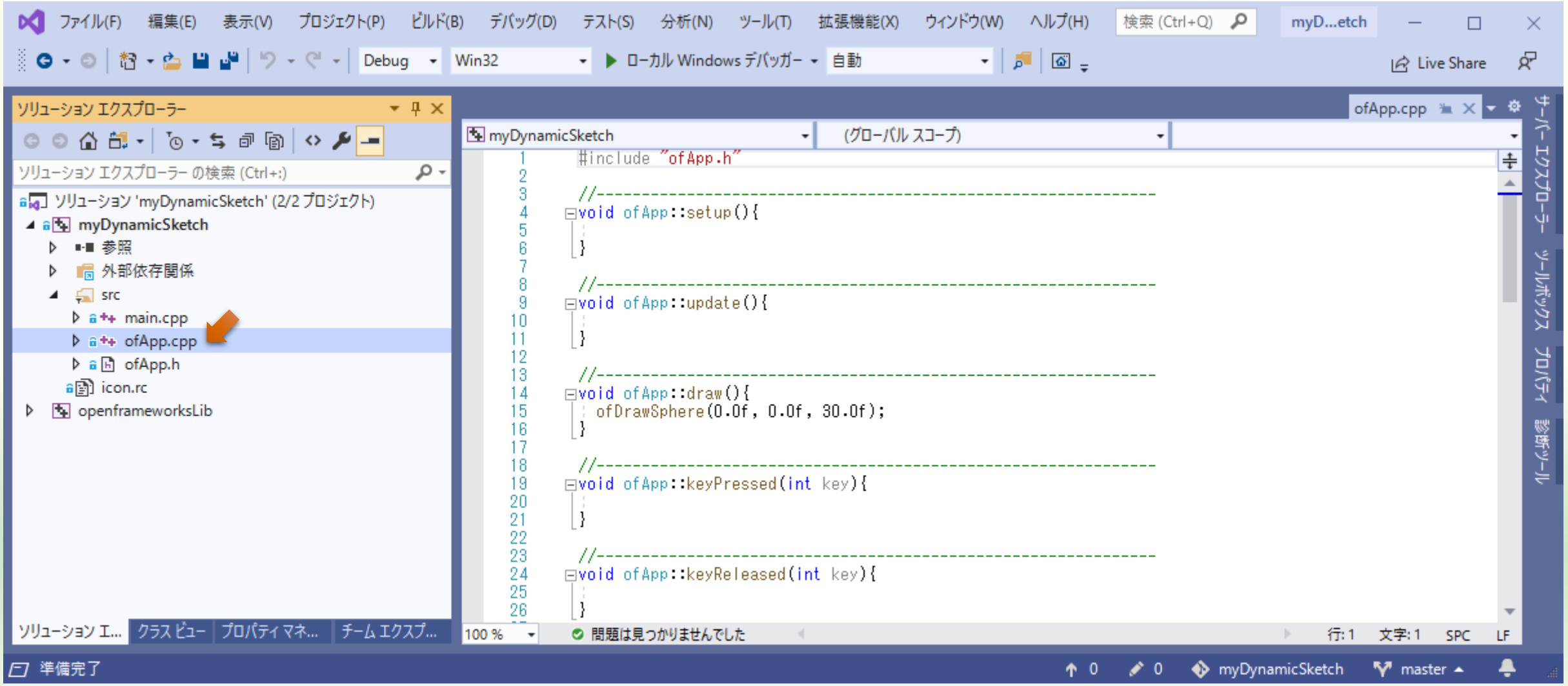
```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofCamera camera;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
  （以下略）
```

- **ofCamera** は 3 次元空間中のカメラのクラス
  - 変数 camera は ofCamera クラスの**インスタンス**

オブジェクトという用語について

- ３次元ＣＧではシーンに配置する**物体**
  - ３次元空間ではカメラもシーンに配置する物体の一つ
  - **シーン**はＣＧの作成対象となる３次元空間
- C++ 言語ではクラスによって定義された構造を持つメモリ上の**データ**
  - camera は位置や方向の情報を保持するメモリが割り当てられた**オブジェクト**
  - オブジェクトはクラスから生成したという文脈では**インスタンス**と呼ばれる

再び ofApp.cpp を開く

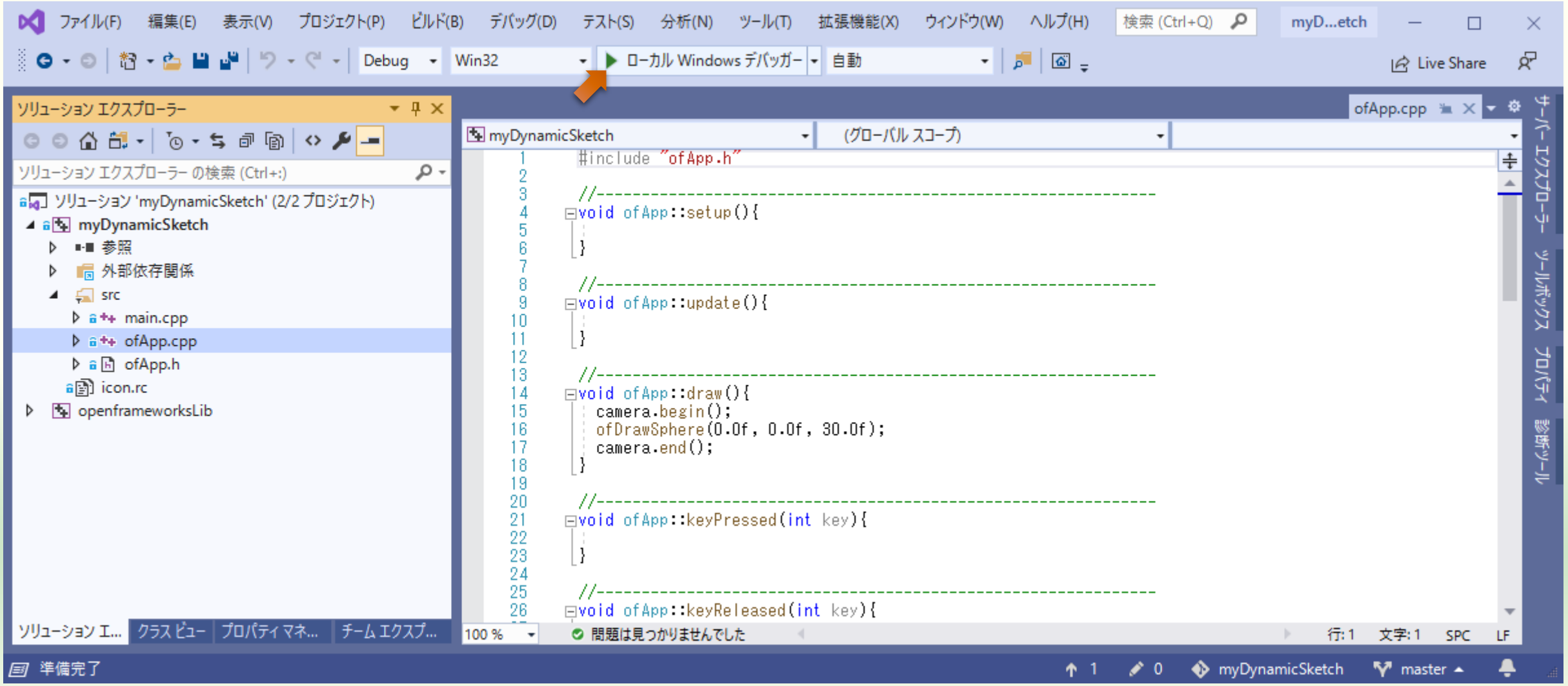


カメラを有効にする

```
//--------------------------------------------
void ofApp::setup(){

}

//--------------------------------------------
void ofApp::update(){

}

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  ofDrawSphere(0.0f, 0.0f, 30.0f);
  camera.end();
}

//--------------------------------------------
void ofApp::keyPressed(int key){

}
```

- camera.begin() と camera.end() の間は３次元空間への描画になる
  - begin() , end() は ofCamera クラスの**メンバ関数**
    - camera というオブジェクトを操作する「方法」なので**メソッド**ともいう
    - “.”（ピリオド）はメンバ関数を呼び出す**ドット演算子**

ビルドと実行


ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) テスト(S) 分析(N) ツール(T) 拡張機能(X) ウィンドウ(W) ヘルプ(H)
検索 (Ctrl+Q)
Debug
Win32
ローカル Windows デバッガー
自動
Live Share
ソリューション エクスプローラー
ソリューション エクスプローラー の検索 (Ctrl+;)
ソリューション 'myDynamicSketch' (2/2 プロジェクト)
myDynamicSketch
参照
外部依存関係
src
main.cpp
ofApp.cpp
ofApp.h
icon.rc
openframeworksLib
ofApp.cpp
myDynamicSketch
(グローバル スコープ)
#include "ofApp.h"
//--
void ofApp::setup(){
}
//--
void ofApp::update(){
}
//--
void ofApp::draw(){
camera.begin();
ofDrawSphere(0.0f, 0.0f, 30.0f);
camera.end();
}
//--
void ofApp::keyPressed(int key){
}
//--
void ofApp::keyReleased(int key){
100 %
問題は見つかりませんでした
行: 1 文字: 1 SPC LF
準備完了
1 0 myDynamicSketch master

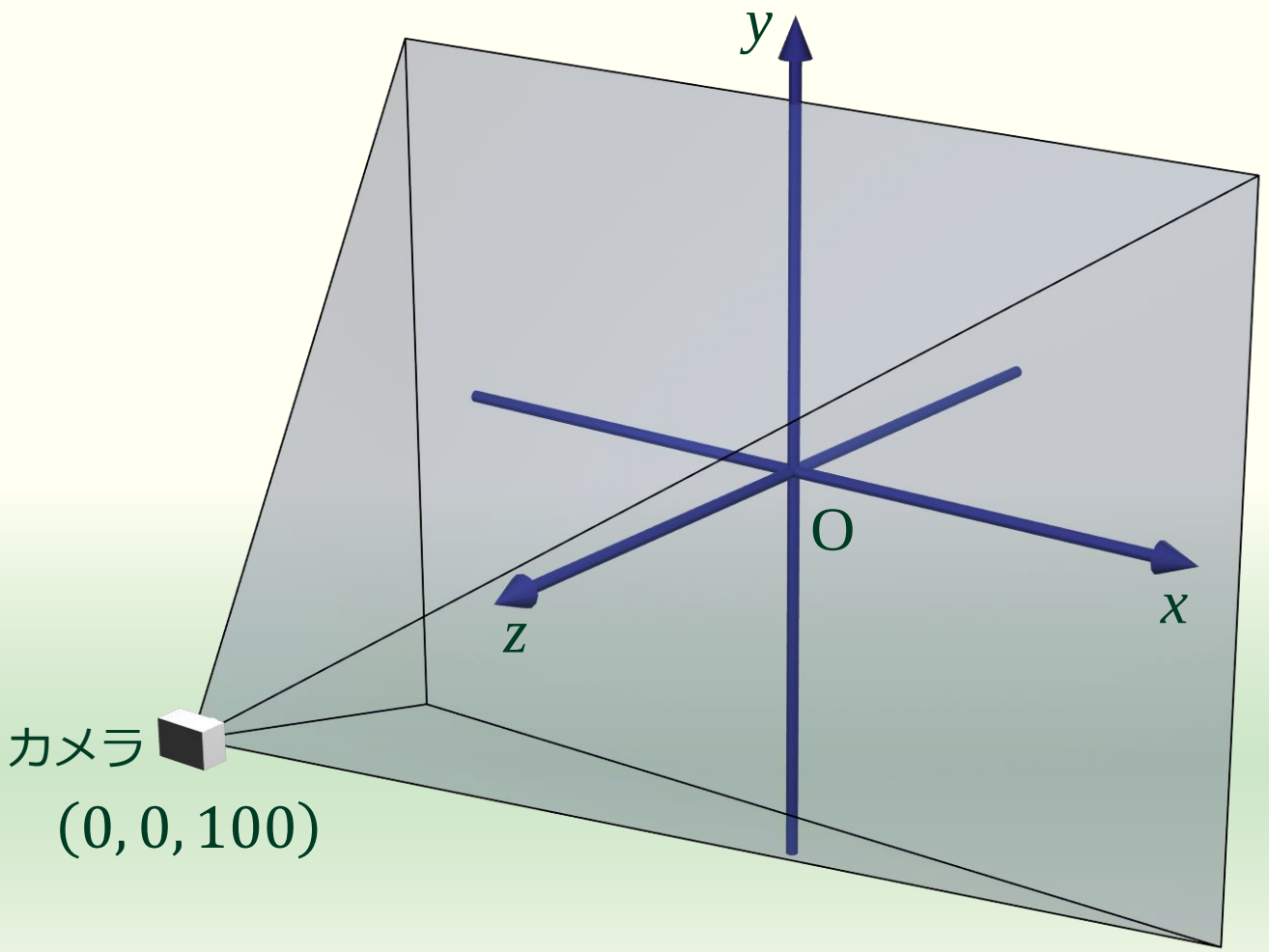
真っ白

カメラを後ろに下げる

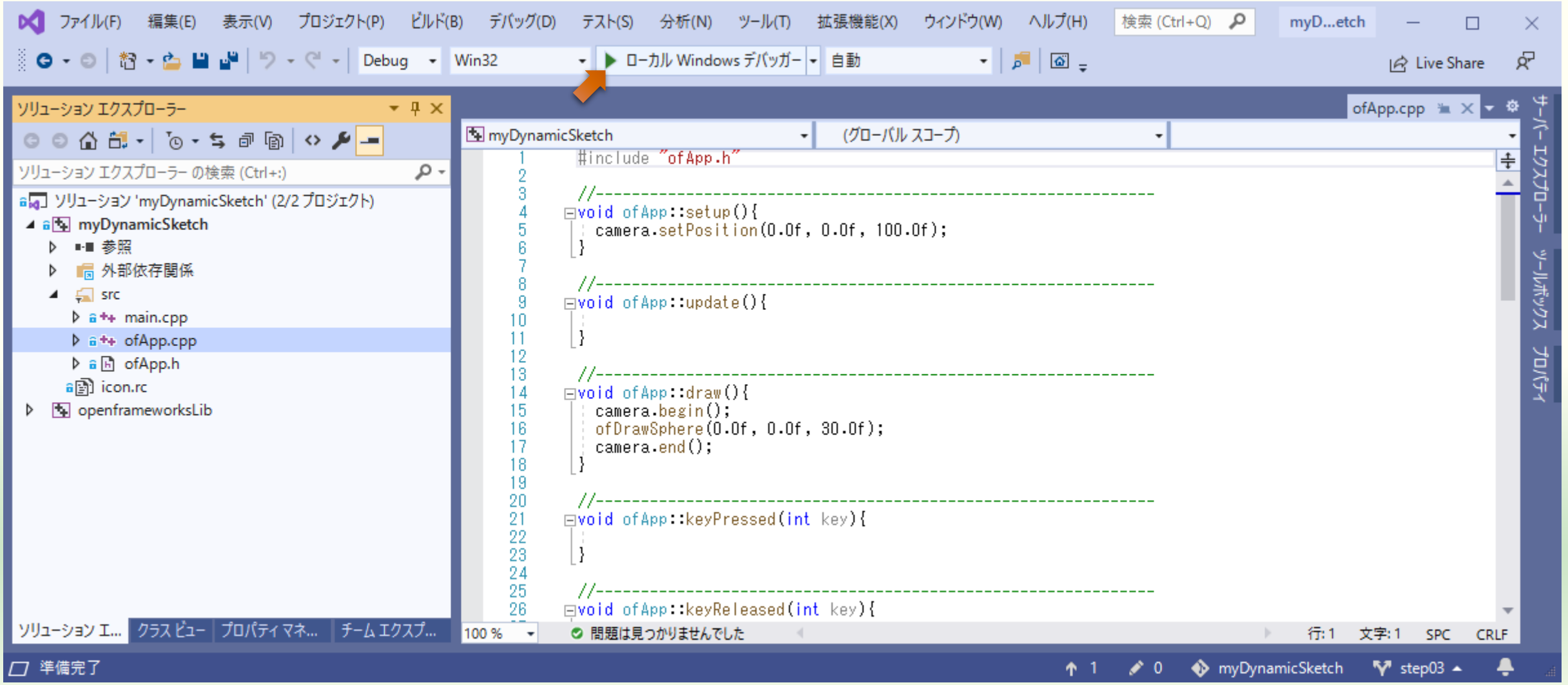
```
//--------------------------------------------
void ofApp::setup(){
  camera.setPosition(0.0f, 0.0f, 100.0f);
}

//--------------------------------------------
void ofApp::update(){

}

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  ofDrawSphere(0.0f, 0.0f, 30.0f);
  camera.end();
}
```

- ofCamera クラスのオブジェクトは setPosition() という**メソッド**でカメラの位置を設定できる
- カメラを動かさないなら setup() で設定する
  - 動かしたければ update() か draw() で設定する

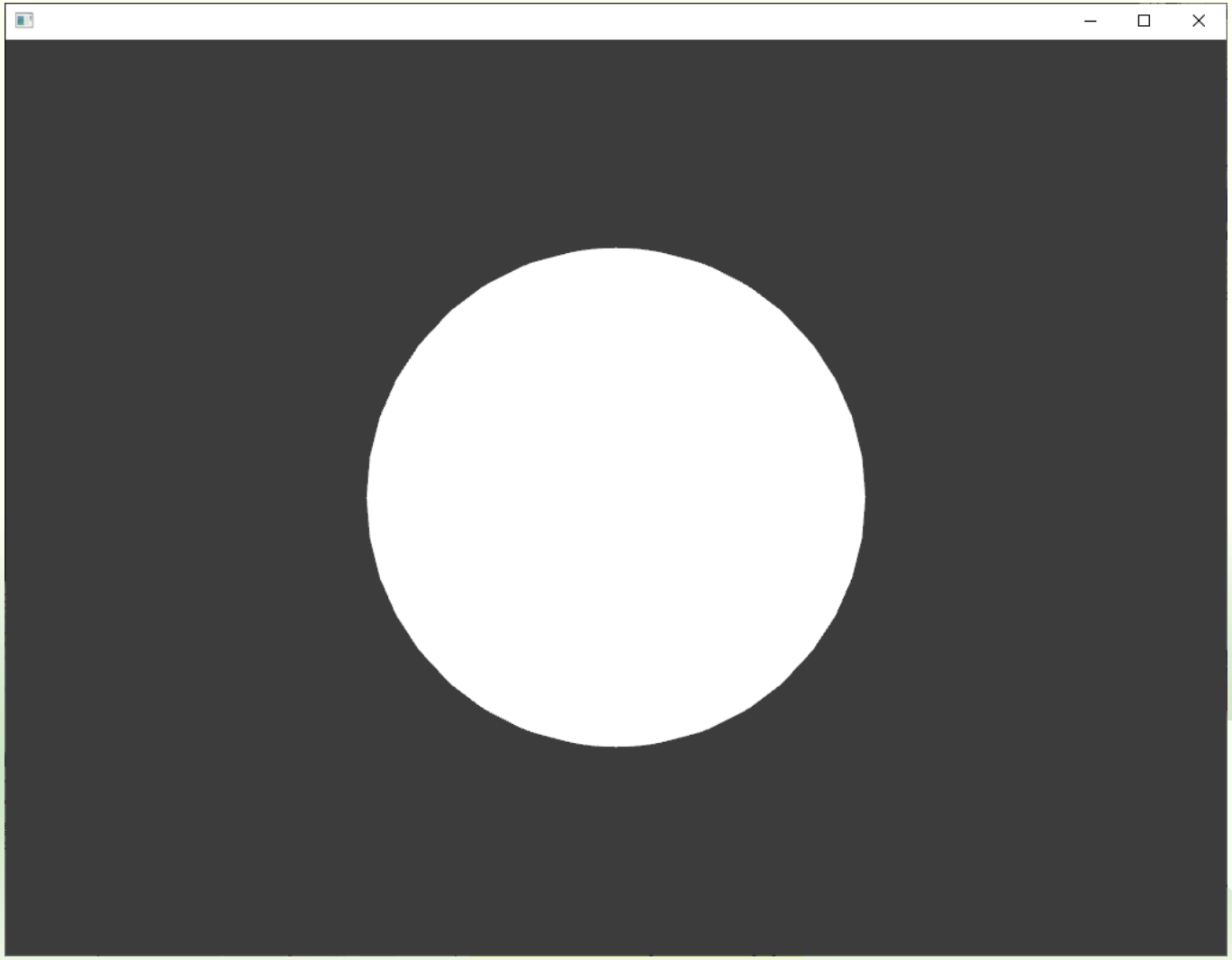
カメラの位置



ビルドと実行


ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) テスト(S) 分析(N) ツール(T) 拡張機能(X) ウィンドウ(W) ヘルプ(H)
検索 (Ctrl+Q)
Debug
Win32
ローカル Windows デバッガー
自動
Live Share
ソリューション エクスプローラー
ソリューション エクスプローラー の検索 (Ctrl+;)
ソリューション 'myDynamicSketch' (2/2 プロジェクト)
myDynamicSketch
参照
外部依存関係
src
main.cpp
ofApp.cpp
ofApp.h
icon.rc
openframeworksLib
ofApp.cpp
myDynamicSketch
(グローバル スコープ)
#include "ofApp.h"
//--
void ofApp::setup(){
camera.setPosition(0.0f, 0.0f, 100.0f);
}
//--
void ofApp::update(){
}
//--
void ofApp::draw(){
camera.begin();
ofDrawSphere(0.0f, 0.0f, 30.0f);
camera.end();
}
//--
void ofApp::keyPressed(int key){
}
//--
void ofApp::keyReleased(int key){
100 %
問題は見つかりませんでした
行: 1 文字: 1 SPC CRLF
準備完了
1 0 myDynamicSketch step03

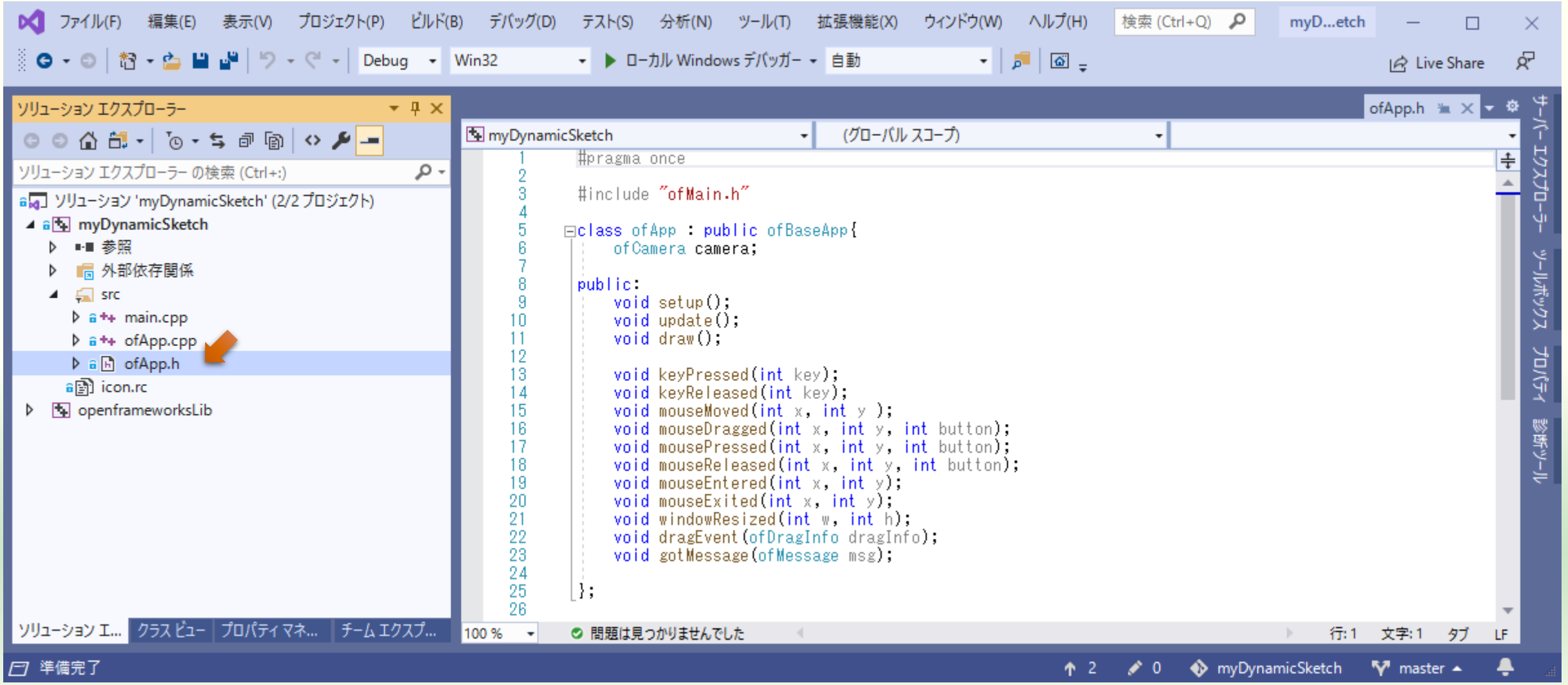
白い円が描かれる



setPosition() メソッド

- void ofNode::setPosition(float px, float py, float pz)
  - px, py, pz: カメラの３次元空間中の位置
- void ofNode::setPosition(const glm::vec3 &p)
  - p: カメラの３次元空間中の位置
  - glm::vec3 は x, y, z の３要素のベクトル
- **ofNode** は３次元のシーンを構成するためのクラス
  - ofCamera のほか３次元のシーンに配置する「部品」の多くはこの ofNode クラスを**継承**（性質を受け継ぐこと）して定義されている
  - 継承元のクラスは**基底クラス**、継承先のクラスは**派生クラス**

また ofApp.h を開く

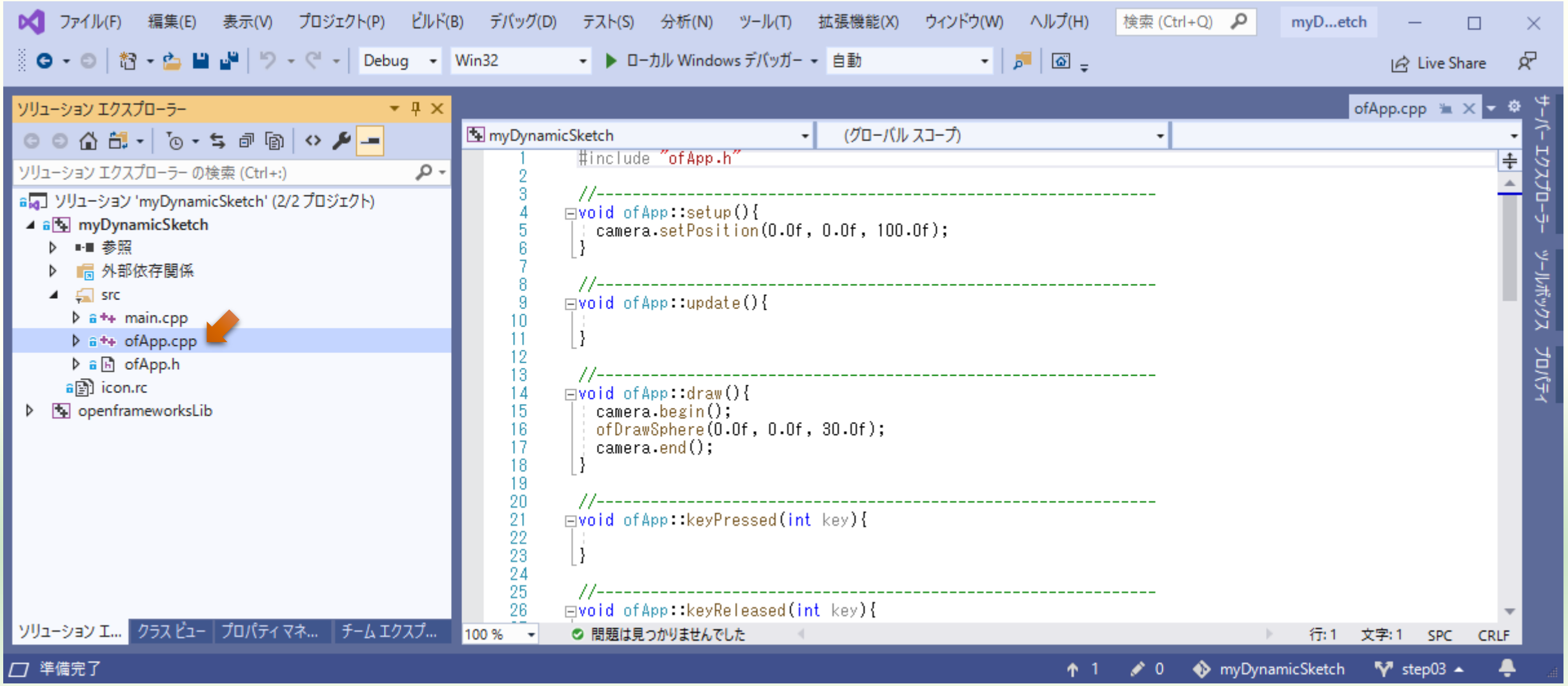


ofApp クラスにライトのメンバ変数を追加する

```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofCamera camera;
  ofLight light;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
  （以下略）
```

- **ofLight** は 3 次元空間中の光源のクラス
  - 変数 light は ofLight クラスのインスタンス

そして ofApp.cpp を開く

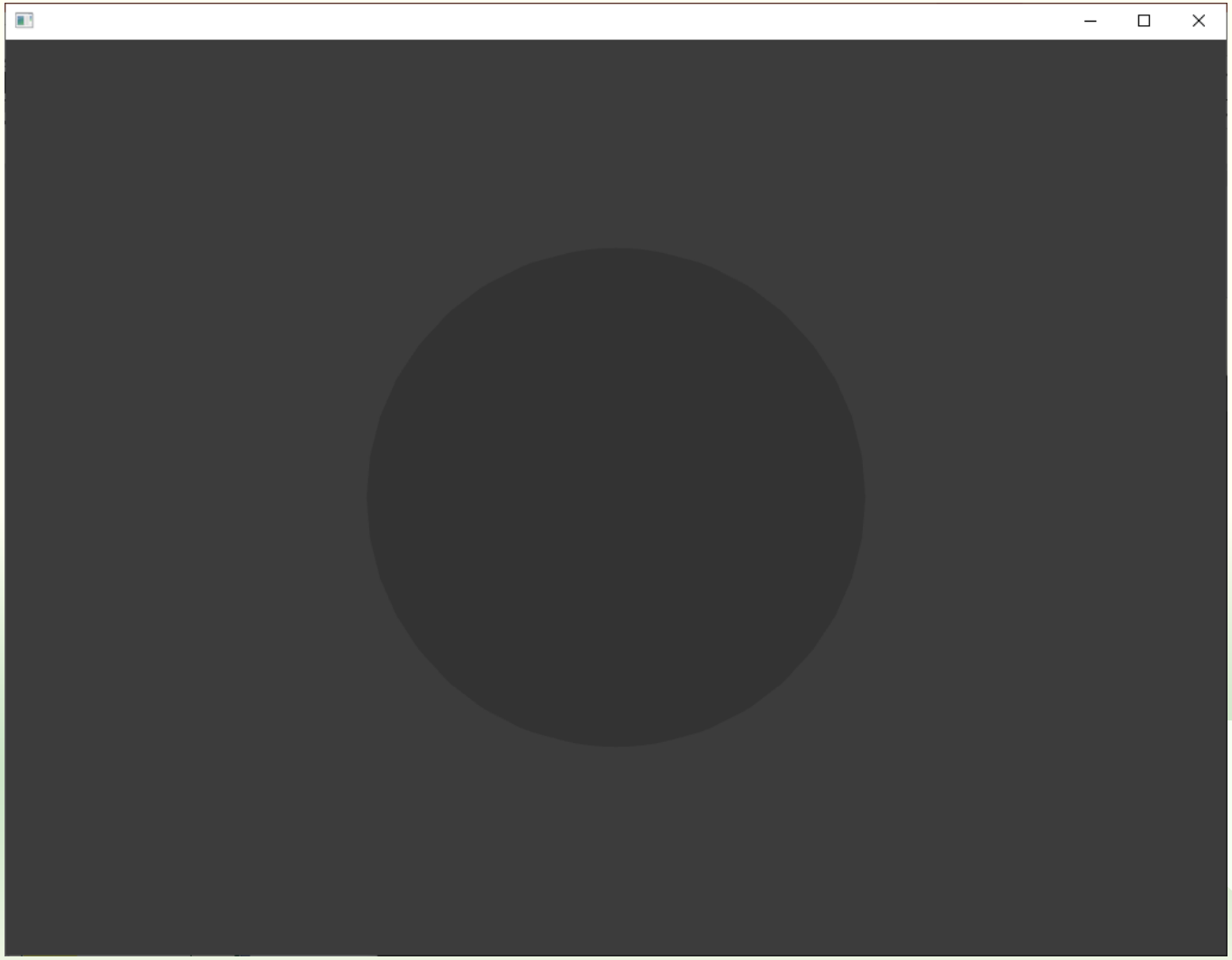


ライトをつける

```
//--------------------------------------------
void ofApp::setup(){
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.enable();
}

//--------------------------------------------
void ofApp::update(){

}

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  ofDrawSphere(0.0f, 0.0f, 30.0f);
  camera.end();
}
```

- light の enable() はライトを点灯するメソッド
- 消すときは disable()
  - つけっぱなしにするなら setup() で enable() する
  - 付けたり消したりしたいときは update() か draw() で必要に応じて enable() か .disable() する

逆に暗い

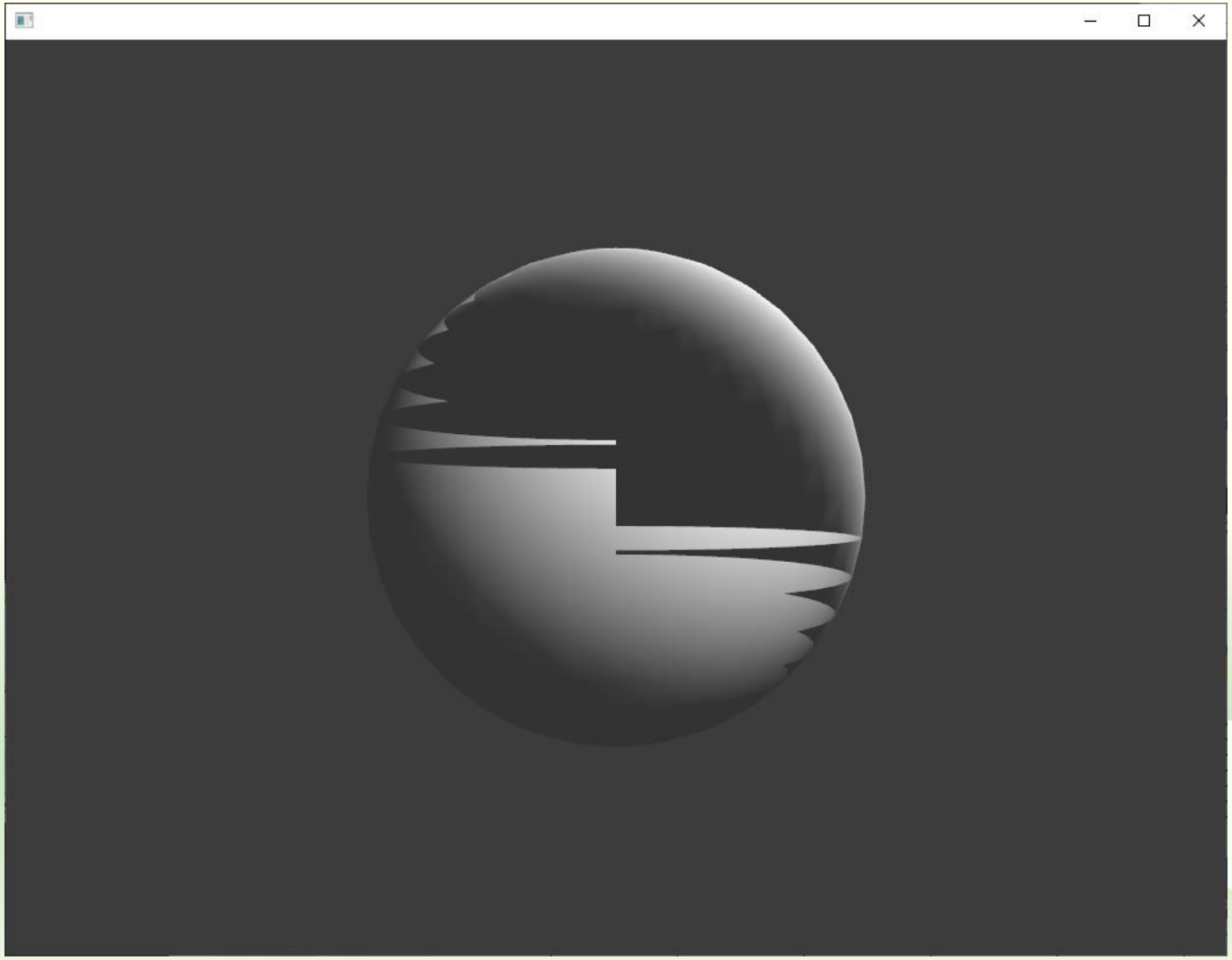


ライトを右上の方に持って行く

```
//--------------------------------------------
void ofApp::setup(){
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();
}

//--------------------------------------------
void ofApp::update(){

}

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  ofDrawSphere(0.0f, 0.0f, 30.0f);
  camera.end();
}
```

- ofLight クラスも ofNode クラスを継承している
  - setPosition() メソッドで位置を指定できる
- light は最初は原点にあった
  - つまり球の内部にあった
  - そのため球の表には光が当たってなかった

明るくなったけど変

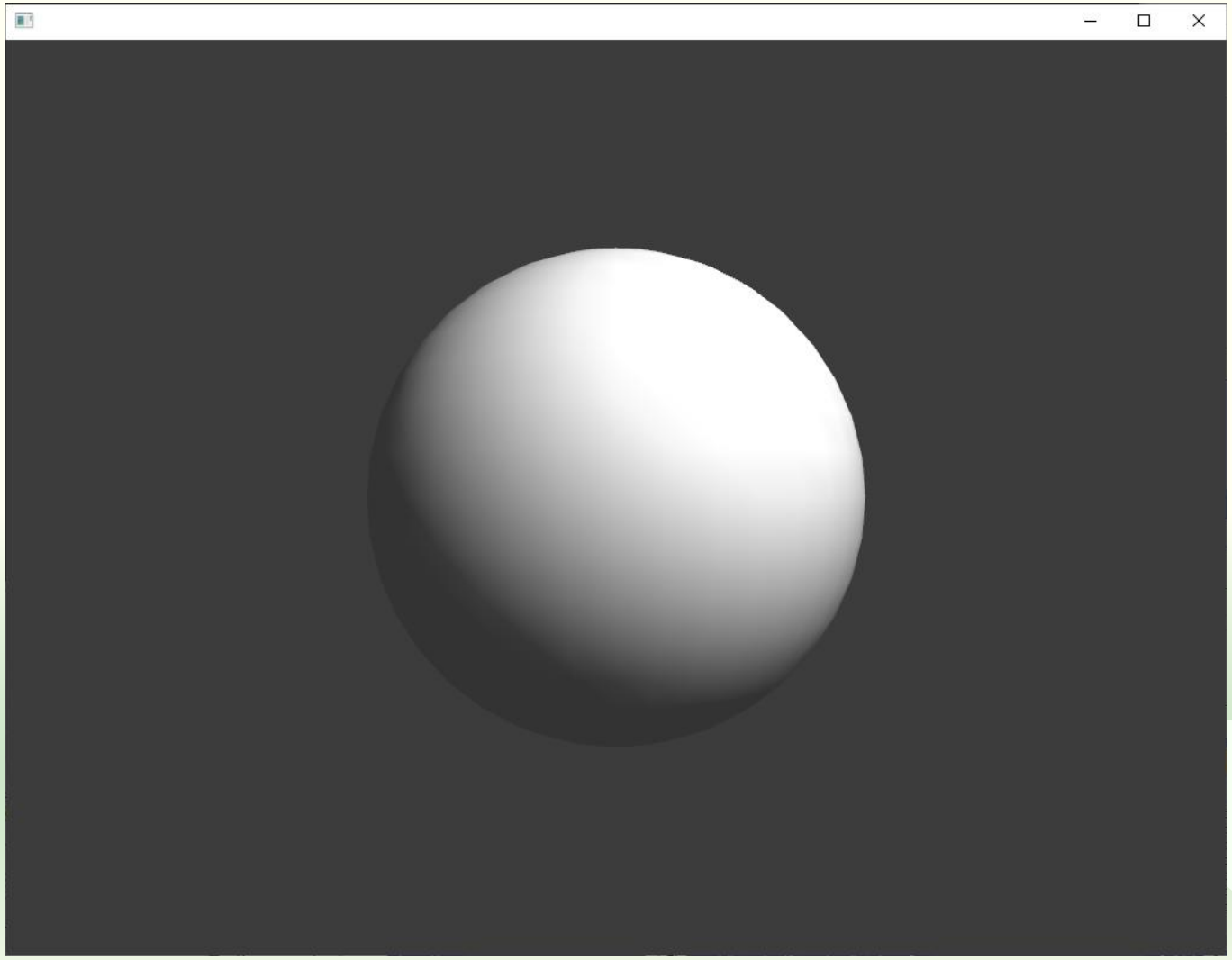


見えないはずの面を描かないようにする

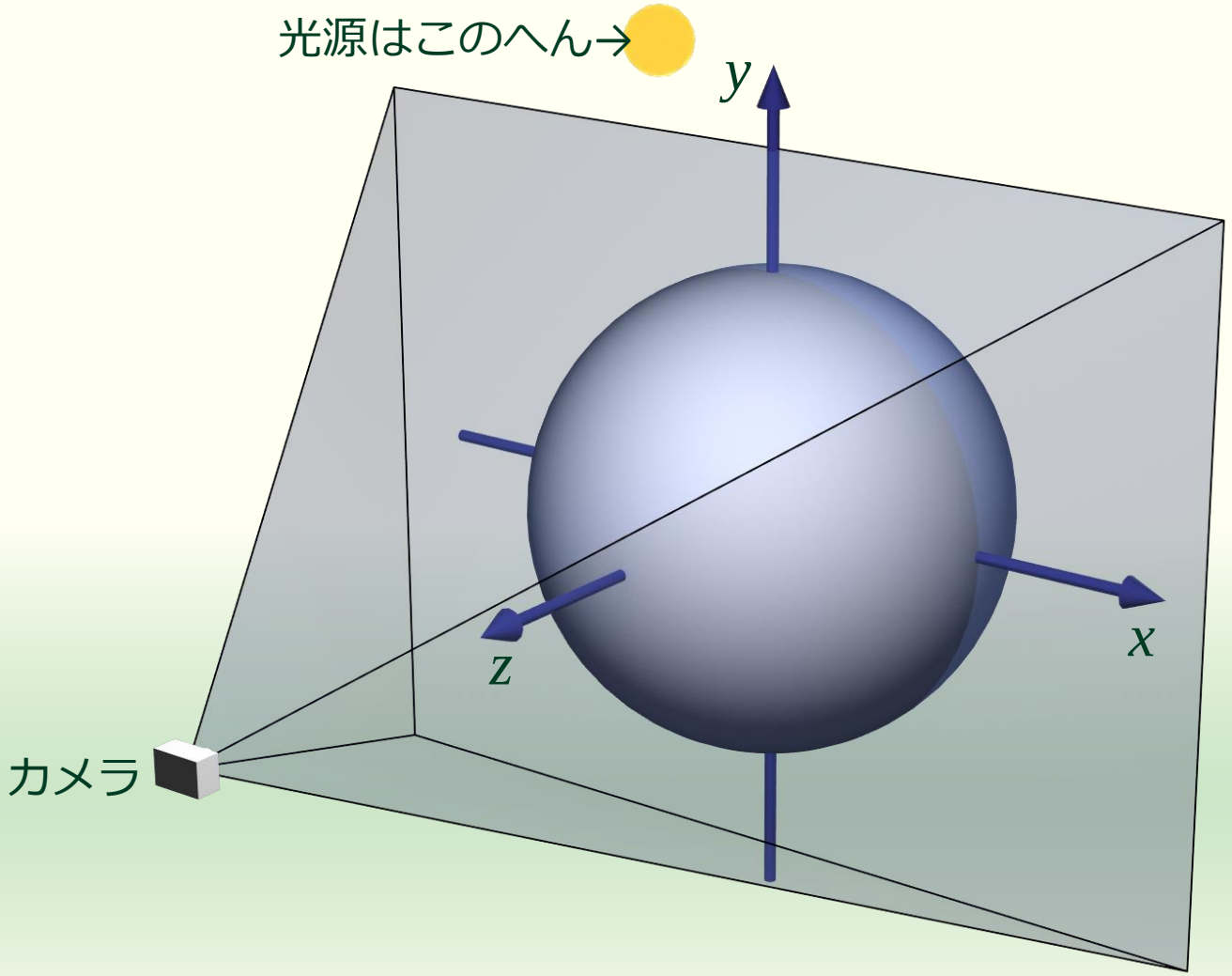
```
//--------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();
}

//--------------------------------------------
void ofApp::update(){

}

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  ofDrawSphere(0.0f, 0.0f, 30.0f);
  camera.end();
}
```

- ３次元ＣＧで順序を考えずに面を描くと手前の面の上に奥の面が後から重ね描きされてしまう
- 面の**奥行き**を画素単位に比較して手前の面だけが描かれるようにする
- **隠面消去処理**という

ついにきれいに描けた



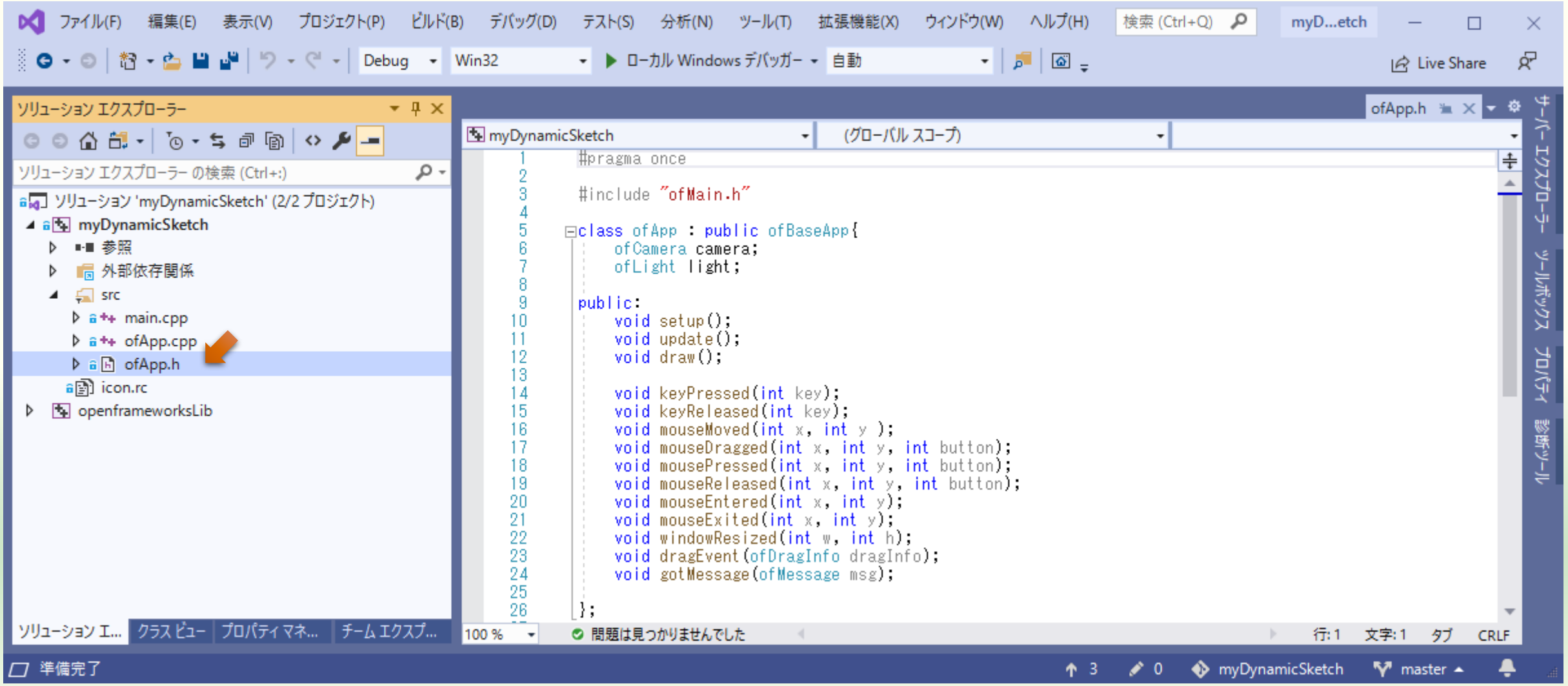
カメラ、光源、球の位置関係



オブジェクト

オブジェクトを使ってオブジェクト（物体）を描く

またまた ofApp.h を開く

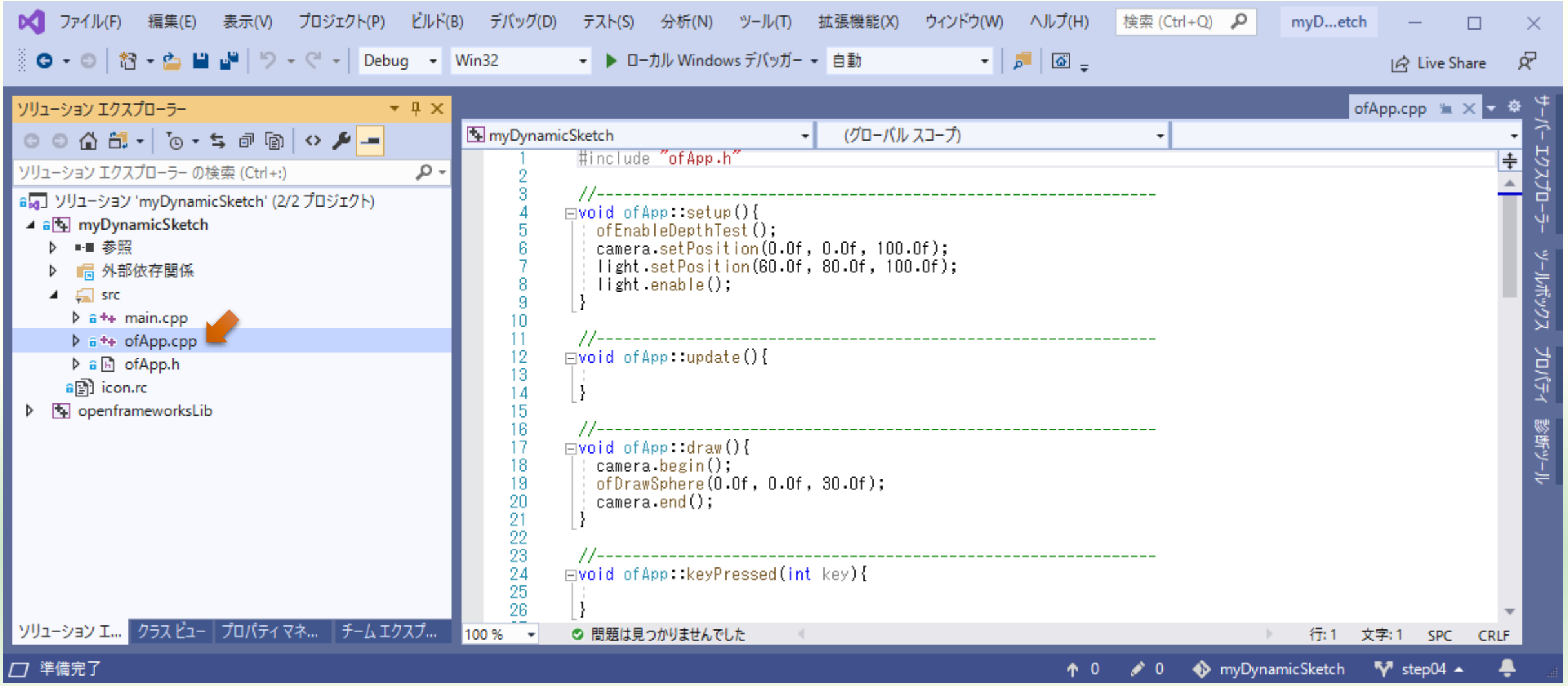


ofApp クラスに球のメンバ変数を追加する

```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofCamera camera;
  ofLight light;
  ofSpherePrimitive sphere;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
  （以下略）
```

- **ofSpherePrimitive** は球の（物体という意味での）オブジェクトの**クラス**
- sphere は
  - 「球という**物体**」という意味でのオブジェクトで
  - 「メモリを与えられた**実体**」という意味でのオブジェクトでもある

そしてもう一度 ofApp.cpp を開く



球の半径と解像度を設定して描画する

```
//--------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  sphere.set(30.0f, 20);
}

 (途中略)

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  sphere.draw();
  camera.end();
}
```

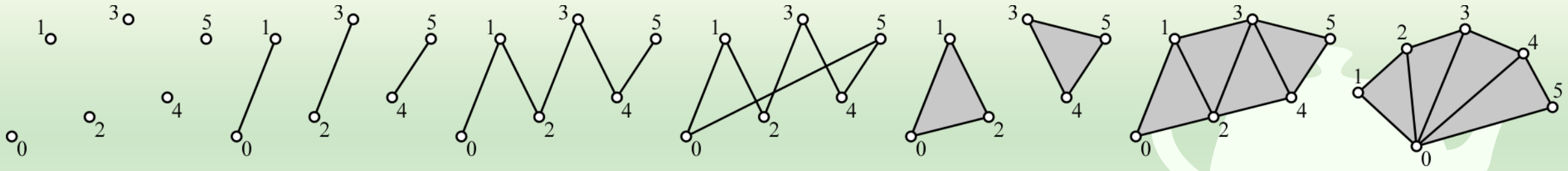
半径

解像度

- set() メソッドを使って半径と解像度を指定する
  - setRadius() メソッドで半径だけを指定できる
  - setResolution() メソッドで解像度だけを指定できる
- draw() メソッドにより図形の描画が行われる

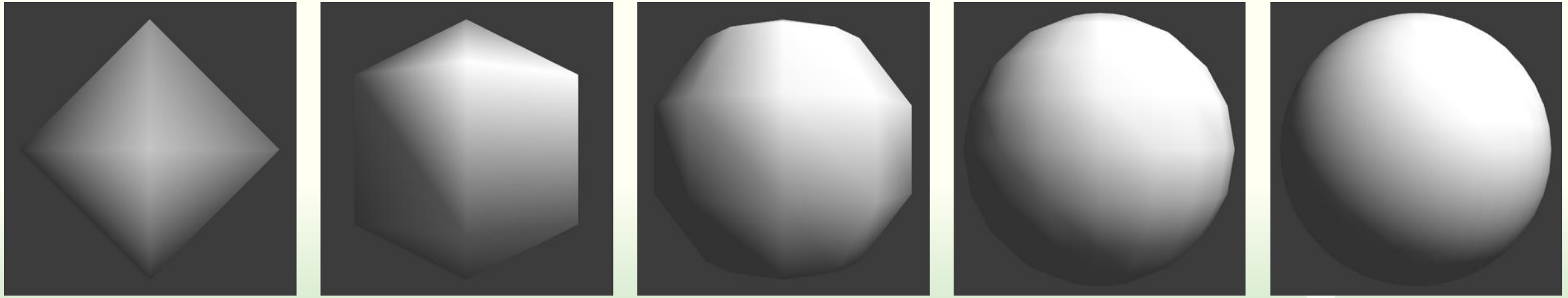
ofSpherePrimitive の set() メソッド

- void ofSpherePrimitive::set(float radius, int resolution, ofPrimitiveMode mode=OF_PRIMITIVE_TRIANGLE_STRIP)
  - radius: 球の半径
  - resolution: 球のメッシュの解像度
  - mode: 球を描画するのに使う図形要素、デフォルトの OF_PRIMITIVE_TRIANGLE_STRIP を使えばよいので省略する

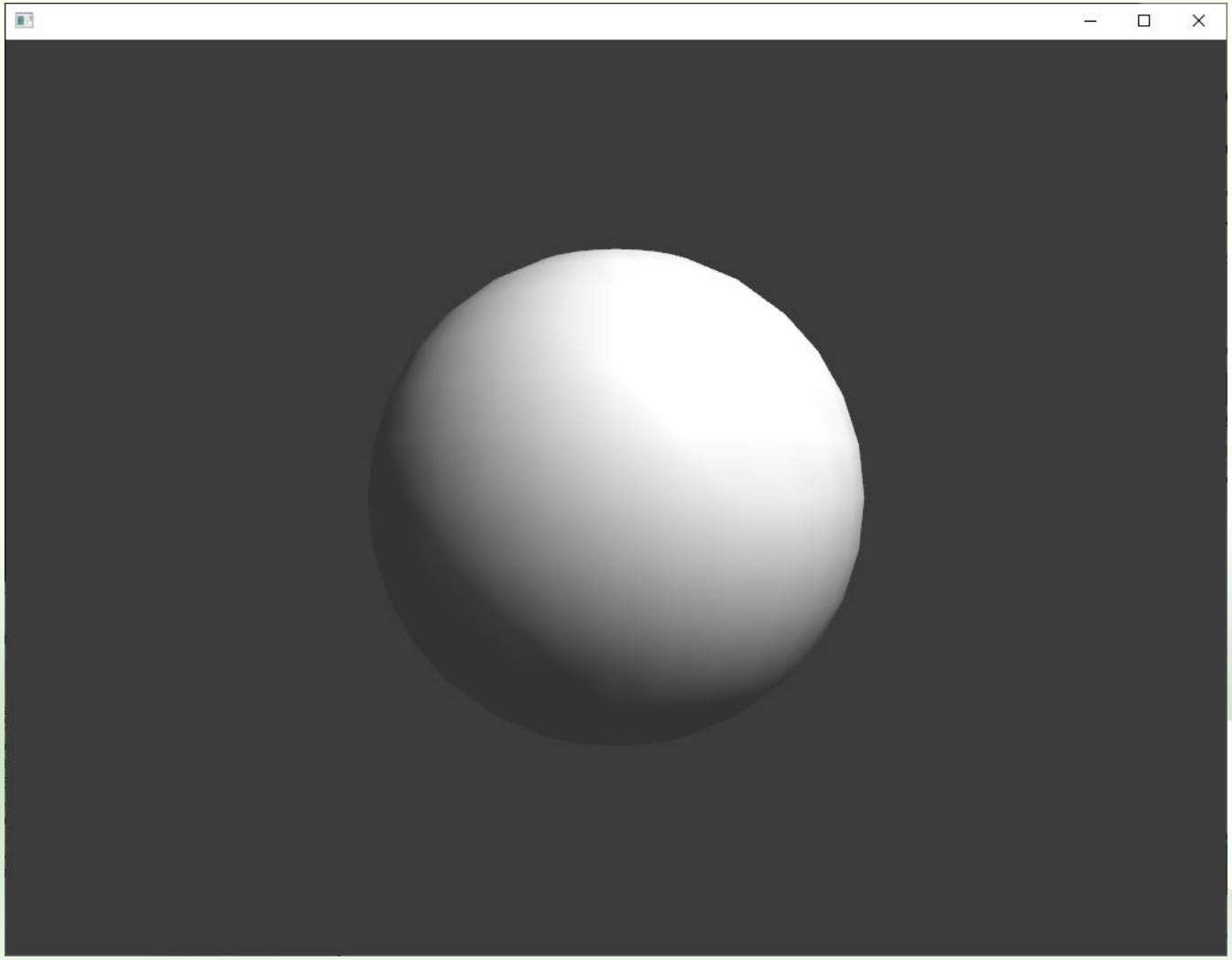


他にOF_PRIMITIVE_LINES_ADJACENCY, OF_PRIMITIVE_LINE_STRIP_ADJACENCY, OF_PRIMITIVE_TRIANGLES_ADJACENCY, OF_PRIMITIVE_TRIANGLE_STRIP_ADJACENCY, OF_PRIMITIVE_PATCHES

球の解像度 (resolution)



結果は同じ



関数とオブジェクト

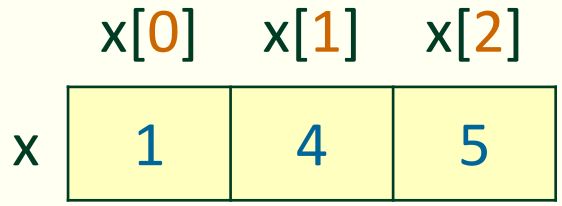
- 関数
  - void ofDrawSphere(float x, float y, float radius) は x, y, radius を指定することにより図形（球）を描画する
  - 関数は球の描き方を知っているが、どういう球を描けばいいか引数に（描画処理とは別に管理する）データを与える必要がある
- オブジェクト
  - sphere はそれ自体が球の描き方（メンバ関数＝メソッド）と、それに必要なデータ（メンバ変数）を保持している
  - オブジェクトは描くという行為を含めてデータ化できる

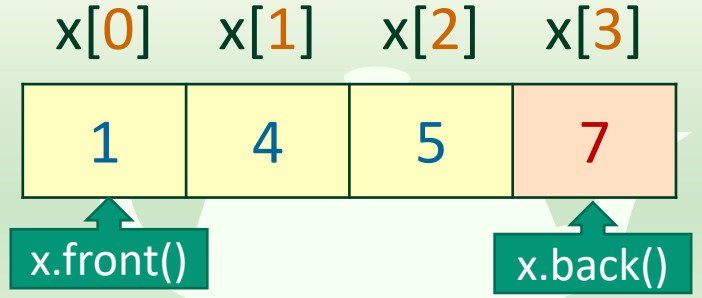
可変長配列

std::vector を使って複数のオブジェクトを描く

可変長配列 std::vector

- 複数のデータを保持できる（配列）
  - vector<int> x { 1, 4, 5 };
- 各要素は添え字でアクセスできる
  - x[0] → 1、x[1] → 4、x[2] → 5、x.size() → 3
- データを後から追加できる（可変長）
  - x.push_back(7);
    - x[3] → 7、x.size() → 4
    - x.front() → x[0]、x.back() → x[3]



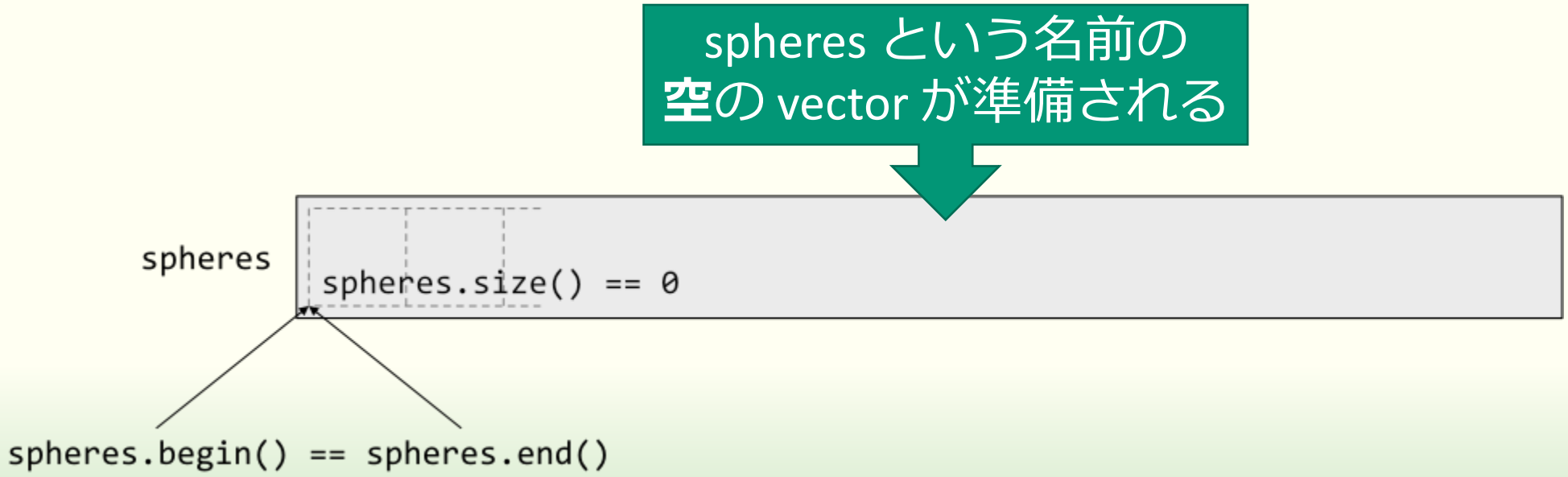


ofApp.h で球を保持する vector を用意する

```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofCamera camera;
  ofLight light;
  vector<ofSpherePrimitive> spheres;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
  （以下略）
```

- ofSpherePrimitive クラス（球）を **vector** にする
  - 複数の ofSpherePrimitive クラスのオブジェクトを保持する**コンテナ**（入れ物）ができる
  - 複数の球が入るので変数名は **spheres** にしている
- vector は標準ライブラリ
  - ofMain.h に using namespace std; が入っているので std:: は省略可能

vector<ofSpherePrimitive> spheres;



ofApp.cpp の setup() で vector に球を追加する

```
//--------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  ofSpherePrimitive sphere{ 10.0f, 20 };
  spheres.push_back(sphere);
}
```

半径

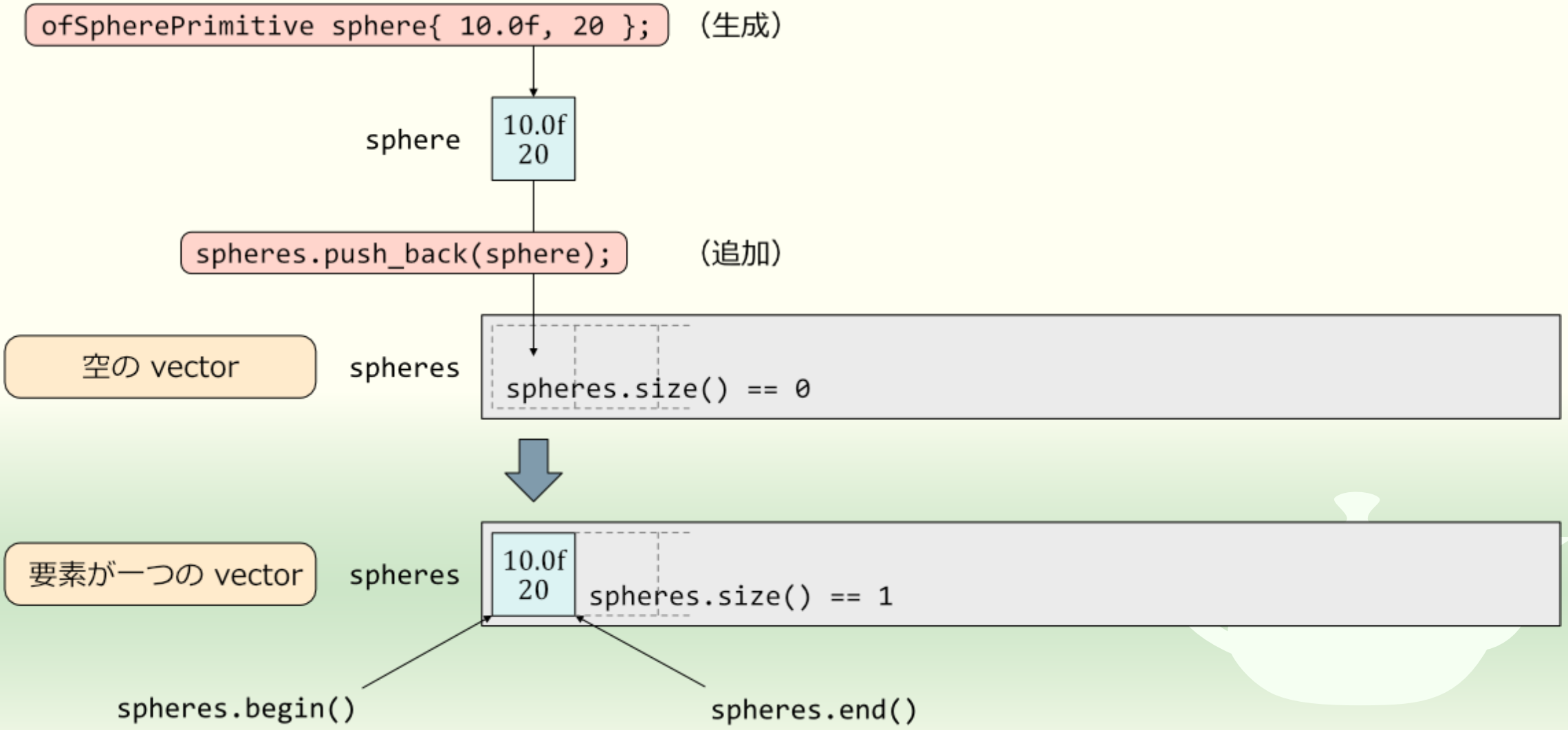
解像度

生成

spheres の
最後に追加

- setup() で球のオブジェクトを一つ生成する
  - ofSpherePrimitive クラスは変数宣言で**半径**と**解像度**を指定できる
- vector の spheres に生成した球を追加する
  - 生成されたオブジェクトのデータは vector の最後に追加されたメモリにコピーされる

spheres という vector に sphere を追加する



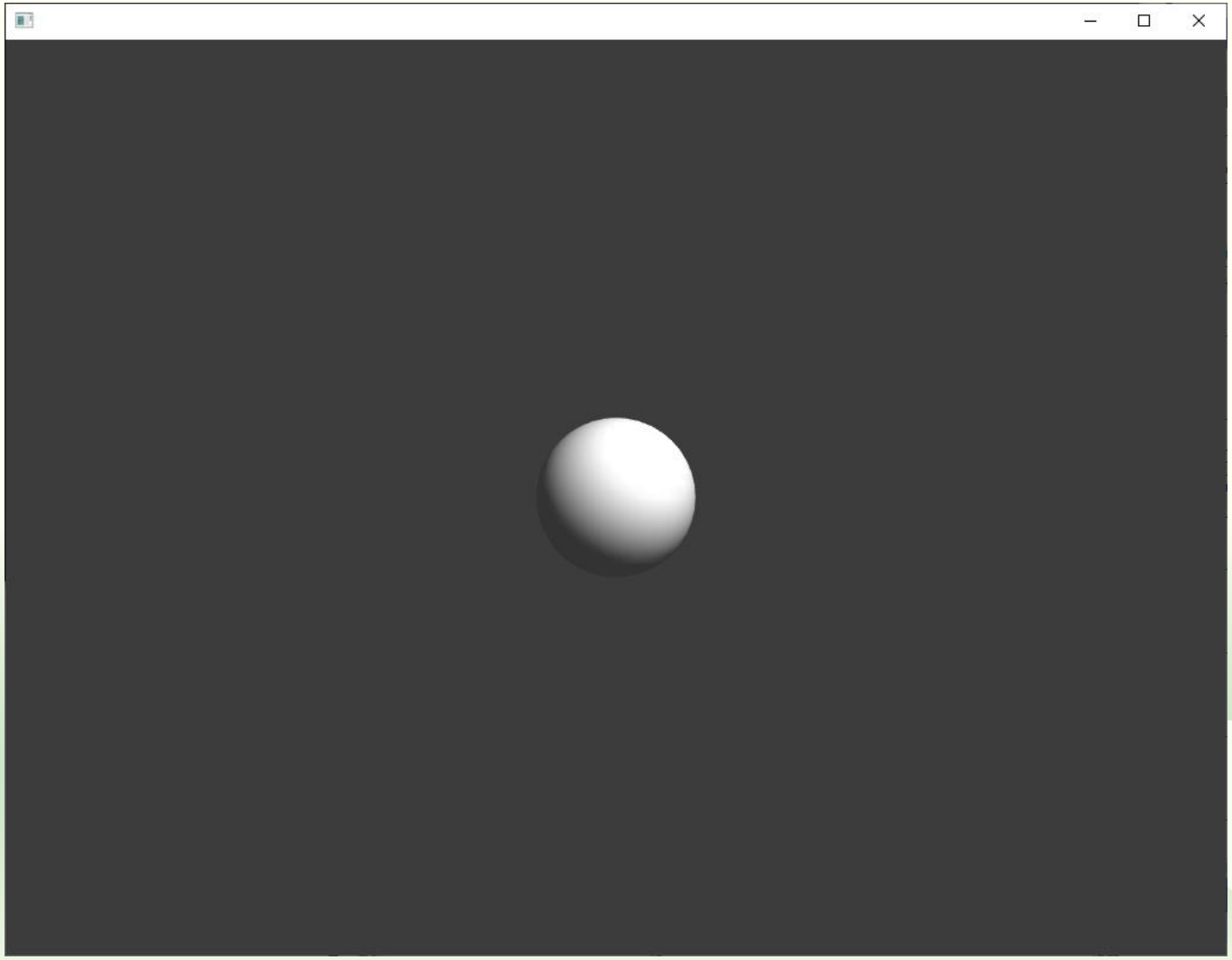
ofApp.cpp の draw() で vector の球を描画する

```
//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  spheres.back().draw();
  camera.end();
}
```

spheres の最後の要素

- vector の spheres に最後に追加した球を描画する
  - **back()** メソッドは vector の最後の要素を取り出す
  - まだ球が一つしか入っていないので spheres.front() でも spheres[0] でも同じ
  - spheres[1] とか spheres[2] とかは実行時にエラーになる
    - spheres.size() で得られる vector の要素数よりも小さくないといけない

球が 1 個描かれる



setup() で球を上に移動して回転する

```
using namespace glm;

//--------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  ofSpherePrimitive sphere{ 10.0f, 20 };
  sphere.move(0.0f, 40.0f, 0.0f);
  sphere.rotateAroundDeg(60.0f,  // 回転角
    vec3{ 0.0f, 0.0f, 1.0f },    // 回転軸
    vec3{ 0.0f, 0.0f, 0.0f });   // 回転中心
  spheres.push_back(sphere);
}
```

- using namespace glm; を入れて vec3 を使うときに glm:: を省略できるようにする
- move() メソッドで球の高さを少し上げる
- rotateAroundDeg() メソッドは第３引数の位置を通る第２引数のベクトルを回転軸として第１引数の角度だけ回転する

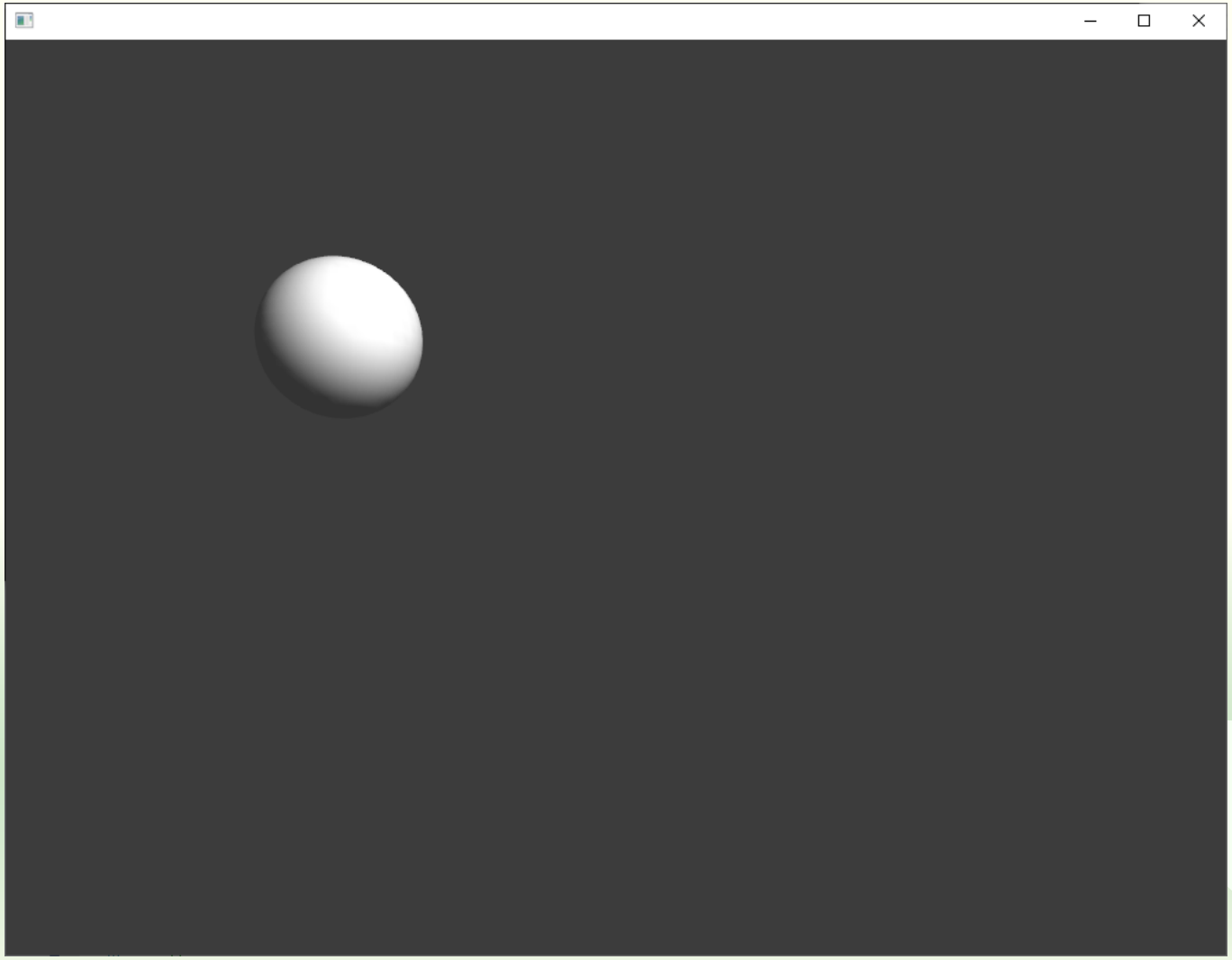
move() メソッド

- void ofNode::move(float x, float y, float z)
  - x, y, z: 物体の平行移動量
- void ofNode::move(const glm::vec3 &offset)
  - offset:物体の平行移動量のベクトル（vec3{ x, y, z }）
- 物体を**現在の位置から** x, y, z だけ平行移動する
  - setPosition() は指定した位置に配置する

rotateAroundDeg() メソッド

- void ofNode::rotateAroundDeg(float degrees, const glm::vec3 &axis, const glm::vec3 &point)
  - degrees: 回転角（度）
  - axis: 回転軸のベクトル（vec3{ vx, vy, vz }）
  - point: 回転中心（vec3{ px, py, pz }）
- 物体の姿勢を point を通り axis の方向を向いた軸を中心に degrees 度回転する
  - 回転角にラジアンが使いたければ rotateAroundRad() を使う

球が上に移動して左に傾く

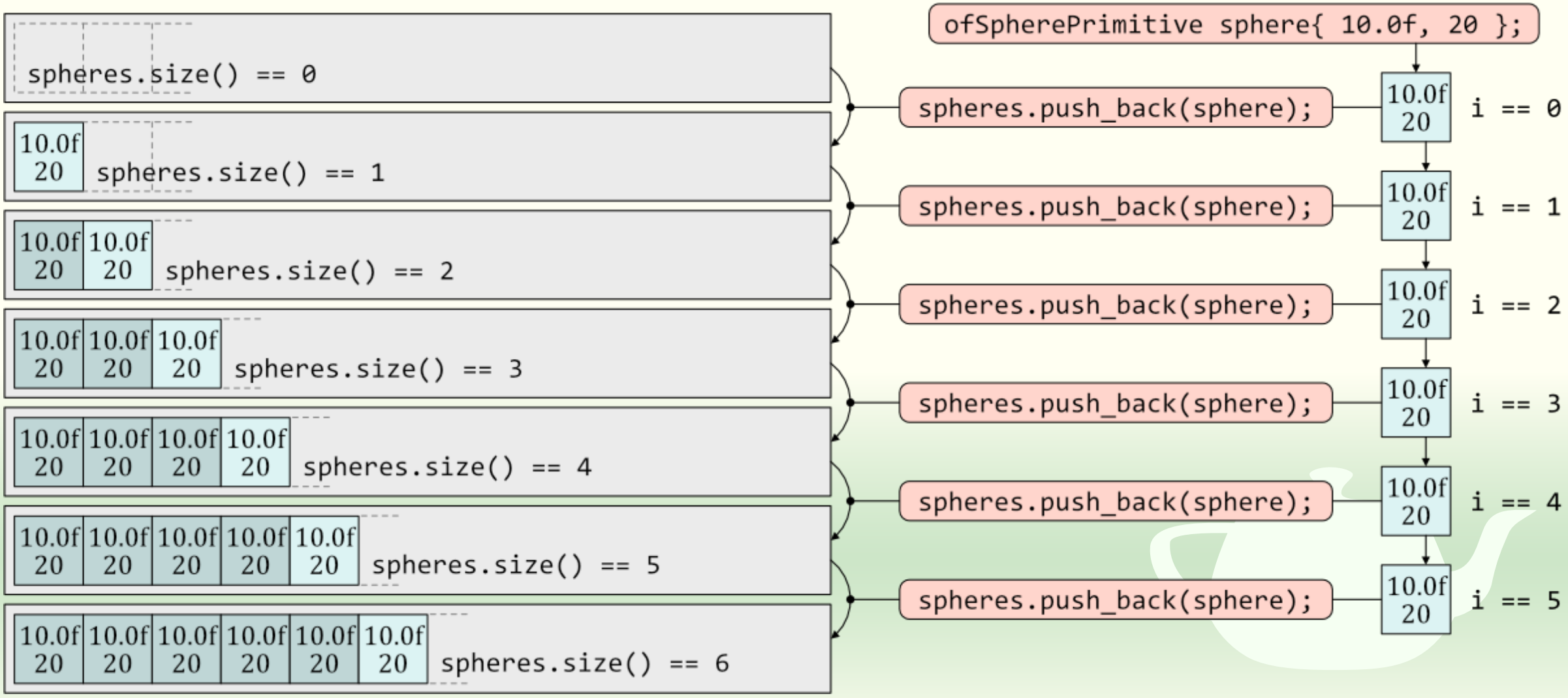


複数の球を回転しながら spheres に追加する

```
using namespace glm;

//--------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  for (int i = 0; i < 6; ++i){
    ofSpherePrimitive sphere{ 10.0f, 20 };
    sphere.move(0.0f, 40.0f, 0.0f);
    sphere.rotateAroundDeg(60.0f * i,
      vec3{ 0.0f, 0.0f, 1.0f },
      vec3{ 0.0f, 0.0f, 0.0f });
    spheres.push_back(sphere);
  }
}
```

- sphere の設定を行ってから push_back() する
  - 設定された内容で sphere が spheres に追加される
  - 角度が同じだと重なってしまうので60°ずつずらす
- 6 回繰り返す
  - i は 0, 1, 2, 3, 4, 5 と変化するので **60 * i** は 0, 60, 120, 180, 240, 300 になる

spheres に複数の sphere を push_back() する



for 文による繰り返し処理

例

```
// 合計
int sum = 0;

// 1 から 5 までの合計を sum に求める
for (int i = 1; i <= 5; ++i){
   sum += i;
}
```

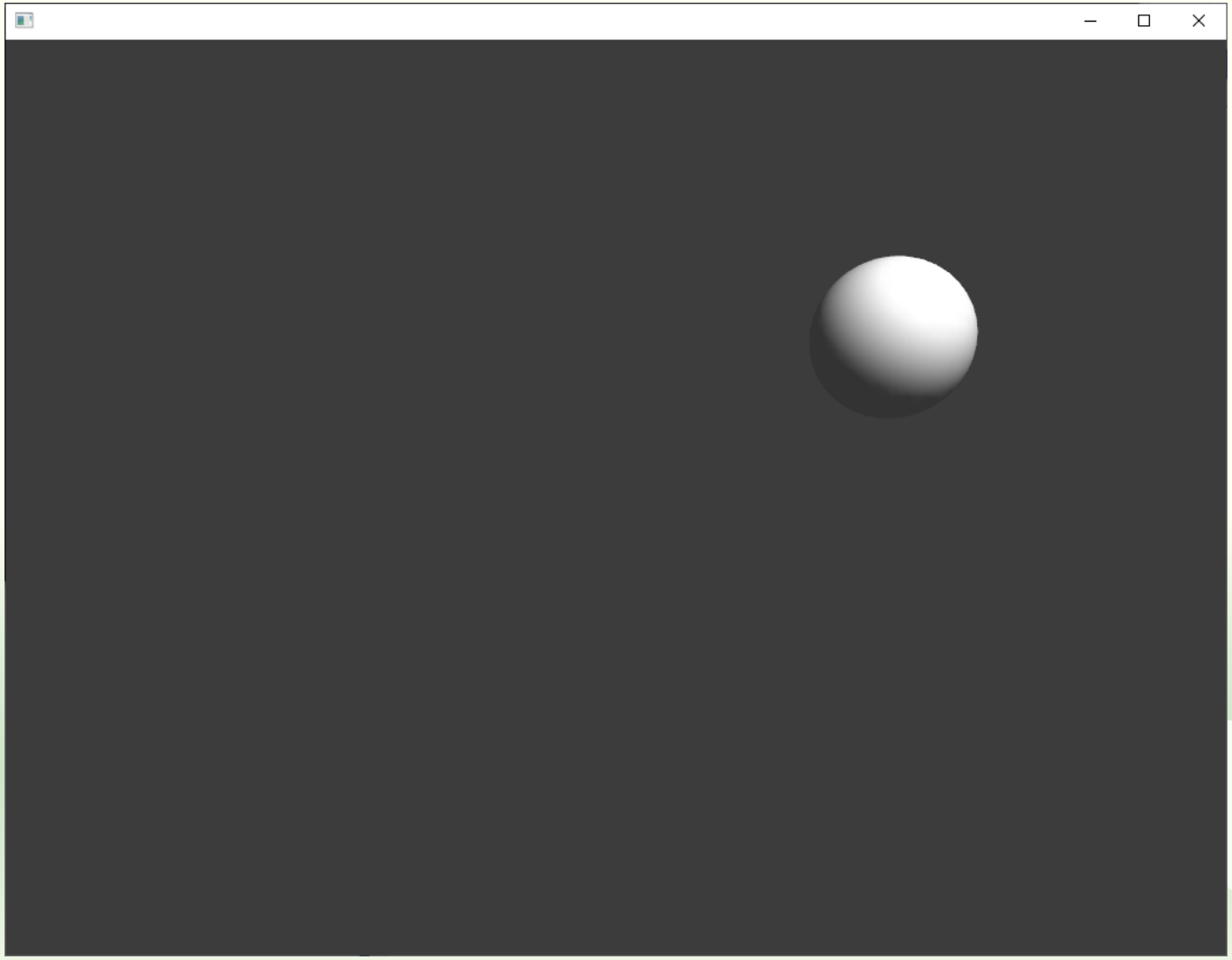
この処理が5回繰り返される

sum=0　i=1

sum	i	i <= 5	sum += i	++i
0	1	true	0+1	1+1
1	2	true	1+2	2+1
3	3	true	3+3	3+1
6	4	true	6+4	4+1
10	5	true	10+5	5+1
15	**6**	**false**	**おわり**	

- `int i = 1`
  - int 型の変数 i を宣言して 1 に初期化する
- `i <= 5`
  - i <= 5 が **true**（i が 5 以下）の間 { } 内を**繰り返す**
- `++i`
  - i に 1 を加算する

しかし最後の１個しか描いてくれない



draw() で vector が保持する全部の球を描画

```
//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  for (auto &sphere : spheres){
    sphere.draw();
  }
  camera.end();
}
```

- vector の spheres には複数の球が入っている
- spheres から球を一つずつ sphere に取り出して描画する
  - sphere に**参照演算子 &** をつけているので sphere に入っているのは spheres の一つの要素の実体になる

vector で範囲ベースの for を使う

例

```
// 合計
int sum = 0;

// データ
std::vector<int> data{ 1, 2, 3, 4, 5 };

// data に入っている全部の数値の合計を sum に求める
for (auto &x : data){
   sum += x;
}
```

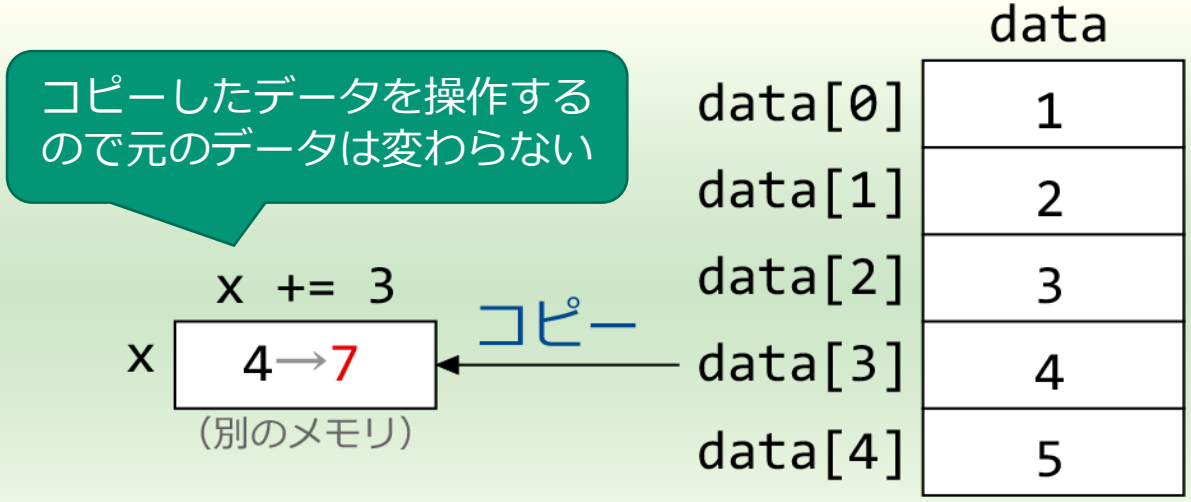
この処理がデータの個数回繰り返される

- data の要素を一つずつ取り出して x に入れる
- { } 内の処理をデータの数だけ繰り返す
- **auto**
  - **型推論**
  - 文脈から自動的にデータ型を推定する機能
  - 例のプログラムでは vector である data の一つの要素のデータ型 (int) になる

代入と参照

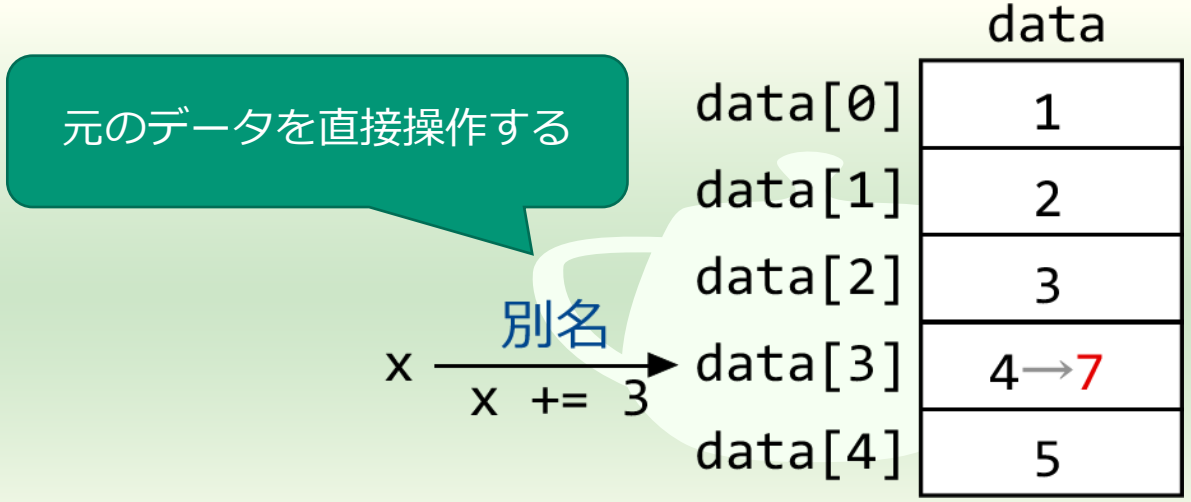
代入はデータのコピー

```
std::vector<int> data { 1,2,3,4,5 };

for (auto x : data){
  x += 3;
}
```

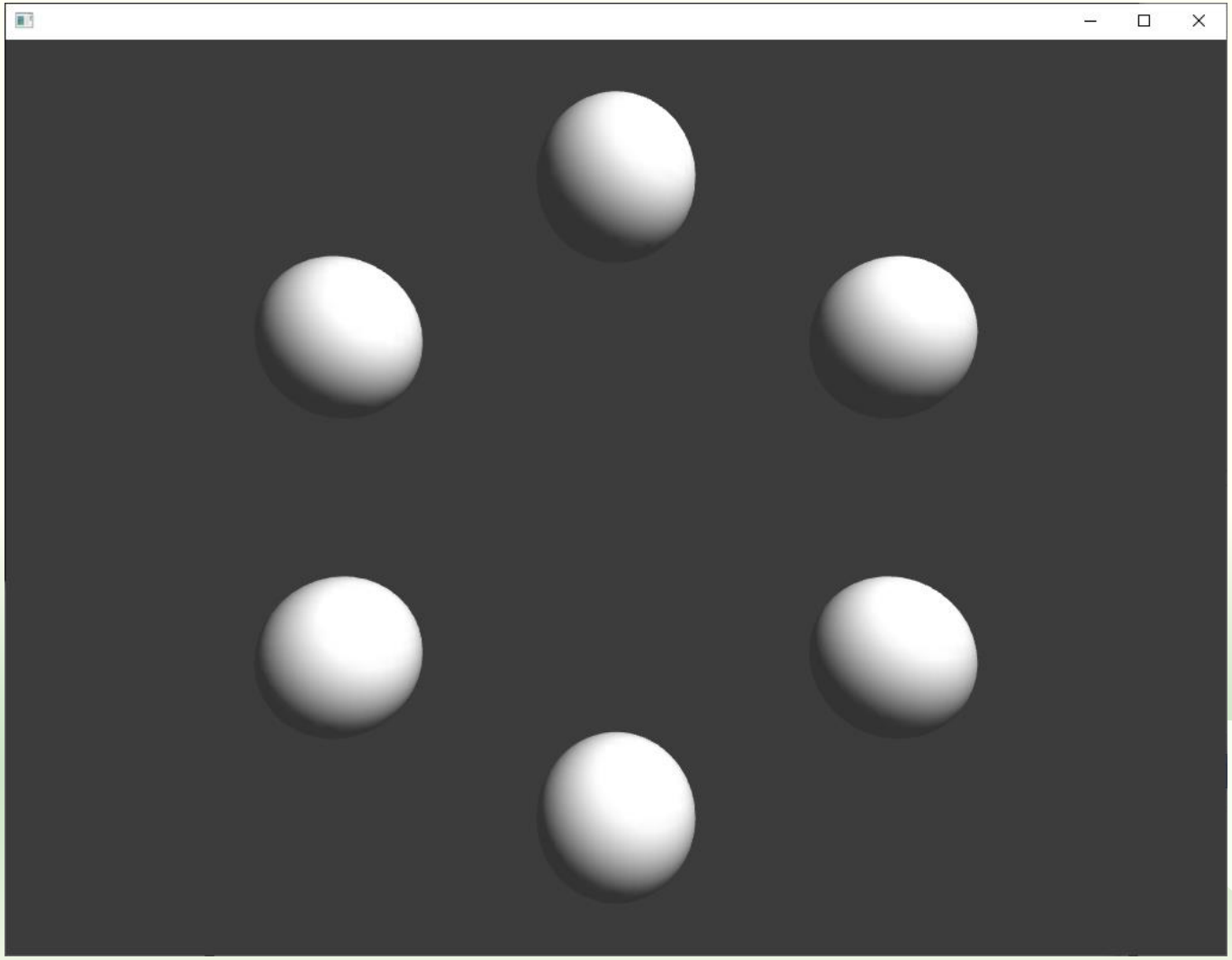


参照は格納場所の共有

```
std::vector<int> data { 1,2,3,4,5 };

for (auto &x : data){
  x += 3;
}
```



全部の球が描かれる



emplace_back() を使う

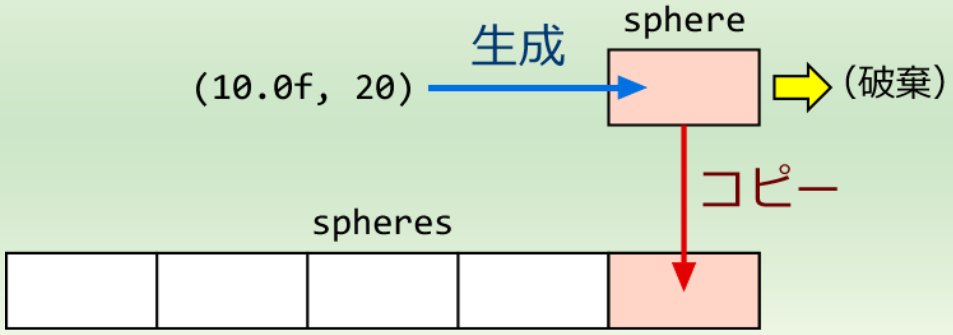
```
using namespace glm;

//--------------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  for (int i = 0; i < 6; ++i){
    spheres.emplace_back(10.0f, 20);
    spheres.back().setPosition(0.0f, 40.0f, 0.0f);
    spheres.back().rotateAroundDeg(60.0f * i,
      vec3{ 0.0f, 0.0f, 1.0f },
      vec3{ 0.0f, 0.0f, 0.0f });
  }
}
```

- emplace_back() メソッドは vector の最後に直接要素を生成する
  - 設定はback() メソッドで最後の要素取り出して行っている
    - ここでは追加する要素を保持する単独の**変数を使っていない**ため

push_back() と emplace_back()

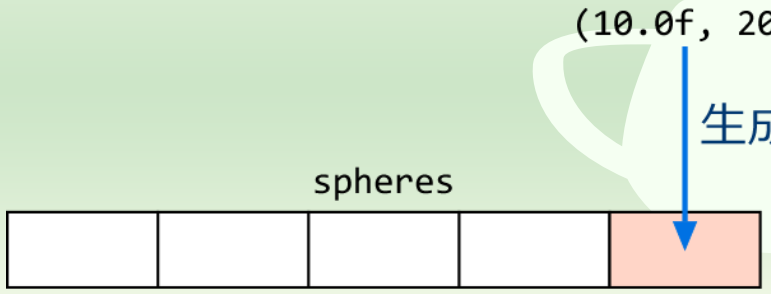
push_back()

```
// vector
std::vector<ofSpherePrimitive> spheres;

// 要素を一つ作成
ofSpherePrimitive sphere(10.0f, 20);

// 作成した要素を vector の最後にコピーして追加
spheres.push_back(sphere);
```

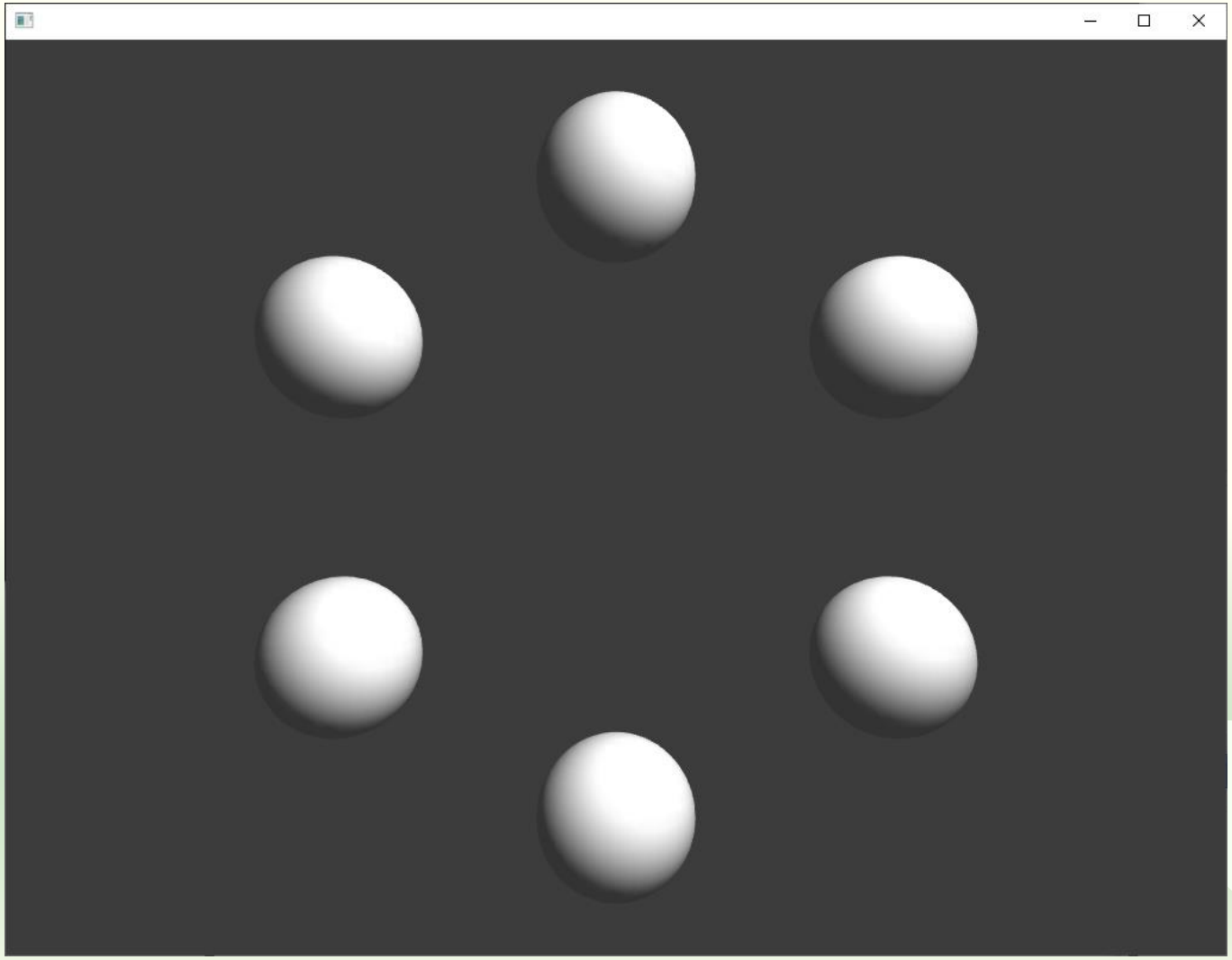


emplace_back()

```
// vector
std::vector<ofSpherePrimitive> spheres;



// vector の最後に要素を作成して追加
spheres.emplace_back(10.0f, 20);
```



特に変わらない



階層構造

図形の親子関係

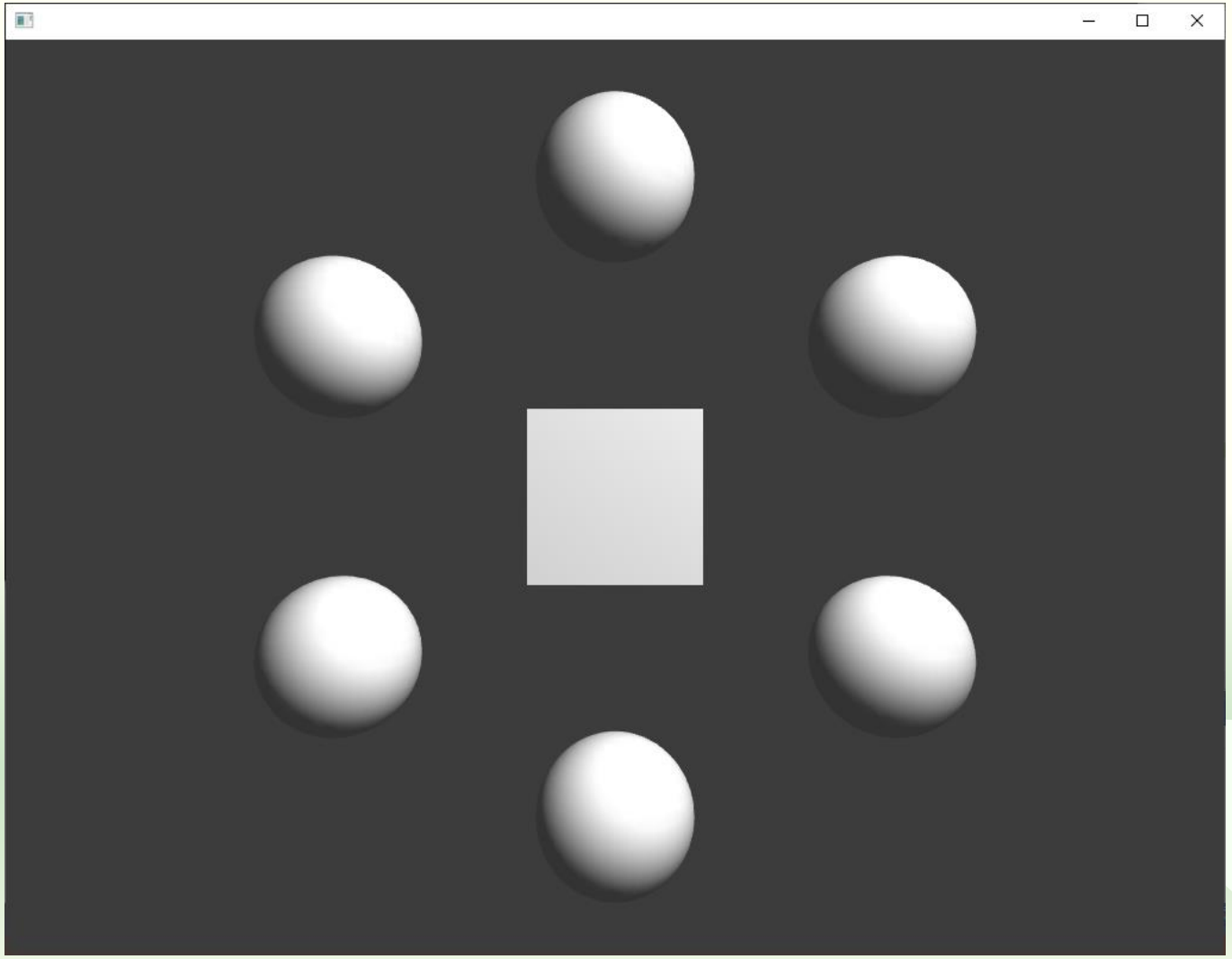
箱を一つ作成して draw() で描画する

幅　高さ　深さ

```
using namespace glm;

static ofBoxPrimitive box{ 20.0f, 20.0f, 20.0f };

 (途中略)

//--------------------------------------------
void ofApp::draw(){
  camera.begin();
  for (auto &sphere : spheres){
    sphere.draw();
  }
  box.draw();
  camera.end();
}
```

- **ofBoxPrimitive** は箱のクラス
  - 変数宣言の時に箱の幅、高さ、深さを指定できる
  - ofApp.h いじるのが面倒なので ofApp.cpp 内に static で宣言する
- draw() で描画する
  - box の draw() メソッド

箱が追加される



update() で箱を回転する

```
using namespace glm;

static ofBoxPrimitive box{ 20.0f, 20.0f, 20.0f };

 (途中略)

//--------------------------------------------
void ofApp::update(){
  box.rotateDeg(1.0f, 0.0f, 0.0f, 1.0f);
}
```

回転角（度）

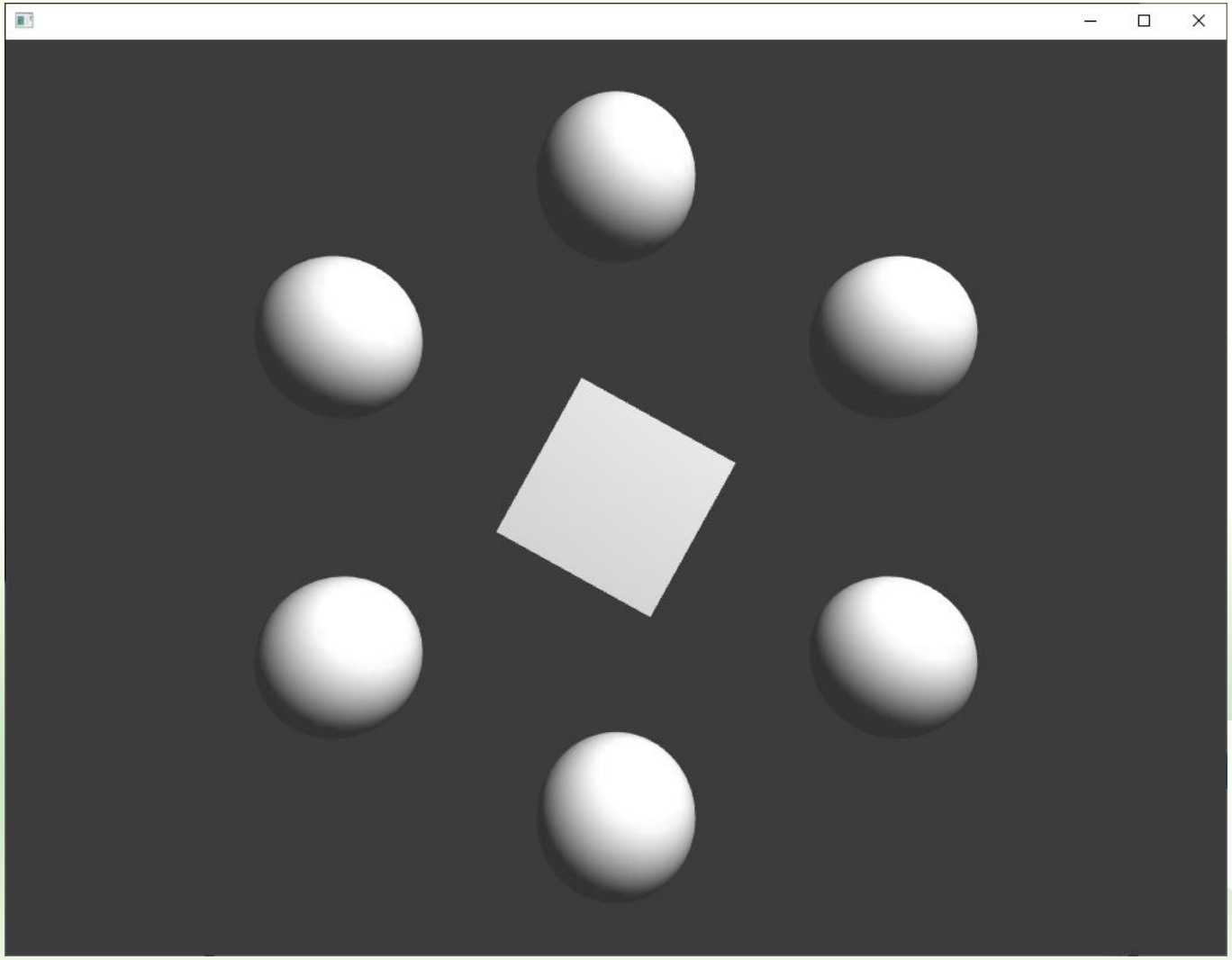
回転軸

- update() で箱を回転する
- rotateDeg() メソッド
  - 物体を**現在の状態から**指定した回転軸を中心に指定した回転角だけ回転する
  - 回転角の単位は度
    - ラジアンで指定したいときは rotateRad() メソッドを使う

rotateDeg() メソッド

- void ofNode::rotateDeg(float degrees, float vx, float vy, float vz)
  - degrees: 回転角（度）
  - vx, vy, vz: 回転軸
- void ofNode::rotateDeg(float degrees, const glm::vec3 &v)
  - degrees: 回転角（度）
  - v: 回転軸（vec3{ vx, vy, vz}）

箱だけが回転する

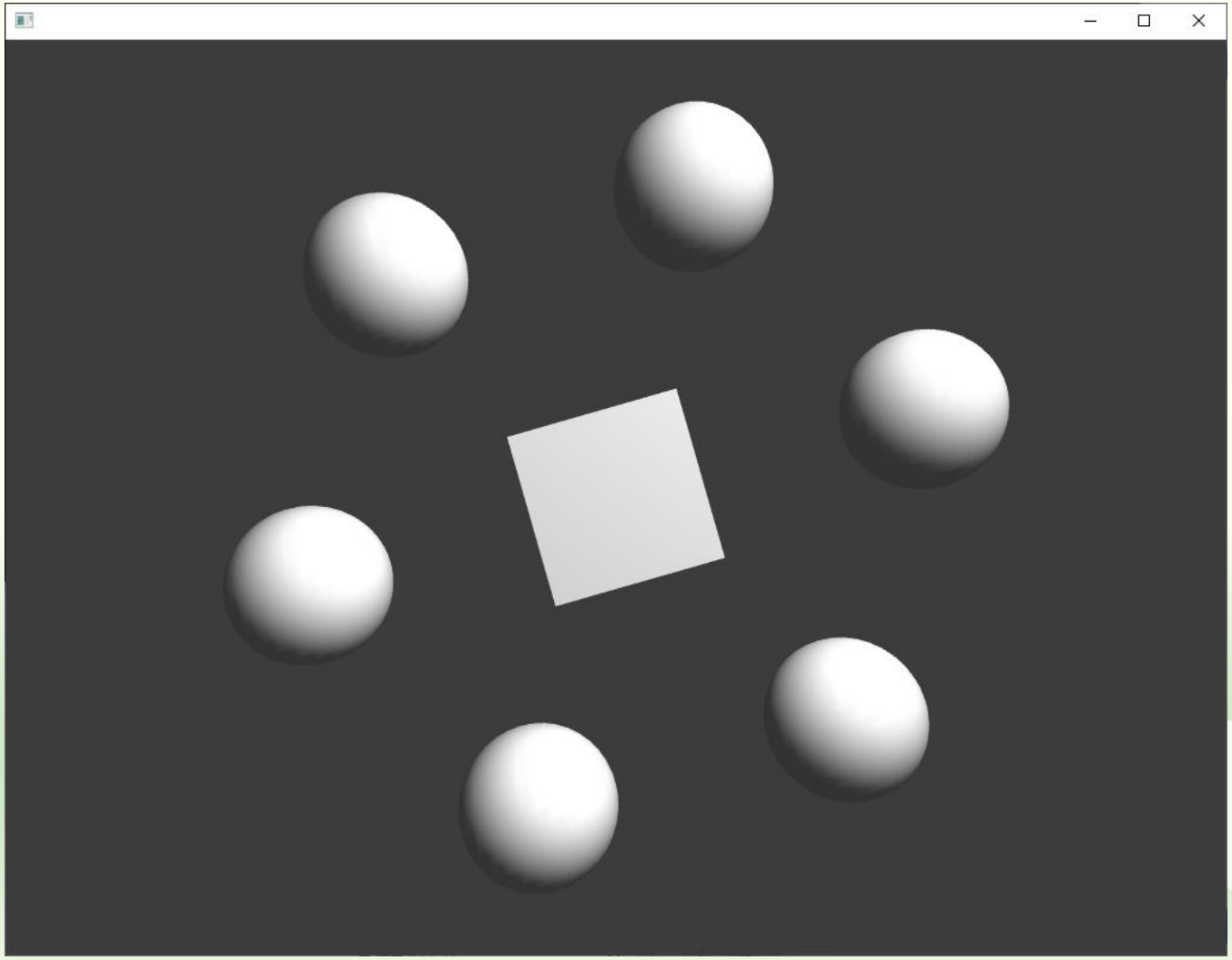


箱をすべての球の親にする

```
using namespace glm;

static ofBoxPrimitive box{ 20.0f, 20.0f, 20.0f };

//--------------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  for (int i = 0; i < 6; ++i){
    spheres.emplace_back(10.0f, 20);
    spheres.back().setPosition(0.0f, 40.0f, 0.0f);
    spheres.back().rotateAroundDeg(60.0f * i,
      vec3{ 0.0f, 0.0f, 1.0f },
      vec3{ 0.0f, 0.0f, 0.0f });
    spheres.back().setParent(box);
  }
}
```

- setParent() メソッドは物体の「親」になる物体を指定する
  - 「子」の物体の位置や姿勢は親の物体を基準にして決定される

全体が回転する



ofApp.h の vector に箱も入れるようにする

```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofCamera camera;
  ofLight light;
  vector<unique_ptr<of3dPrimitive>> parts;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
   (以下略)
```

- parts という vector にオブジェクトの**ポインタ**を保存する
- **unique_ptr**< … > は他の変数と同じポインタを格納しないポインタのコンテナのデータ型
  - この変数を削除すると入っているポインタが指す実体も削除される
- **of3dPrimitive** は ofBoxPrimitive や ofSpherePrimitive の**基底クラス**
  - 基底クラスのポインタのコンテナは**派生クラス**のポインタを格納可

球や箱は動的に生成する

```
using namespace glm;

static ofBoxPrimitive box{ 20.0f, 20.0f, 20.0f };

//--------------------------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  parts.emplace_back(new ofBoxPrimitive{ 20.0f, 20.0f, 20.0f });

  for (int i = 0; i < 6; ++i){
    parts.emplace_back(new ofSpherePrimitive{ 10.0f, 20 });
    parts.back()->setPosition(0.0f, 40.0f, 0.0f);
    parts.back()->rotateAroundDeg(60.0f * i,
      vec3{ 0.0f, 0.0f, 1.0f },
      vec3{ 0.0f, 0.0f, 0.0f });
    parts.back()->setParent(*parts.front());
  }
}
```

削除

parts の最初には箱のオブジェクトのポインタを入れる

箱のポインタが指す実体を取り出す

- **new** はオブジェクトの生成（メモリの確保と初期化）を行ってデータの格納場所（**ポインタ**）を返す**演算子**
  - 得られたポインタを emplace_back することで unique_ptr のスマートポインタが生成されて parts に追加される
- “->” はポインタが指しているオブジェクトのメンバを示す**アロー演算子**

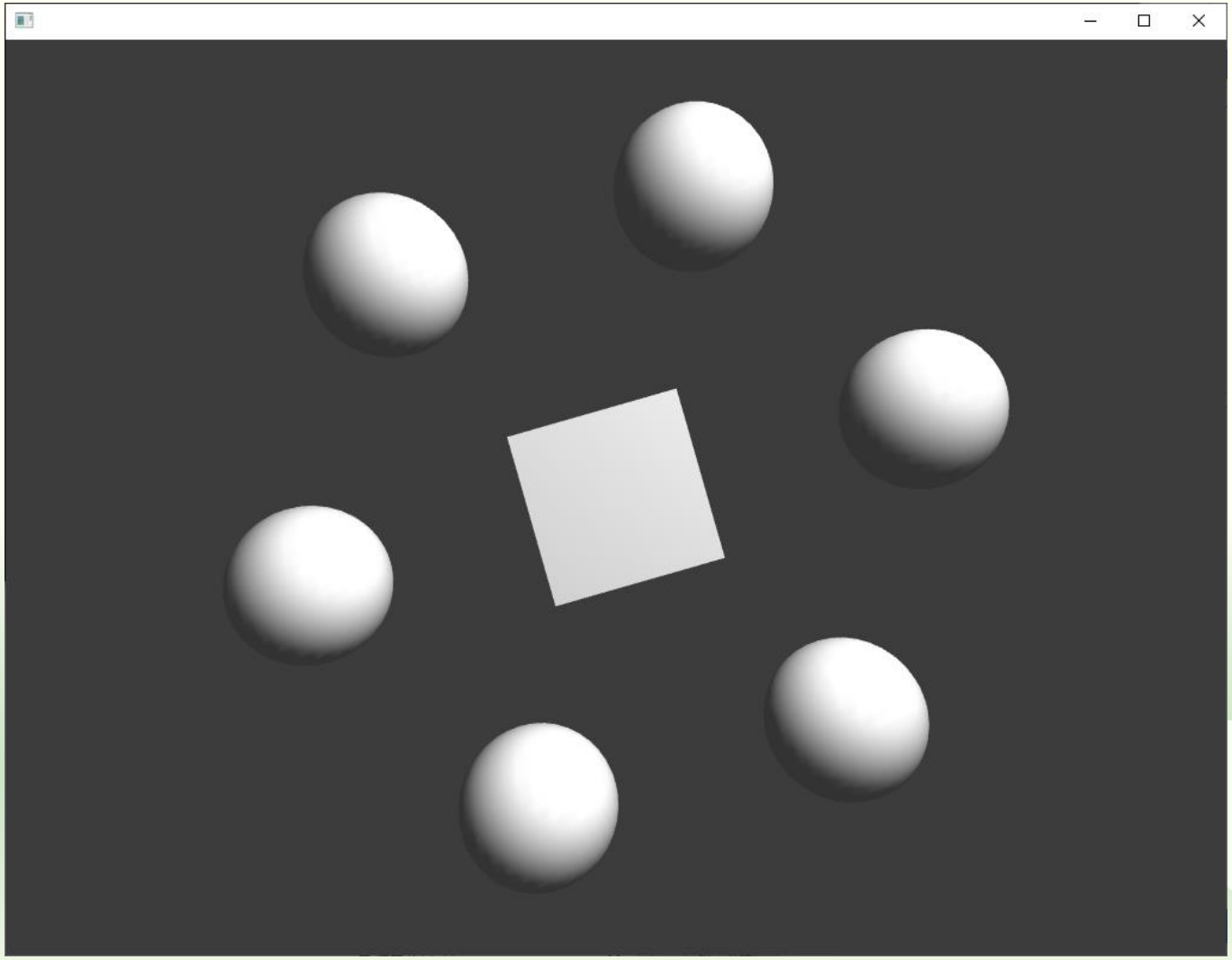
箱も球も parts という vector に入っている

```
//--------------------------------------------------------------
void ofApp::update(){
  parts.front()->rotateDeg(1.0f, 0.0f, 0.0f, 1.0f);
}

//--------------------------------------------------------------
void ofApp::draw(){
  camera.begin();
  for (auto &part : parts){
    part->draw();
  }
  box.draw();
  camera.end();
}
```

削除

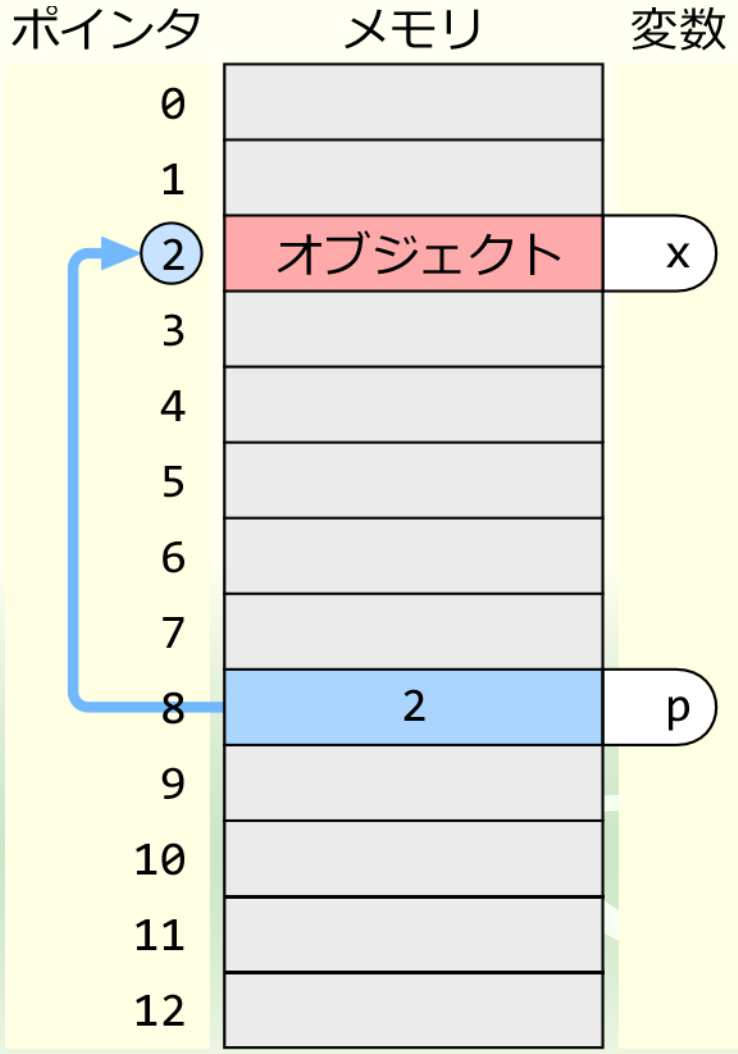
- parts.front() には parts に一番最初に 追加した箱 ofBoxPrimitive のポインタが入っている
- part->draw() で箱も球も混在して描画される

box.draw() がなくても箱が描かれる



オブジェクトとポインタ

- オブジェクト
  - クラスで定義された構造を持つメモリ上のデータ
- 変数
  - データを格納するために確保したメモリの領域に名前（変数名）を付けたもの
- ポインタ
  - データの格納場所を示すデータ



変数 x のメモリの場所②がポインタである

これは参照演算子 & を使って取り出せる

&x → ②

変数 p には変数 x のメモリの場所②がデータとして格納されているとする

p ← &x

変数 p に格納されているポインタ②を使って変数 x に格納されているオブジェクトは関節参照演算子 * を使って取り出せる

*p → オブジェクト

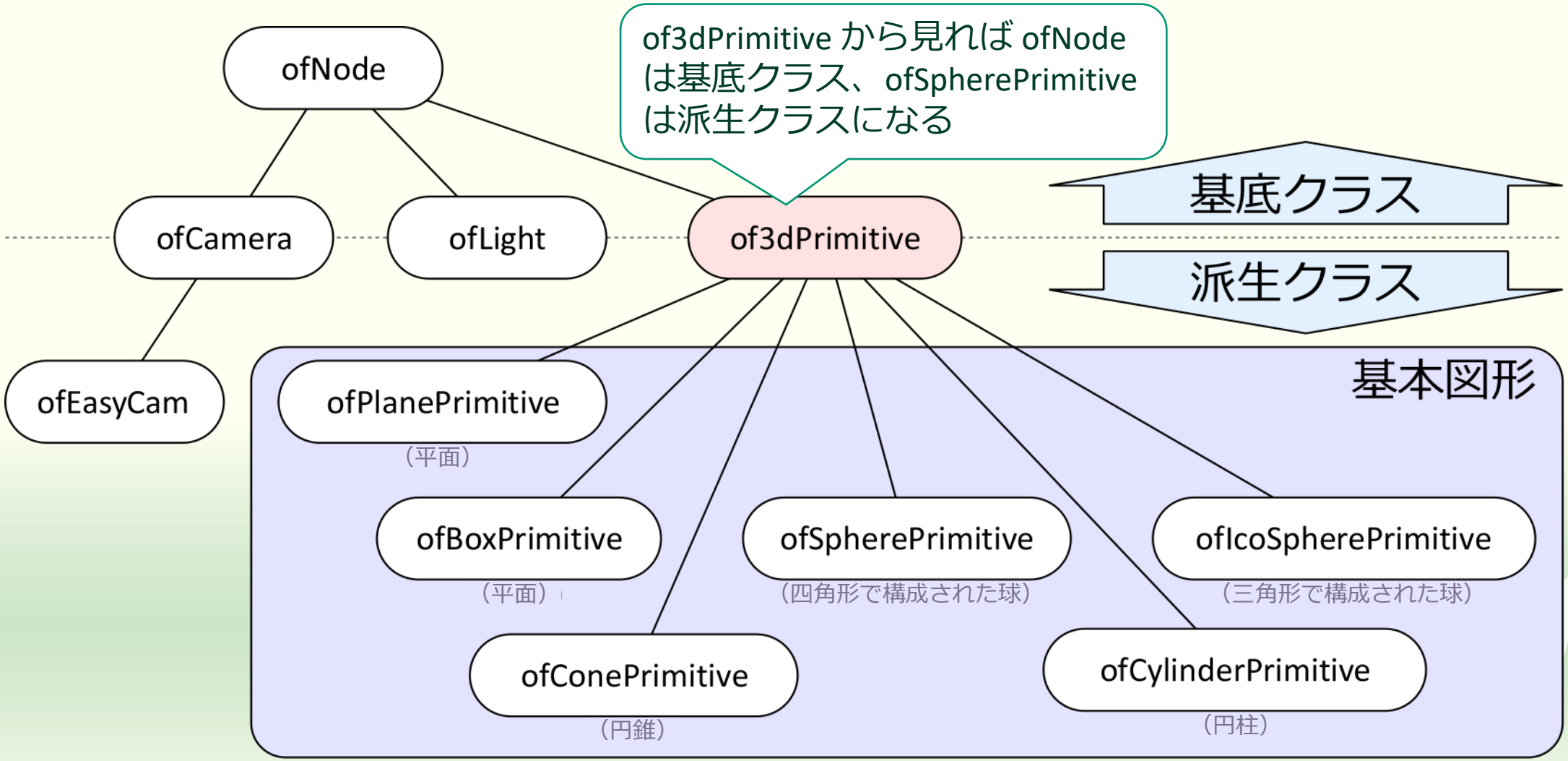
変数と new 演算子

- 変数のオブジェクト
  - 変数宣言によってメモリが確保されそこに生成される
  - 変数のスコープから外れると消去されメモリが解放される
- new 演算子によるオブジェクト
  - new 演算子によりメモリが確保されそこに生成される
  - delete 演算子により消去されメモリが解放される
  - new 演算子はポインタを返すがメモリに**変数名は付かない**
    - ポインタを保存しておかないと使えないし後で削除できない

new 演算子と std::unique_ptr

- new 演算子で生成したオブジェクトは使わなくなったら delete 演算子で**削除**しないといけない
  - delete 演算子により**後始末**も実行される
  - 削除し忘れると使えないメモリが残る（**メモリリーク**）
  - new 演算子が返したポインタを失うと削除できない
- ポインタを失ったらオブジェクトも削除したい
  - ポインタを **std::unique_ptr** 型の変数に入れておけば変数が削除された（スコープを外れた）ときにそのポインタが指しているオブジェクトも自動的に削除される（**スマートポインタ**）

of3dPrimitive



vector<**unique_ptr**<of3dPrimitive>> parts;

- of3dPrimitive クラスのスマートポインタのコンテナの vector
  - parts に格納した of3dPrimitive クラスのポインタが指すオブジェクトは parts が削除されるときに一緒に削除される
- 基底クラスのポインタのコンテナには派生クラスのポインタを格納できる
- parts へのポインタの追加方法の実際
  - parts.emplace_back(new **ofBoxPrimitive**{ 20.0f, 20.0f, 20.0f });
  - parts.emplace_back(new **ofSpherePrimitive**{ 10.0f, 20 });
    - ofBoxPrimitive と ofSpherePrimitive はどちらも of3dPrimitive の**派生クラス**なので of3dPrimitive のポインタのコンテナに格納できる

parts.**back**()->setParent(*parts.**front**());

- parts.back() は最後 parts に追加したデータ（球）のポインタを取り出す
- parts.front() は最初に parts に追加したデータ（箱）のポインタを取り出す
- *（関節参照演算子）はポインタが指しているデータの実体（データを格納しているメモリ）を取り出す
- つまり**箱**のデータを**球の親**に設定している

for (auto &part : parts) { **part->draw**(); }

- part には parts の要素の of3dPrimitive クラスまたはそれを継承したクラスのスマートポインタである
- draw() は of3dPrimitive のメソッドである
- of3dPrimitive クラスを継承している ofSpherePrimitive クラスや ofBoxPrimitive クラスの draw() メソッドは of3dPrimitive クラスのもの

課題３－１

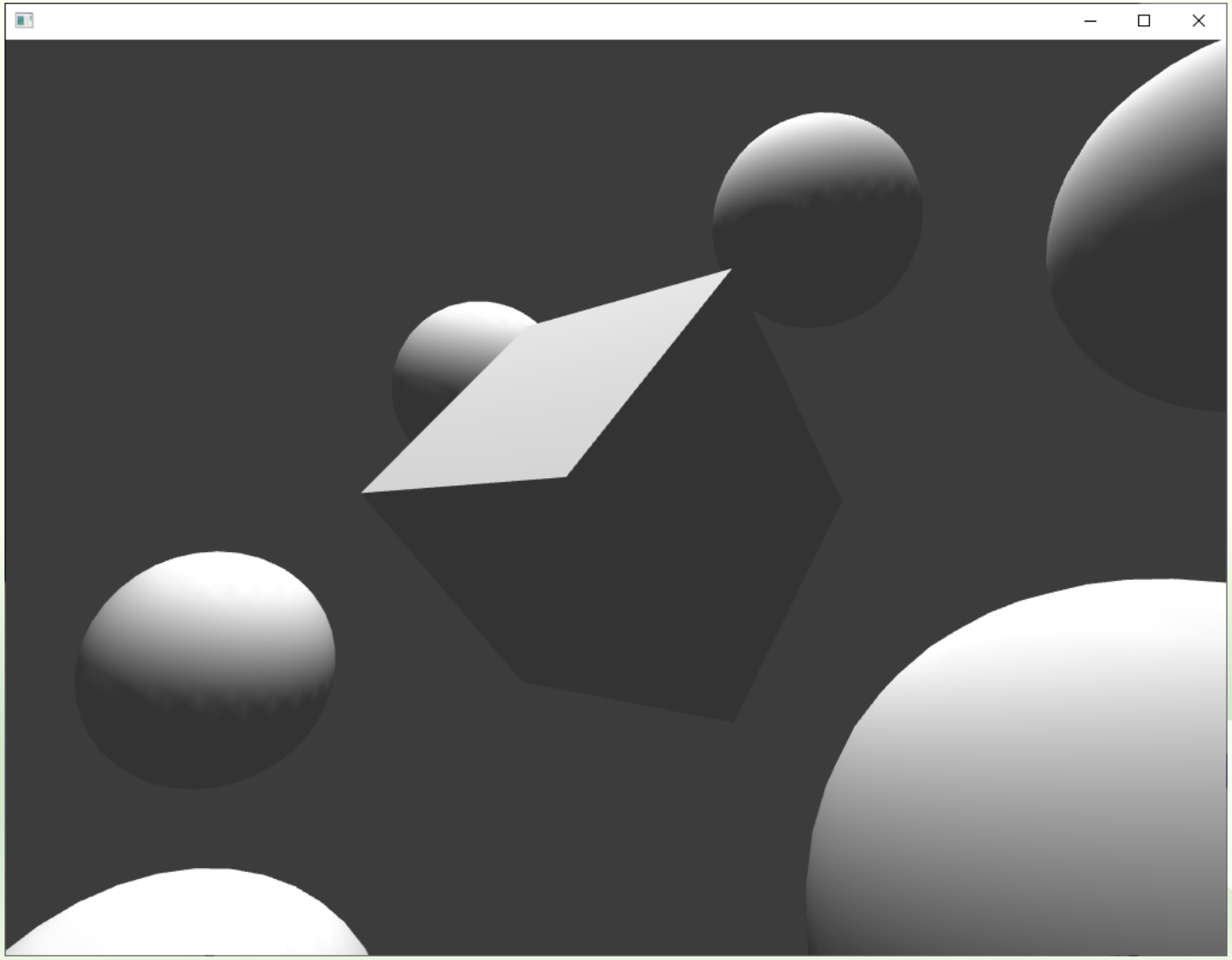
左腕を振る

ofApp.h の ofCamera を ofEasyCam に替える

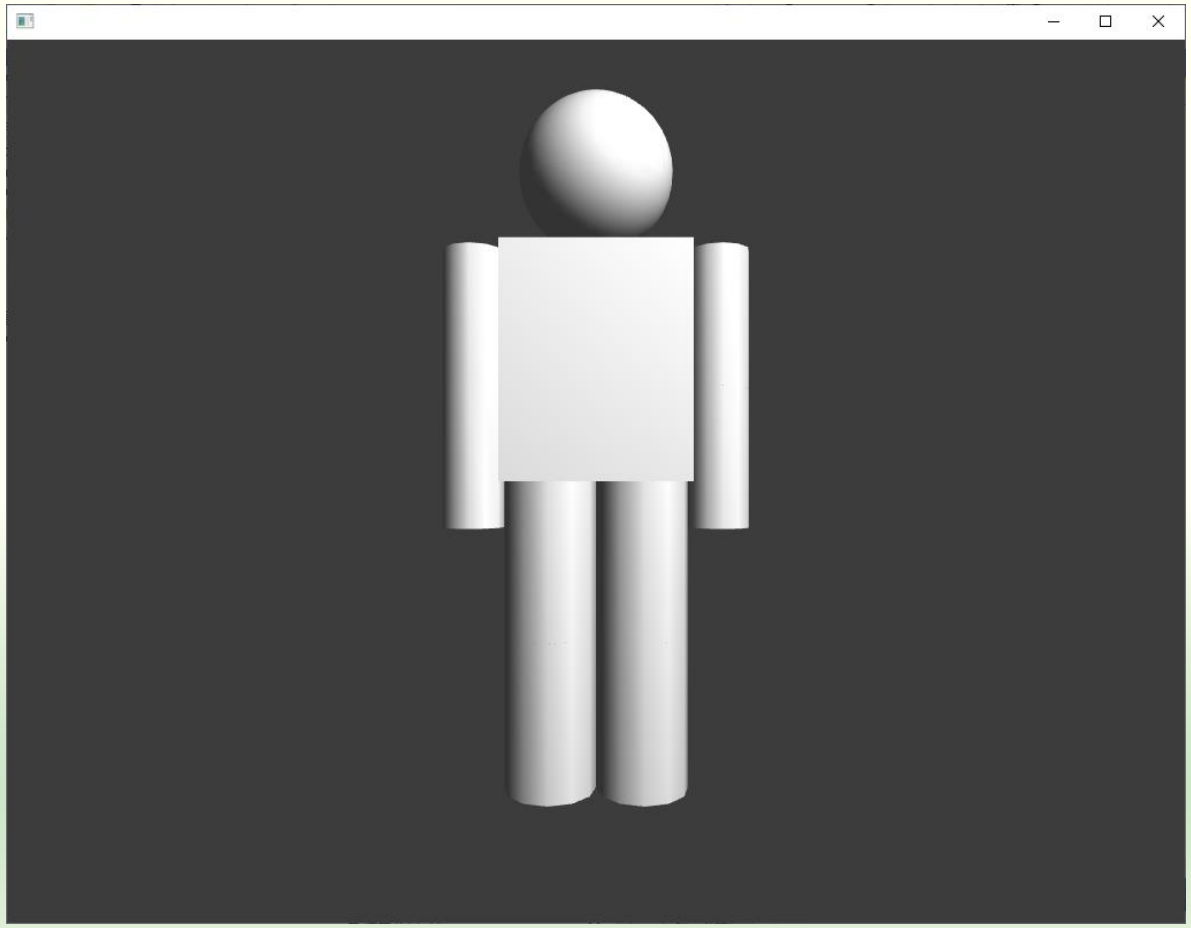
```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofEasyCam camera;
  ofLight light;
  vector<unique_ptr<of3dPrimitive>> parts;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
   (以下略)
```

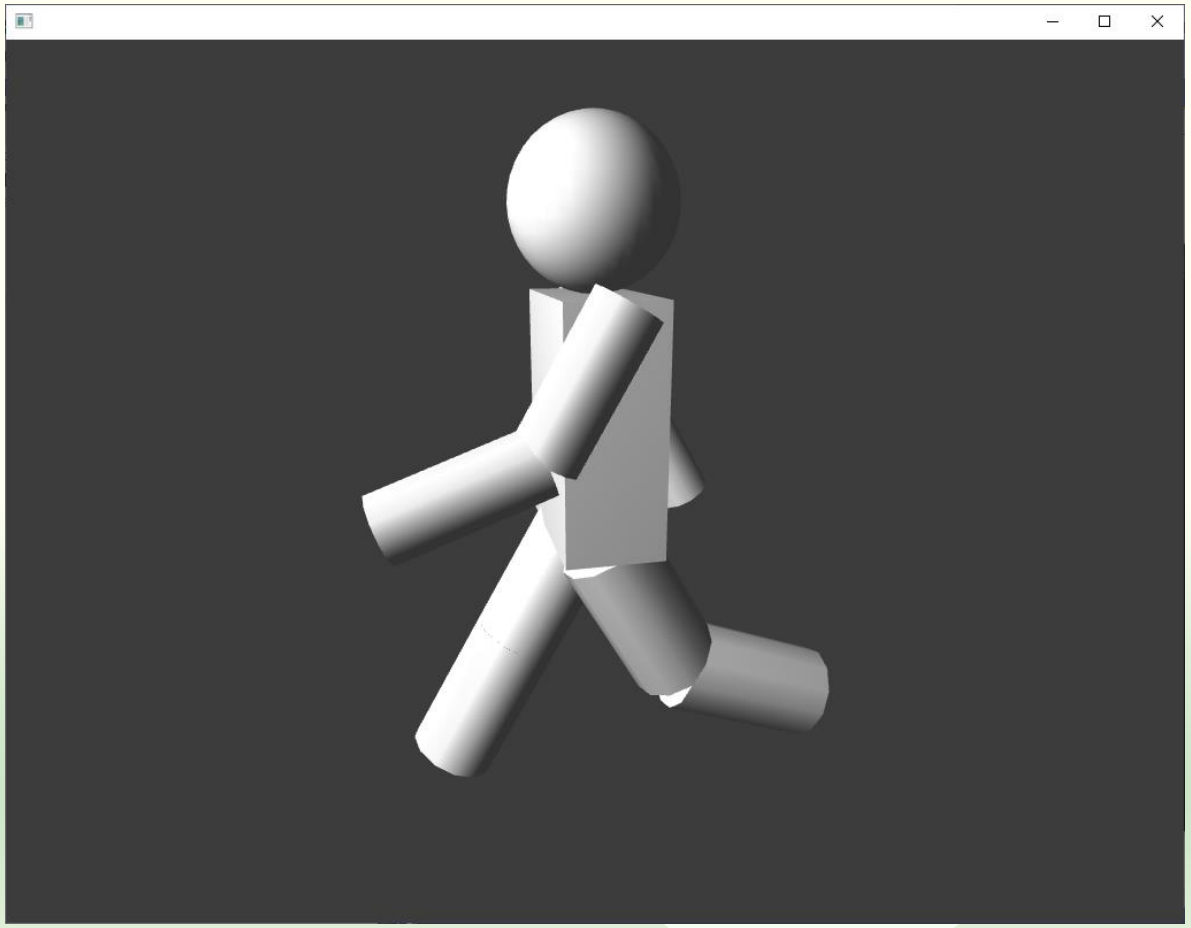
- **ofEasyCam** はマウス操作で視点位置を変更できる
  - 左ドラッグ：回転
  - 中ドラッグ：平行移動
  - 右・ホイール：前後移動
- ofCamera から**派生**したクラス
  - ofCamera を継承している

ofEasyCam でカメラの位置を変更する

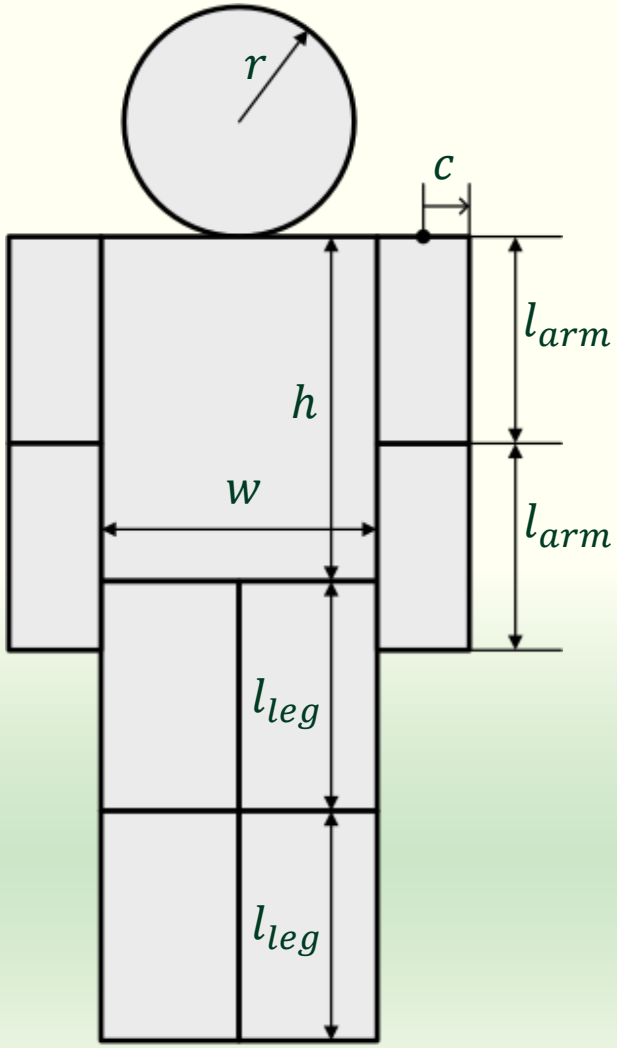


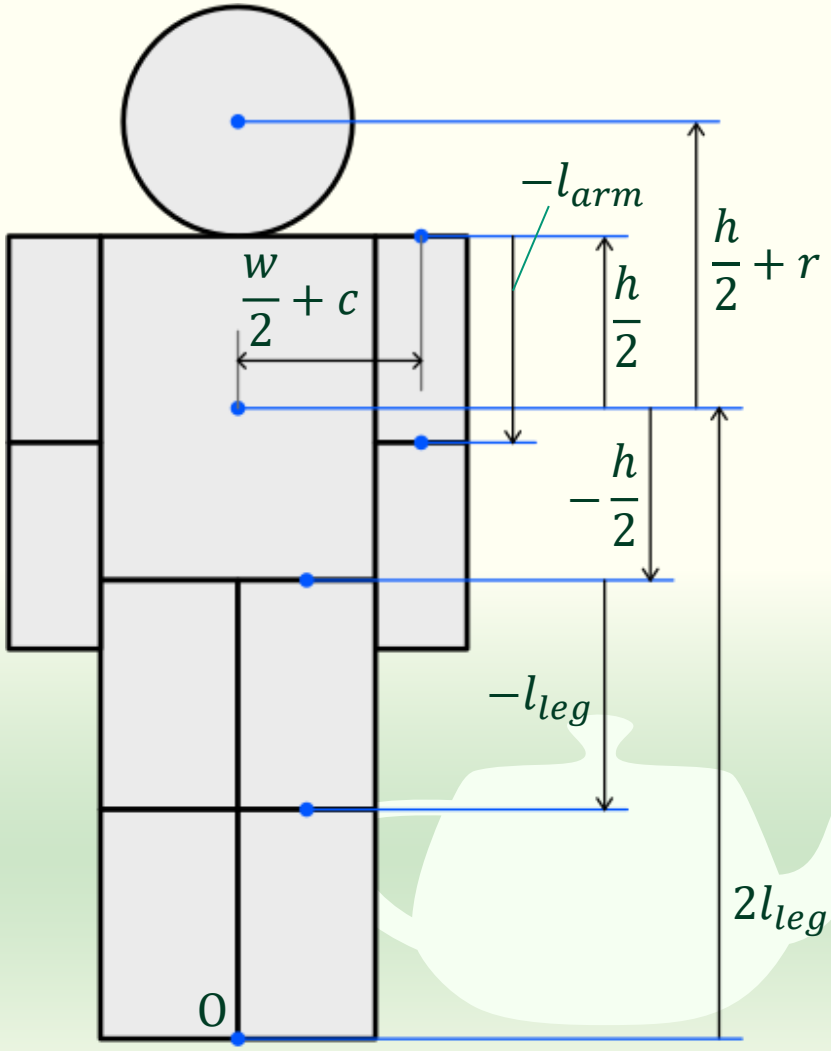
表示する形状をこういう形に作り替える





各パーツのサイズと位置





胴を parts に追加する

```
#include "ofApp.h"

using namespace glm;

//--------------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  // 0: 胴
  auto body = new ofBoxPrimitive{ w, h, d };
  parts.emplace_back(body);
  parts.back()->move(0.0f, 2 l_leg + h / 2, 0.0f);
```

- ofBoxPrimitive のオブジェクトを生成する
  - ポインタを body に入れておく
  - body を parts に追加する
    - 変数 body は setup() が終了すると失われるが parts にいれるので問題ない
  - w, h, d（d はパーツの奥行あるいは厚さ）や l_{leg}（腿・脛の長さ）は適当に決めてください
  - $2\,l_{leg} + h / 2$ は自分で計算してください

頭を parts に追加して胴体を親にする

```
#include "ofApp.h"

using namespace glm;

//--------------------------------------------------
void ofApp::setup(){
  ofEnableDepthTest();
  camera.setPosition(0.0f, 0.0f, 100.0f);
  light.setPosition(60.0f, 80.0f, 100.0f);
  light.enable();

  // 0: 胴
  auto body = new ofBoxPrimitive{ w, h, d };
  parts.emplace_back(body);
  parts.back()->move(0.0f, 2 l_leg + h / 2, 0.0f);

  // 1: 頭
  parts.emplace_back(new
    ofSpherePrimitive{ r, 20 });
  parts.back()->move(0.0f, h / 2 + r, 0.0f);
  parts.back()->setParent(*parts.front());
```

- ofSpherePrimitive のオブジェクトを生成して parts に追加する
  - ポインタを body に入れておく
  - body という変数は setup() が終了すると失われてしまうが parts にいれてしまうので問題ない
  - r を適当に決めて $h / 2 + r$ は自分で計算してください
- parts.front() は parts の先頭の要素の胴体のポインタを取り出す
- setParent() で胴体を親にする

時間の計測の準備

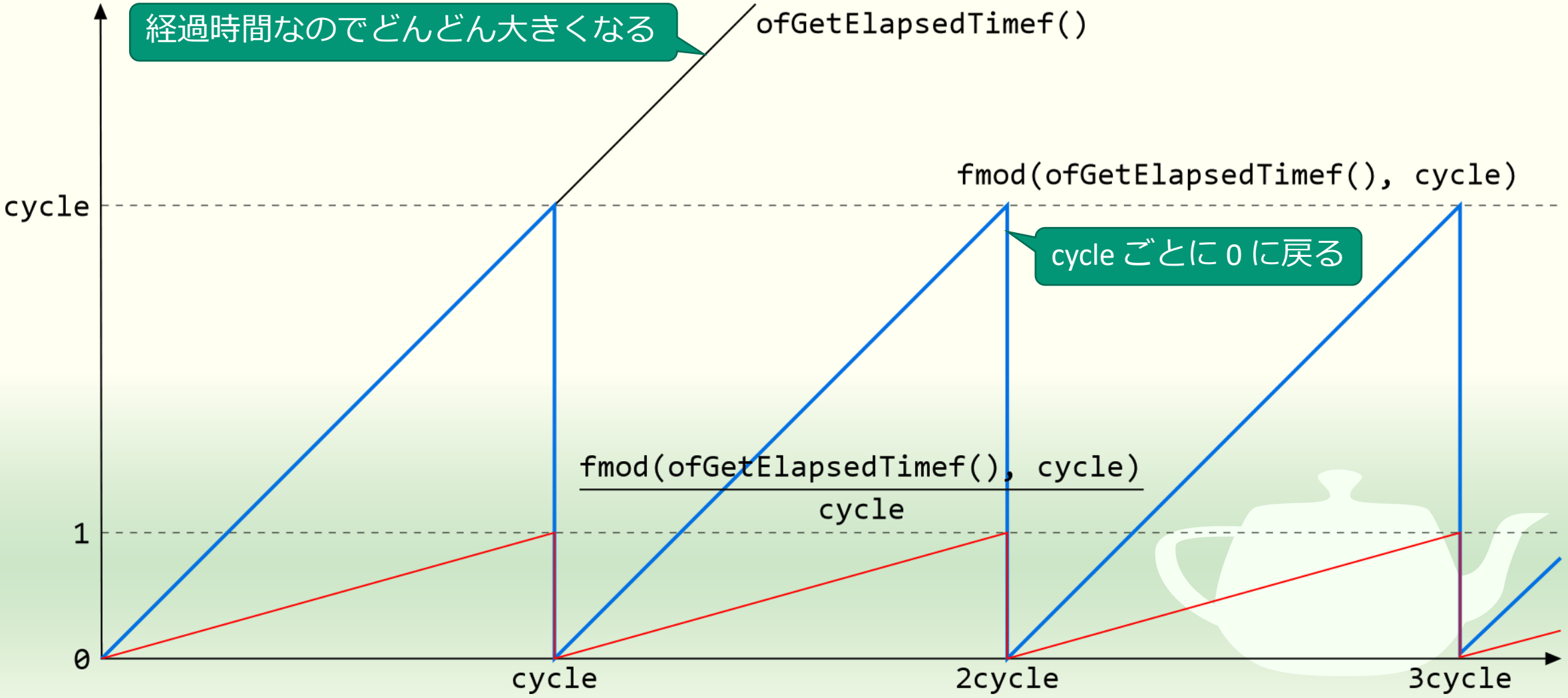
```
#include "ofApp.h"

using namespace glm;

const double cycle{ 2.0 };

//--------------------------------------------
void ofApp::setup(){
   (途中略)
}

//--------------------------------------------
void ofApp::update(){
  const float t{ TWO_PI
    * fmod(ofGetElapsedTimef(), cycle) / cycle };
```

アニメーションの周期

t は 2 秒間で 0 → 2π に変化する

- openFrameworks には円周率 π が **PI**、2π が **TWO_PI** という記号定数で定義されている
  - 4π は **FOUR_PI**、$\pi/2$ は **HALF_PI**
- ofGetElapsedTimef() はプログラム起動時からの経過時間を返す
- fmod(x, y) は実数 x を実数 y で割った剰余を実数で返す
  - さらに y で割ると 0～1 の値になる

fmod(ofGetElapsedTimef(), cycle)



左上腕を追加する

```
//--------------------------------------------------------------
void ofApp::setup(){
（途中略）

  // 0: 胴
  auto body = new ofBoxPrimitive{ w, h, d };
  parts.emplace_back(body);
  parts.back()->move(0.0f, 2 l_leg + h / 2, 0.0f);

  // 1: 頭
  parts.emplace_back(new ofSpherePrimitive{ r, 20 });
  parts.back()->move(0.0f, h / 2 + r, 0.0f);
  parts.back()->setParent(*parts.front());

  // 2: 左上腕
  auto larm = new ofCylinderPrimitive{ c, l_arm, 12, 1 };
  for (auto &vertex : larm->getMeshPtr()->getVertices()) {
    vertex.y -= l_arm / 2;
  }
  parts.emplace_back(larm);
  parts.back()->setParent(*body);
```

- 腕には ofCylinderPrimitive のオブジェクト（円柱）を使う
  - c や l_{arm} は適当に決めてください
  - ofCylinderPrimitive 等 of3dPrimitive を継承したオブジェクトは原点が図形の中心にある
  - 回転中心の原点が円柱の上面に来るように頂点の y 座標値をずらす
    - この処理の説明は割愛
- parts に追加する
- body を親にする

左前腕を追加する

```
// 0: 胴
auto body = new ofBoxPrimitive{ w, h, d };
parts.emplace_back(body);
parts.back()->move(0.0f, 2 l_leg + h / 2, 0.0f);

// 1: 頭
parts.emplace_back(new ofSpherePrimitive{ r, 20 });
parts.back()->move(0.0f, h / 2 + r, 0.0f);
parts.back()->setParent(*parts.front());

// 2: 左上腕
auto larm = new ofCylinderPrimitive{ c, l_arm, 12, 1 };
for (auto &vertex : larm->getMeshPtr()->getVertices()) {
  vertex.y -= l_arm / 2;
}
parts.emplace_back(larm);
parts.back()->setParent(*body);

// 3: 左前腕
parts.emplace_back(new ofCylinderPrimitive{ *larm });
parts.back()->setParent(*larm);
```

データをコピーした
オブジェクトが生成される

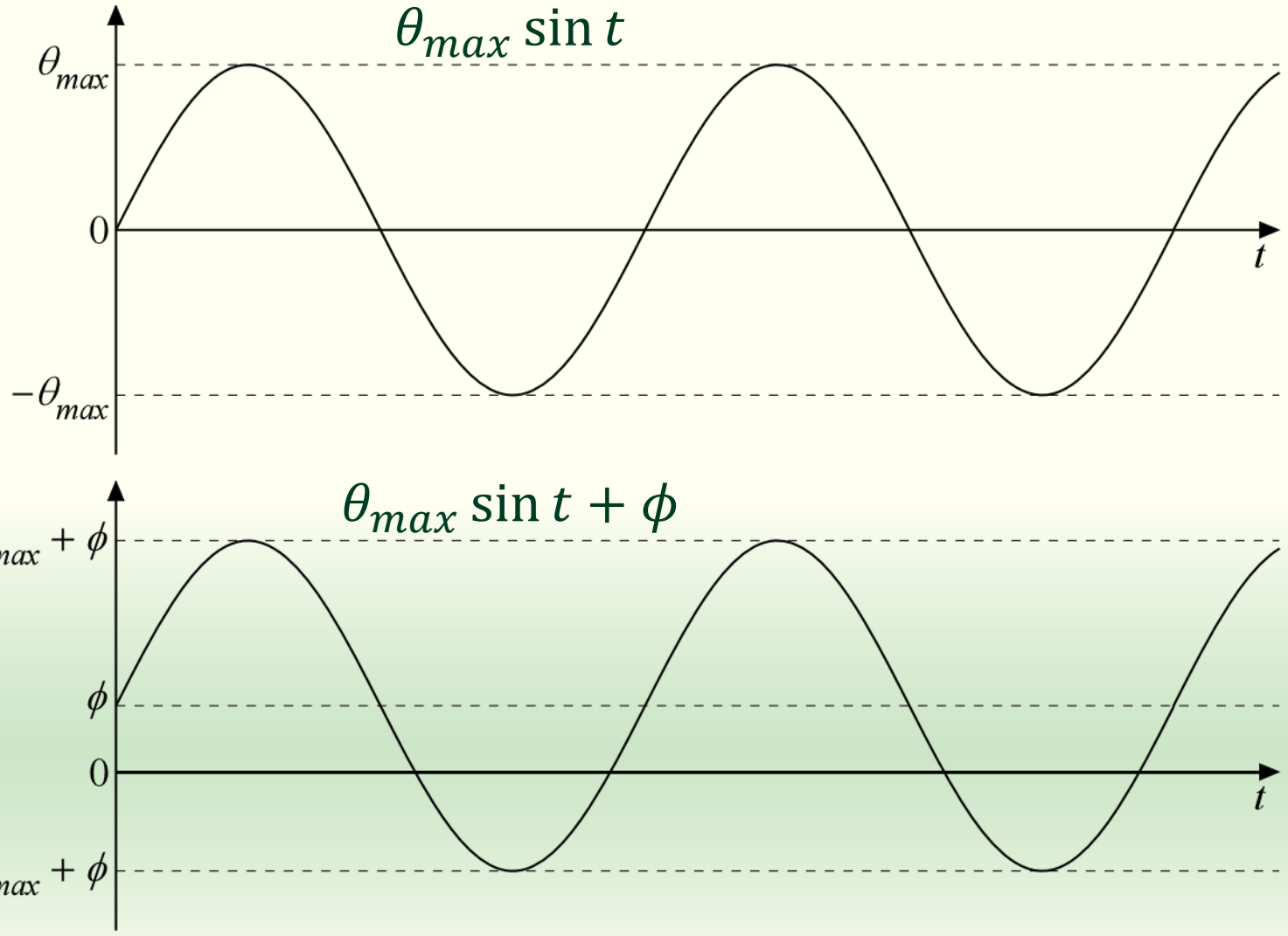
- 左前腕は左上腕をコピーする
  - 左上腕を使って左前腕のオブジェクトを生成する
- 左前腕の親は左上腕にする

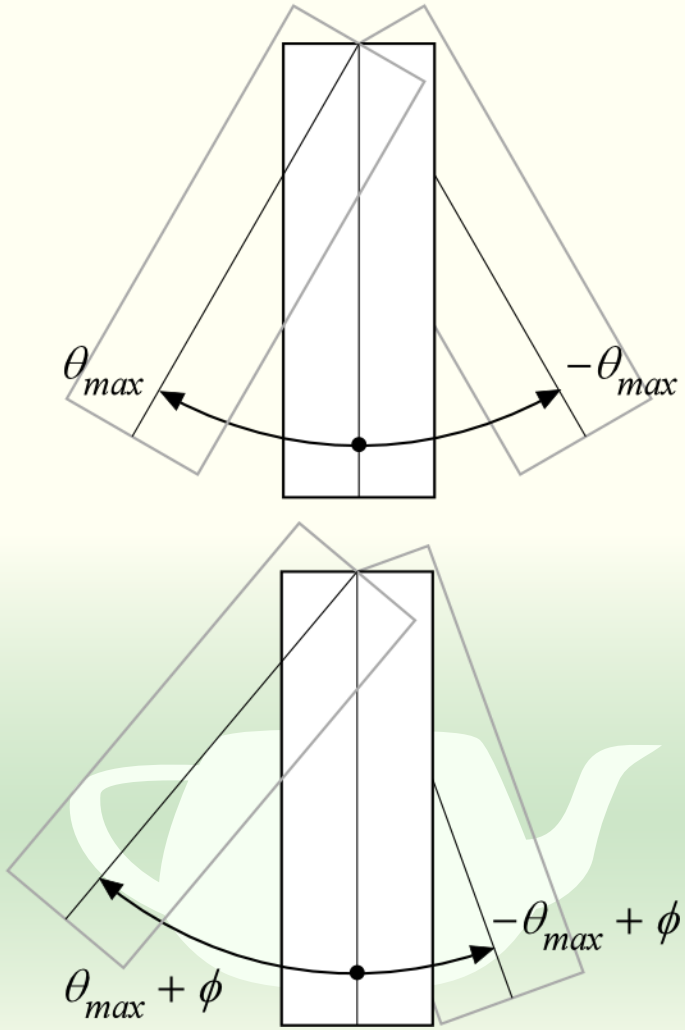
左上腕を配置する

```
const double cycle{ 2.0 };
const vec3 xAxis{ 1.0f, 0.0f, 0.0f };

 (途中略)

//--------------------------------------------
void ofApp::update(){
  const float t{ TWO_PI
    * fmod(ofGetElapsedTimef(), cycle) / cycle };

  const float t2 = θ2max * sin(t);
  parts[2]->resetTransform();
  parts[2]->move(w / 2 + c, h / 2, 0.0f);
  parts[2]->rotateDeg(t2, xAxis);
```

- x 軸方向のベクトルを多用するので定数 xAxis として用意する
- move() メソッドは現在位置からの平行移動なので update() のたびに resetTransform() で元に戻す
- const float t2 = 30.0f * sin(t);
  - 腕を振る動作を再現するために腕の角度を三角関数で変化させる
  - 最適な θ_{2max} を決めてください
  - $w / 2 + c$ や $h / 2$ は自分で計算してください

腕の振れ角を三角関数で変化させる



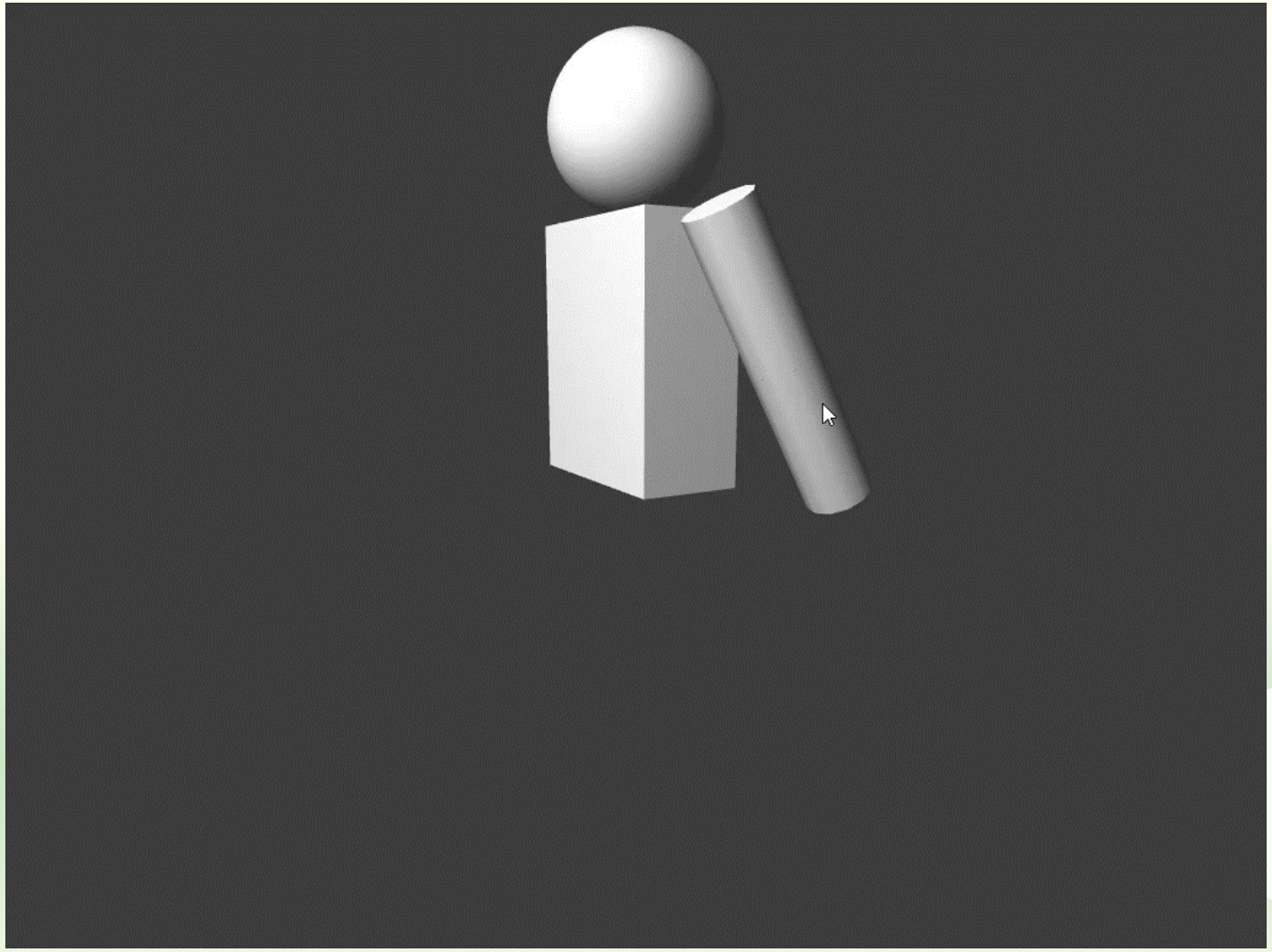


左前腕を配置する

```
//--------------------------------------------
void ofApp::update(){
 const float t{ TWO_PI
    * fmod(ofGetElapsedTimef(), cycle) / cycle };

  const float t2 = θmax * sin(t);
  parts[2]->resetTransform();
  parts[2]->move(w / 2 + c, w / 2, 0.0f);
  parts[2]->rotateDeg(t2, xAxis);

  const float t3 = θ3max * sin(t) - ϕ3;
  parts[3]->resetTransform();
  parts[3]->move(0.0f, –larm, 0.0f);
  parts[3]->rotateDeg(t3, xAxis);
```

- 左前腕は左上腕の振りに追従することに加えて肘の屈伸による振りを追加する
  - θ_{3max} や ϕ_3 を調整して自然な腕の振りを再現してください

課題３－１実行例



課題のアップロード

- 作成したプログラムの実行結果のスクリーンショットを撮って **3-1.png** というファイル名で保存し、Moodle の第 3 回課題にアップロードしてください

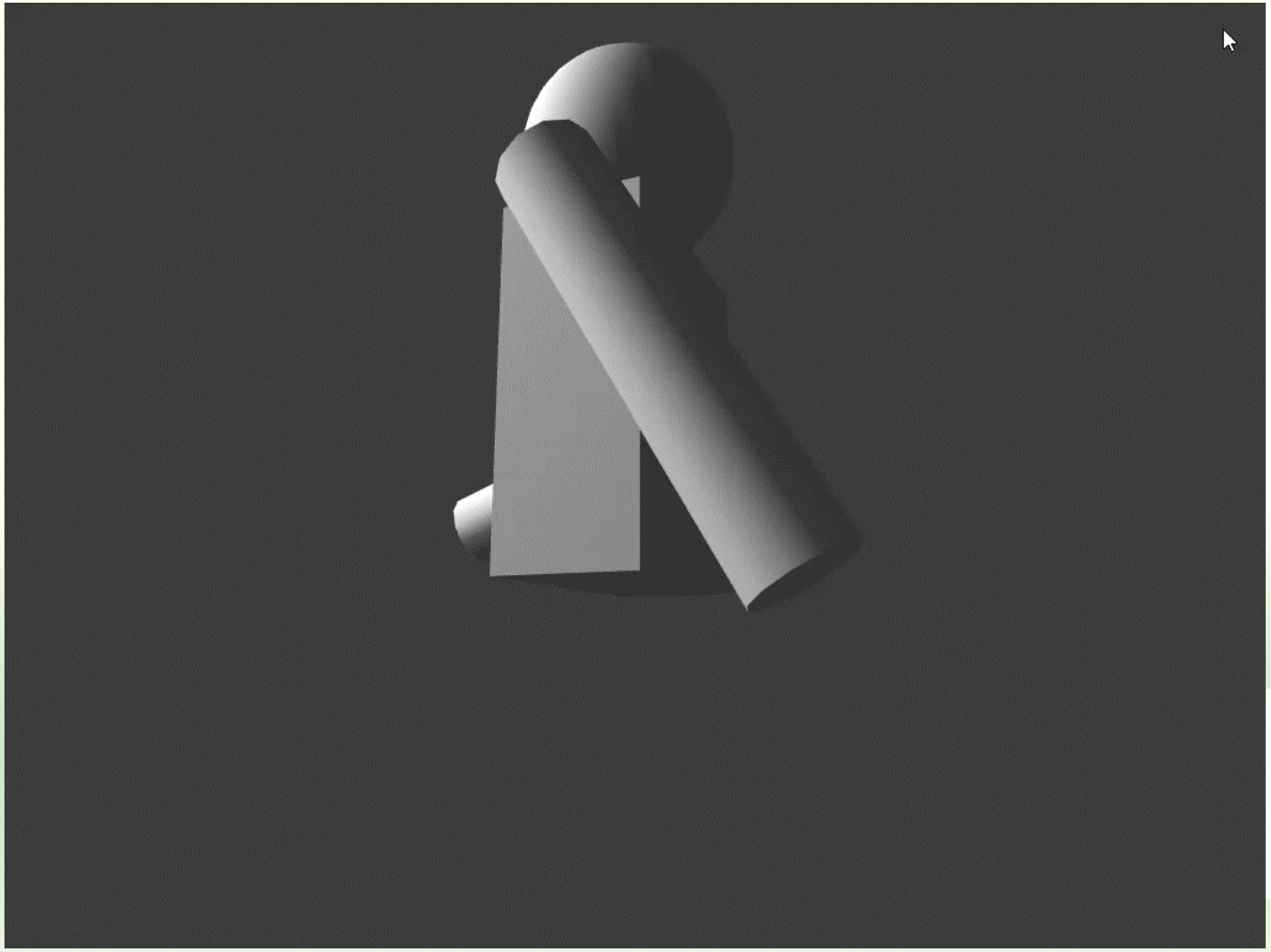
課題３－２

右腕を振る

右腕を振る

- 右腕の上腕と前腕を追加して左腕と反対のタイミングで振るようにしてください
- 右腕の上腕と前腕は左腕と同じ形なので右腕のオブジェクトをコピーして使うとよいでしょう
- このとき右腕の上腕の振りの周期は左腕の上腕と逆になるようにしてください
  - 肘が逆方向に折れ曲がらないように前腕の角度を設定する必要があります

課題３－２実行例



課題のアップロード

- 作成したプログラムの実行結果のスクリーンショットを撮って **3-2.png** というファイル名で保存し、Moodle の第 3 回課題にアップロードしてください

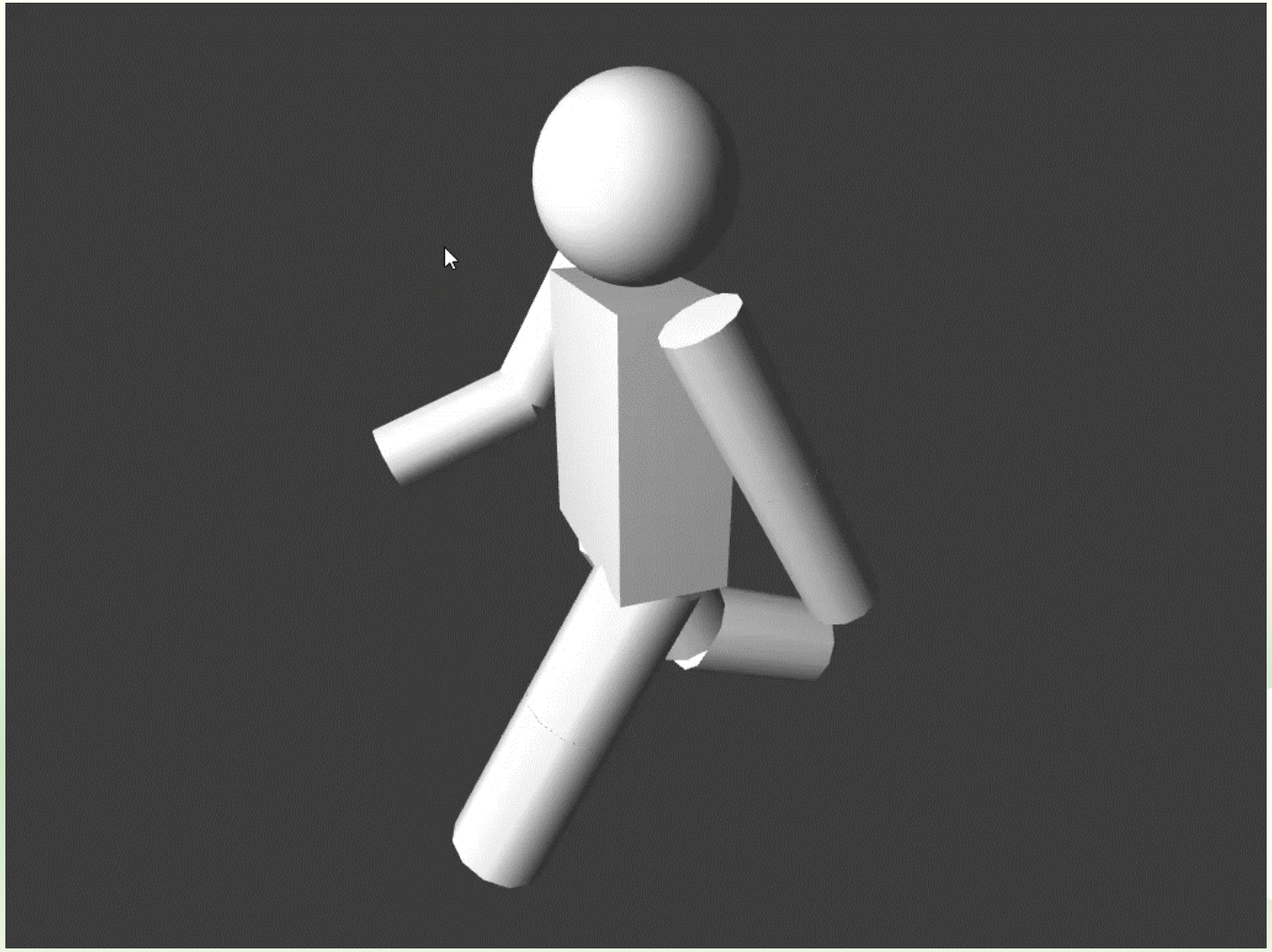
課題３－３

両足を振る

両足を振る

- 右足の腿と脛、左足の腿と脛を追加して、それぞれ右腕、左上と反対のタイミングで振るようにしてください
- 腿と脛は同じ形ですが腕とは形が違うので、腿のオブジェクトを新規に作成して、残りの腿と脛にコピーして使うとよいでしょう
- このとき右足の腿の振りの周期は左足の腿と逆になるようにしてください
  - 膝が逆方向に折れ曲がらないように前腕の角度を設定する必要があります

課題３－３実行例



課題のアップロード

- 作成したプログラムの実行結果のスクリーンショットを撮って **3-3.png** というファイル名で保存し、Moodle の第 3 回課題にアップロードしてください

課題３－４

円上を歩くようにする

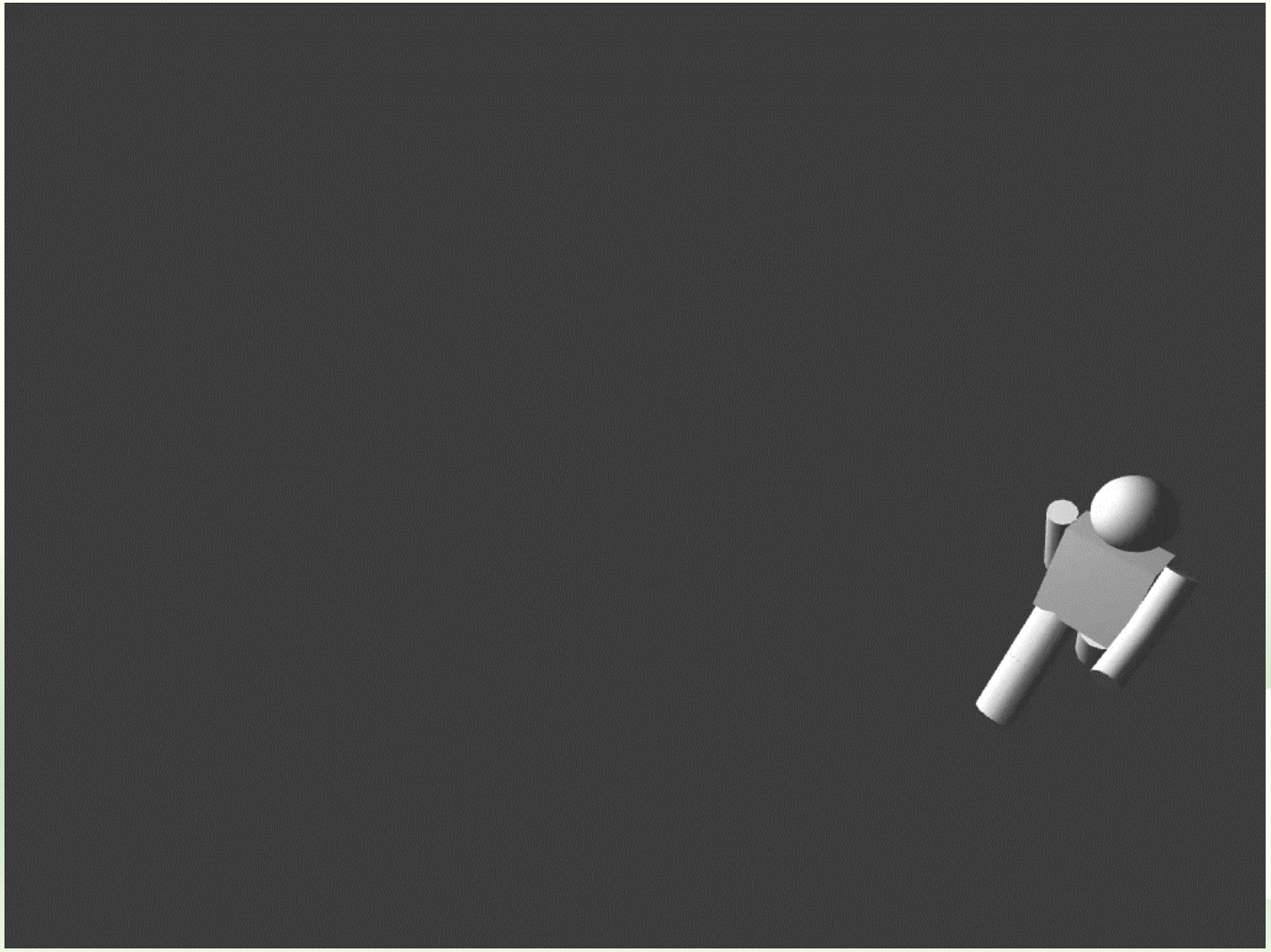
円上を歩く

- 中心を設定して胴のオブジェクトをｙ軸周りに回転させれば円上を歩くようになる

課題のアップロード

- 作成したプログラムの実行中のウィンドウを**５秒以内**で動画キャプチャして、**3-4.mp4** というファイル名で Moodle の第３回課題にアップロードしてください
  - 実行例と同じだと評価できないので形や動きを変えてください
  - 動画のキャプチャができないときはスクリーンショットを撮って 3-4.png というファイル名でアップロードしてください
- ソースプログラム **ofApp.h** と **ofApp.cpp** を Moodle の第３回課題にアップロードしてください

課題３－４実行例



補足

カメラの向き、画角、縦横比、前方面、後方面を設定する

lookAt() メソッド

- void ofNode::lookAt(const glm::vec3 &target)
  - カメラの向きを target の位置に向ける
- void ofNode::lookAt(const glm::vec3 & target, glm::vec3 up)
  - 上方向を up にしてカメラの向きを target の位置に向ける
- void ofNode::lookAt(const ofNode &node)
  - カメラの向きを他の node の位置に向ける
- void ofNode::lookAt(const ofNode &node, const glm::vec3 &up)
  - 上方向を up にしてカメラの向きを他の node の位置に向ける

カメラを移動しても常に１点を向くようにする

```
//--------------------------------------------
void ofApp::update(){
  const vec3 up{ 0.0f, 1.0f, 0.0f };
  const vec3 center{ 0.0f, 0.0f, 0.0f };

  camera.rotateAroundDeg(1.0f, up, center);
  camera.lookAt(center, up);

   (途中略)
}
```

- rotateAroundDeg() メソッドでカメラの位置を回転してもカメラの方向は変わらない
  - rotateAroundDeg() で camera の位置を回転するときカメラは center からずれた位置にないといけない
- lookAt() メソッドでカメラが常に原点を向くようにしている
  - この場合 up を指定しなくても y 方向が上になる

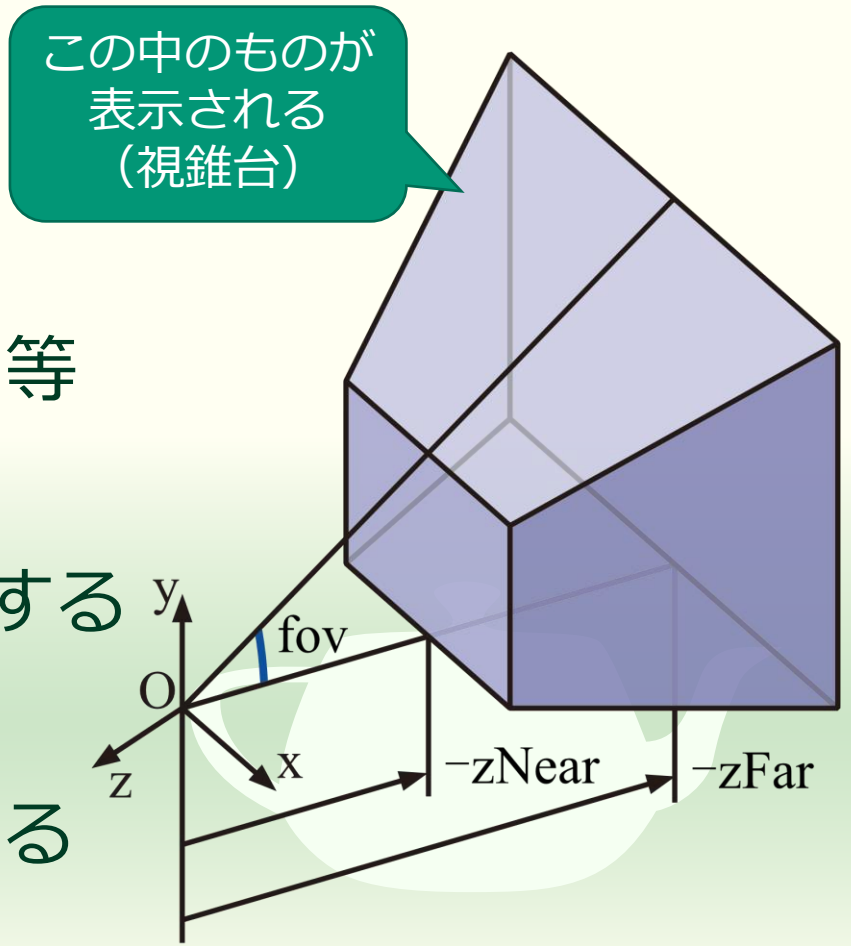
カメラが他の図形を追いかけるようにする

```
//--------------------------------------------
void ofApp::update(){
  const vec3 up{ 0.0f, 1.0f, 0.0f };
  const vec3 center{ 0.0f, 0.0f, 0.0f };

  parts[0]->rotateAroundDeg(-0.3f, up, center);
  parts[0]->rotate(-0.3f, up);

  camera.lookAt(*parts[0]);

   (途中略)
}
```

- lookAt() の目標にはほかの物体 (ofNode) が指定できる
- この camera は常に parts[0] の方向を向く
- lookAt() は ofNode クラスのメソッドなので ofCamera 以外の of3dPrimitive などの向きも制御できる

カメラの画角、縦横比、前方面、後方面

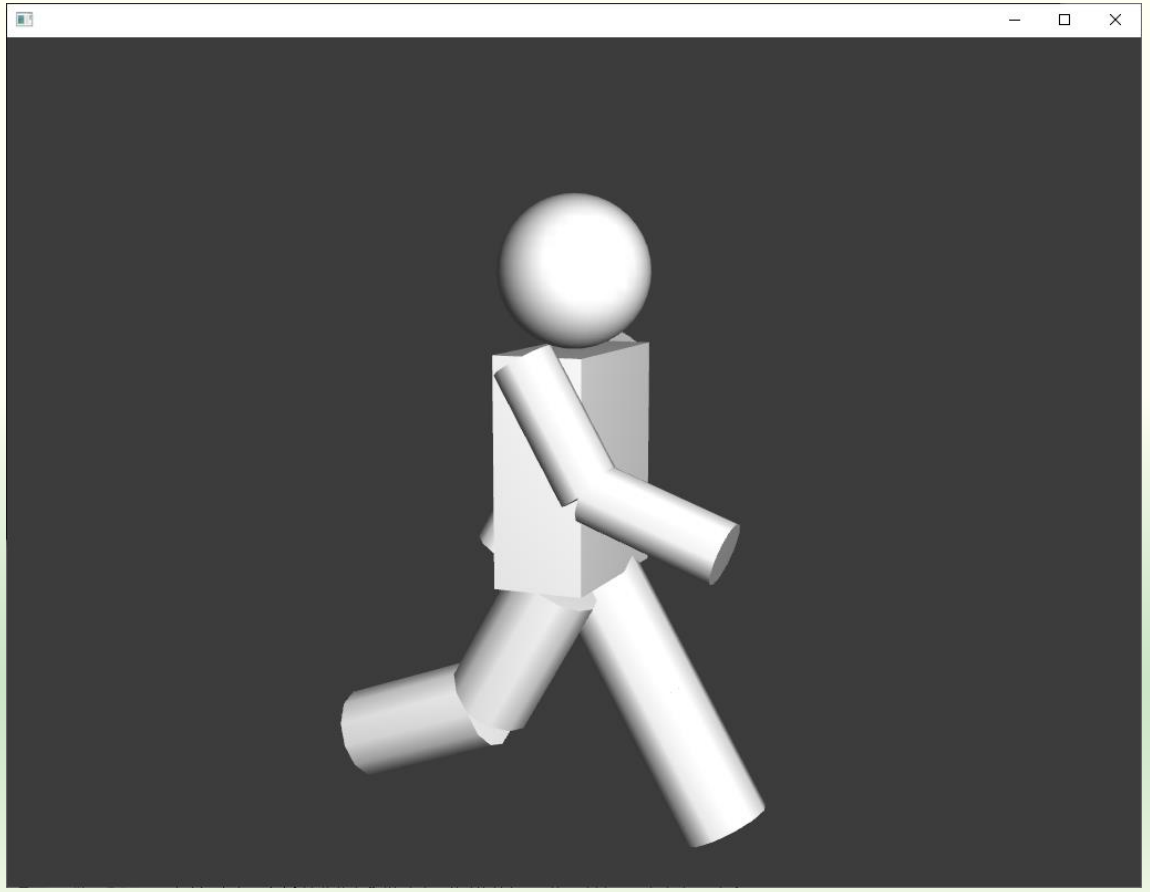
- void ofCamera::setFov(float fov)
  - fov にカメラの画角を度で指定する
- void ofCamera::setAspectRatio(float a)
  - a にウィンドウの縦横比を指定する
  - a は float(ofGetWidth()) / float(ofGetHeight()) 等
- void ofCamera::setNearClip(float zNear)
  - zNear に前方面のカメラからの距離を指定する
- void ofCamera::setFarClip(float zFar)
  - zFar に後方面のカメラからの距離を指定する



画角 fov が小さいほど望遠になる

camera.setFov(60.0f);

- **camera.setFov(30.0f);**

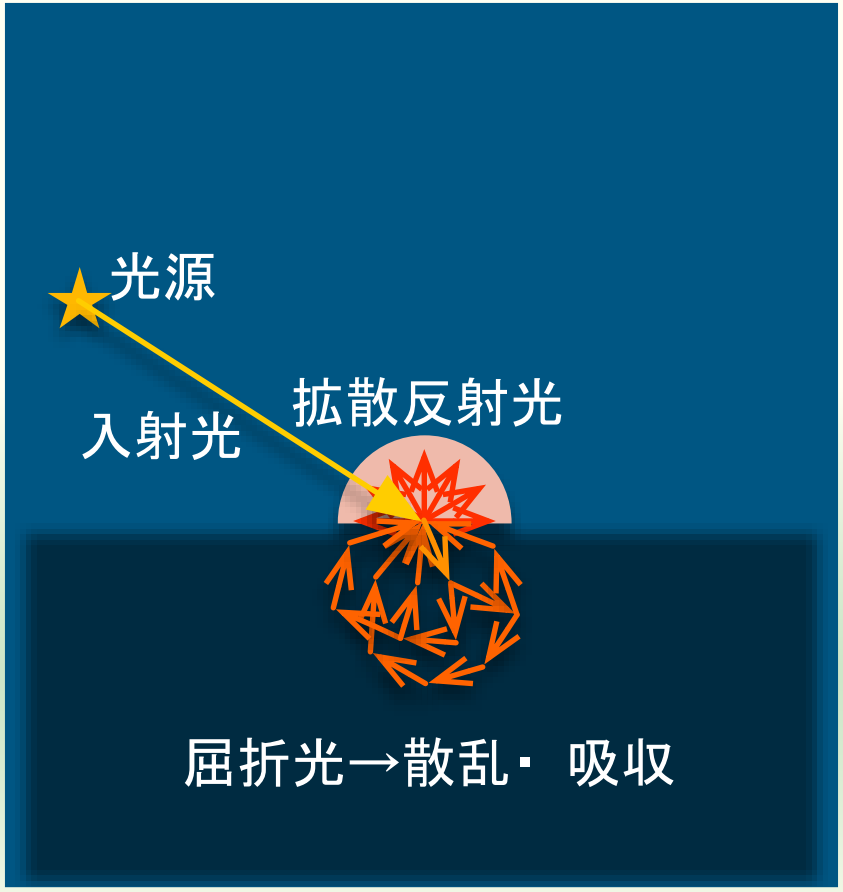


補足

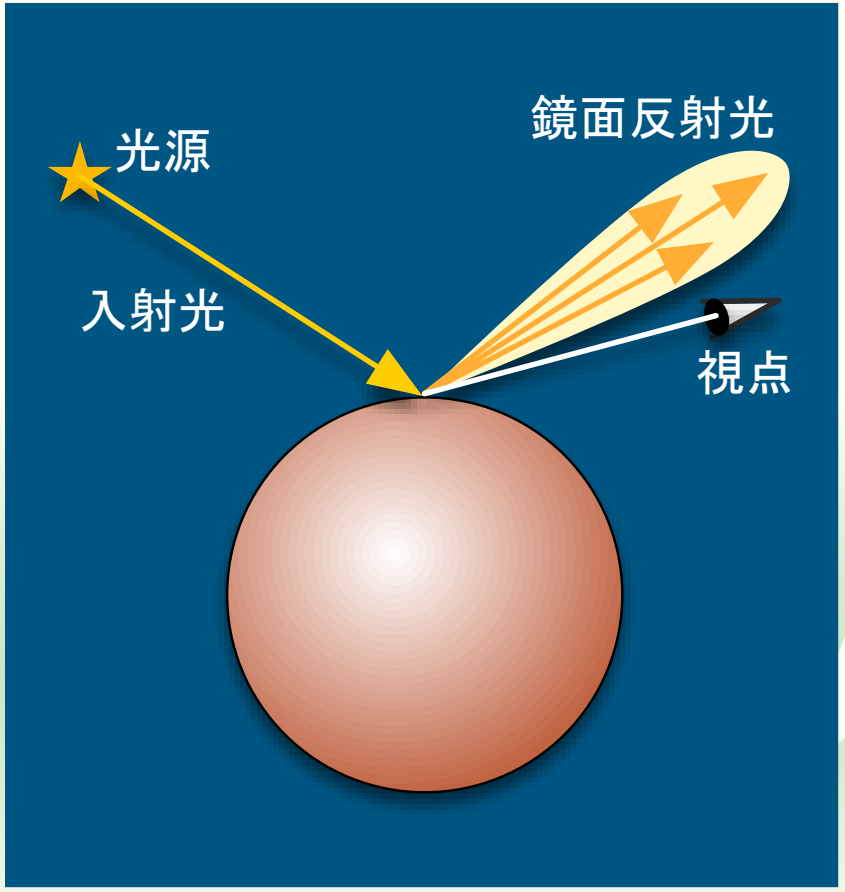
材質

二色性陰影付けモデル

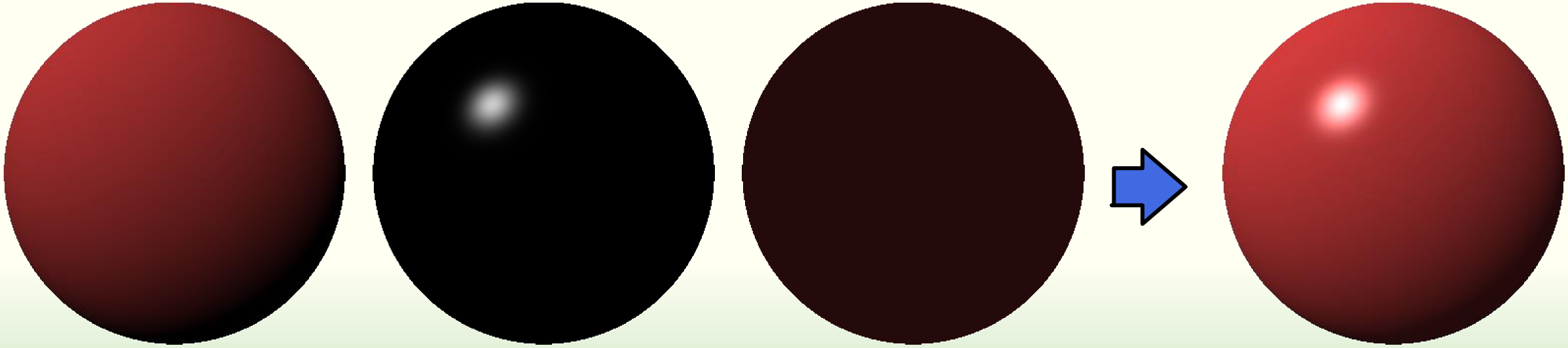
拡散反射光 (diffuse)



鏡面反射光 (specular)



陰影付け方程式



拡散反射光

I_{diff}

鏡面反射光

I_{spec}

環境光の反射光

I_{amb}

反射光強度

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

材質の設定と描画

ofApp.h 等

```
// 材質
ofMaterial material;
```

```
// 拡散反射係数と鏡面反射係数
const ofFloatColor diffuse{ 0.8f, 0.2f, 0.4f };
const ofFloatColor specular{ 0.3f, 0.3f, 0.3f };

// 輝き係数
const float shininess = 30.0f;

// 材質の設定
material.setAmbientColor(diffuse);
material.setDiffuseColor(diffuse);
material.setSpecularColor(specular);
material.setShininess(shininess);
```

エネルギー保存の法則に従うなら上下を足して 1 を超えたらまずい

diffuse と ambient の色は同じにするのが基本

setup() 等

```
// 描画
material.begin();
part->draw();
material.end();
```

draw() 等

- ofMaterial クラスのオブジェクトを生成する
- 拡散反射係数、鏡面反射係数、および輝き係数を決定する
  - 輝き係数が大きいほどハイライトは小さくなる
  - 非金属の材質では鏡面反射係数はグレーにしておくのが基本
  - 金属・半導体では鏡面反射光にも色が付く場合がある
- begin()～end() の間で描画する

補足

std::unique_ptr と std::shared_ptr

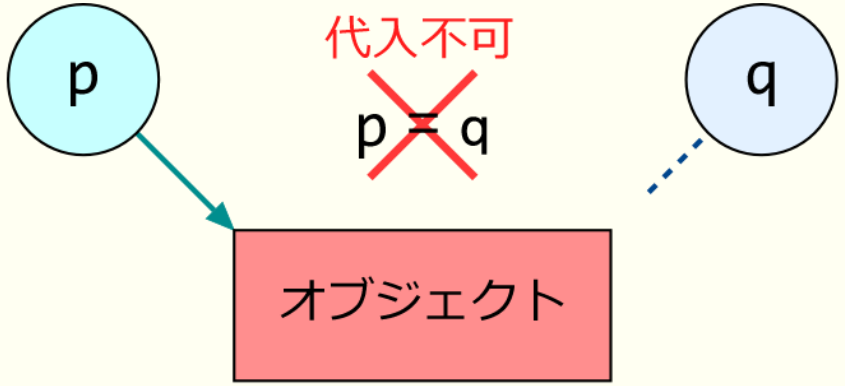
unique_ptr を shared_ptr に替えても動く

```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{
  ofEasyCam camera;
  ofLight light;
  vector<shared_ptr<of3dPrimitive>> parts;

public:
  void setup();
  void update();
  void draw();

  void keyPressed(int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
   (以下略)
```

- unique_ptr は複数の変数で同じポインタを保持できない
  - その分性能は良い
- **shared_ptr** は単一のオブジェクトのポインタを複数の変数で保持できるスマートポインタ
  - オブジェクトはすべての変数が削除されたときに削除される
  - unique_ptr のポインタは複数の変数で保持不可

unique_ptr と shared_ptr

unique_ptr



shared_ptr