

コンピュータグラフィックス 基礎

第4回 GLUTを使ったアプリケーション開発

遠藤結城



学習の目標

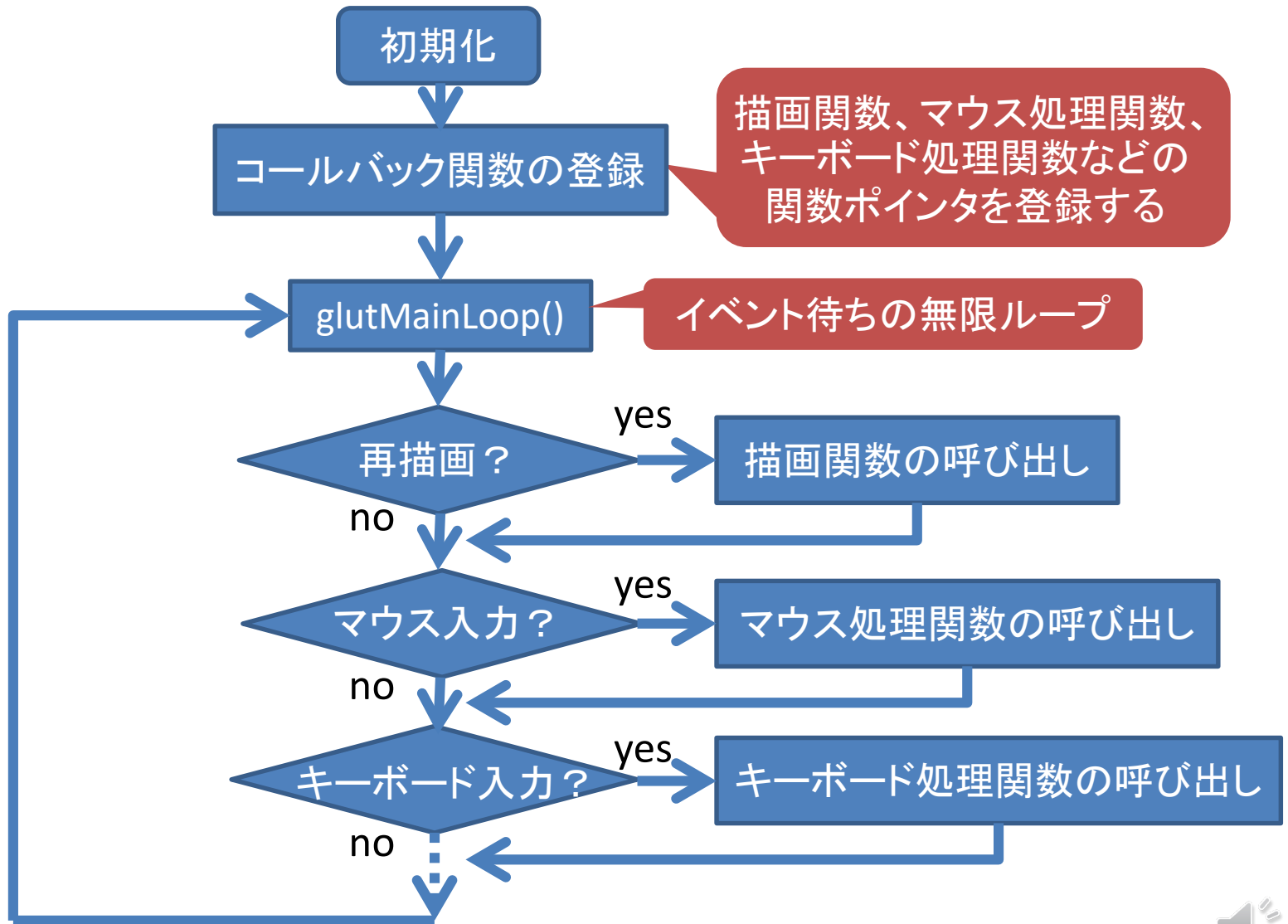
- GLUT のより高度な機能について理解する
- GLUT を用いた対話的なアプリケーションを作成できるようになる
- C++ の初歩的な機能について理解する



キーボードとマウスイベント



GLUT のイベント処理の仕組み



主なコールバック関数の登録関数

- glutDisplayFunc // 描画処理
- glutReshapeFunc // 画面サイズ変更時の処理
- glutKeyboardFunc // キーボード処理
- glutSpecialFunc // 特殊キー押下時の処理
- glutMouseFunc // マウスクリック時の処理
- glutMotionFunc // マウスドラッグ時の処理
- glutTimerFunc // タイマー処理
- その他は <http://opengl.jp/glut/section07.html> 参照



glutKeyboardFunc (キーボード処理)

```
void keyboard(unsigned char key, int x, int y){
    switch (key) {
        case 'A':
        case 'a':
            // キーボードの[A]が押されたときの処理
            break;
        case 'B':
        case 'b':
            // キーボードの[B]が押されたときの処理
            break;
            // 以下同様の case 文
    }
}
```

※ x, yの値はキーが押されたときのマウスカーソルの座標

```
glutKeyboardFunc(keyboard); // 関数ポインタの登録
```



glutSpecialFunc (特殊キー押下時の処理)

```
void special(int key, int x, int y)  
// 中身は keyboard とほぼ同じ
```

```
glutSpecialFunc(special); // 関数ポインタの登録
```

引数 key で与えられる特殊キー

- | | |
|----------------------------------------|--------------|
| • GLUT_KEY_F1 ~ GLUT_KEY_F12 | ファンクションキー |
| • GLUT_KEY_LEFT ~ GLUT_KEY_DOWN | カーソルキー |
| • GLUT_KEY_PAGE_UP, GLUT_KEY_PAGE_DOWN | Page Up/Down |
| • GLUT_KEY_HOME, GLUT_KEY_END | Home, End |
| • GLUT_KEY_INSERT | Insert |

※Esc キー、Backspace キー、Delete キーは keyboard 関数で処理



Shift, Alt, Ctrl キーの押下チェック

```
if(glutGetModifiers() & GLUT_ACTIVE_SHIFT) {  
    // shift キーが押されている  
}
```

```
if(glutGetModifiers() & GLUT_ACTIVE_ALT) {  
    // Alt キーが押されている  
}
```

```
if(glutGetModifiers() & GLUT_ACTIVE_CTRL) {  
    // Ctrl キーが押されている  
}
```



glutMouseFunc (マウスクリック時の処理)

```
void mouse(int button, int state, int x, int y)  
// ※ (x,y) はクリックされた位置  
// 画面の左上が原点、ピクセル数単位
```

```
glutMouseFunc(mouse); // 関数ポインタの登録
```

引数 button の種類

- GLUT_LEFT_BUTTON 左ボタン
- GLUT_MIDDLE_BUTTON 中央ボタン (3 ボタンマウスの場合)
- GLUT_RIGHT_BUTTON 右ボタン

これを区別しないと、ボタンを押して離すまでに同じイベントが2回発生

引数 state の種類

- GLUT_DOWN マウスのボタンが押された
- GLUT_UP マウスのボタンが離された



glutMotionFunc (マウスドラッグ時の処理)

```
void motion(int x, int y)
```

```
// ※ (x,y) はクリックされた位置で、
```

```
// 画面の左上が原点、ピクセル数単位
```

```
glutMotionFunc(motion); // 関数ポインタの登録
```



文字列の表示



文字列の表示

- フォントとサイズの7つの組み合わせから選ぶ

GLUT_BITMAP_8_BY_13

GLUT_BITMAP_9_BY_15

GLUT_BITMAP_TIMES_ROMAN_10

GLUT_BITMAP_TIMES_ROMAN_24

GLUT_BITMAP_HELVETICA_10

GLUT_BITMAP_HELVETICA_12

GLUT_BITMAP_HELVETICA_18

例

```
glColor3f(1,0,0); // 文字列の色の指定 ※照明計算は無効にする必要あり
char str[64];     // 描画したい文字列
sprintf(str, "(%03f, %03f, %03f)", 10.f, 10.f, 10.f);
glRasterPos3f(10.f, 10.f, 10.f); // 3次元の描画開始位置

for (unsigned i = 0; str[i] != '\0'; i++)
    glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[i]);
```

または sprintf_s 関数



透視変換・逆透視変換 とピッキング処理

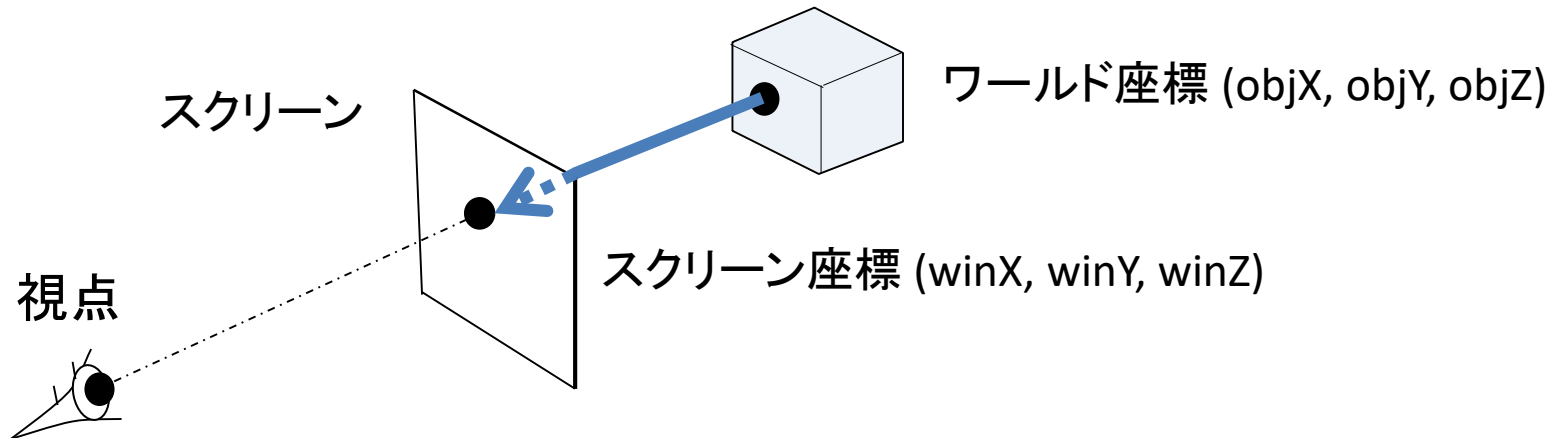


gluProject

ワールド座標をスクリーン座標に変換

OpenGL の座標変換を特定の点に関して計算

```
gluProject(objX, objY, objZ, // ワールド座標  
          model, // モデルビュー行列  
          proj, // 透視投影行列  
          view, // ビューポート  
          winX, winY, winZ) // スクリーン座標
```



gluProject の使用例

```
// ワールド座標 (x,y,z) について計算
```

```
double x = 10, y = 20, z = 30;
```

```
double M[16]; // モデルビュー行列の取得
```

```
glGetDoublev(GL_MODELVIEW_MATRIX, M);
```

```
double P[16]; // 透視投影行列の取得
```

```
glGetDoublev(GL_PROJECTION_MATRIX, P);
```

```
int v[4]; // ビューポートの情報を取得
```

```
glGetIntegerv(GL_VIEWPORT, v);
```

```
double winX, winY, winZ; // スクリーン座標を計算
```

```
gluProject(x, y, z, M, P, v, &winX, &winY, &winZ);
```

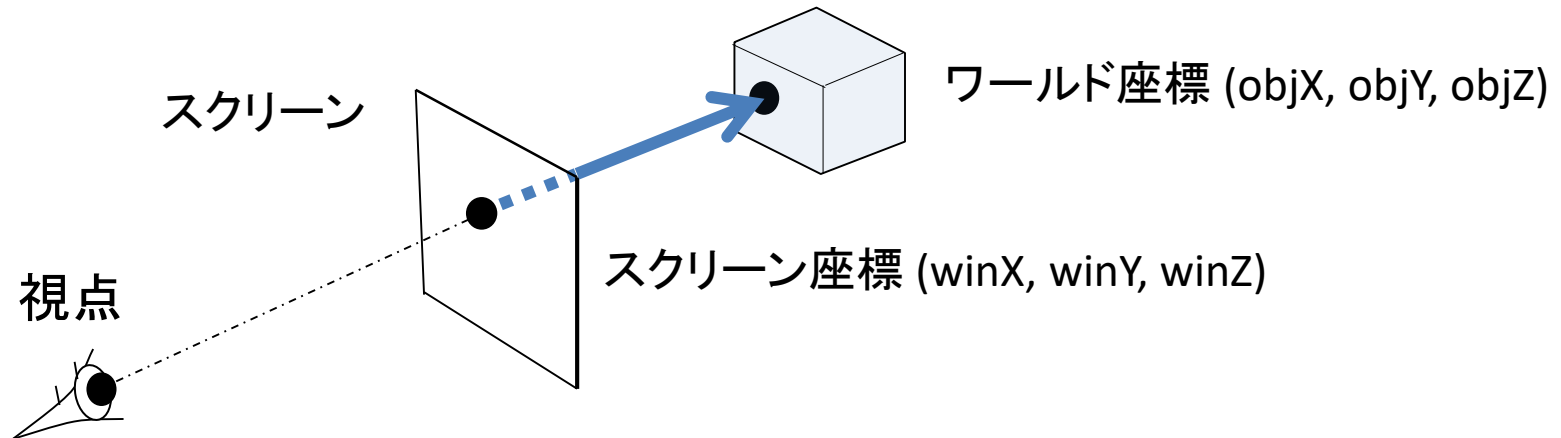


gluUnProject

スクリーン座標をワールド座標に変換

マウスでクリックした点の3次元位置を計算できる

```
gluUnProject(winX, winY, winZ, // スクリーン座標  
            model, // モデルビュー行列  
            proj, // 透視投影行列  
            view, // ビューポート  
            objX, objY, objZ) // ワールド座標
```



gluUnProject の使用例

```
double M[16]; // モデルビュー行列の取得
glGetDoublev(GL_MODELVIEW_MATRIX, M);
double P[16]; // 透視投影行列の取得
glGetDoublev(GL_PROJECTION_MATRIX, P);
int v[4]; // ビューポートの情報を取得
glGetIntegerv(GL_VIEWPORT, v);

int x = 30, y = 30; // スクリーン座標 (x, y) について計算

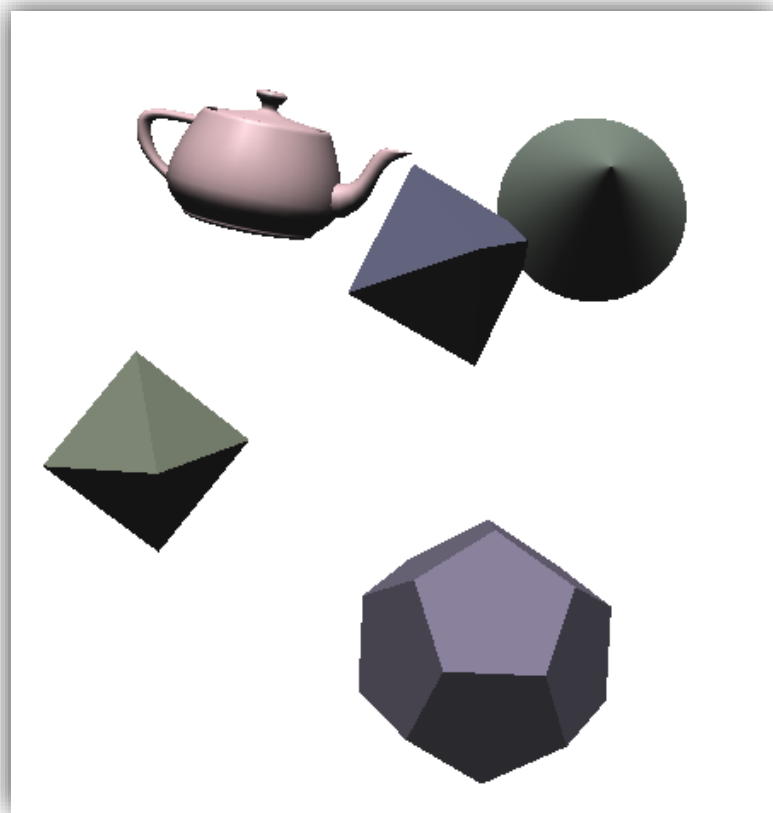
float z; // (x, y) の奥行き値 (デプス) を取得
glReadPixels(x, y, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &z);

double objX, objY, objZ; // ワールド座標を計算
gluUnProject(x, y, z, M, P, v, &objX, &objY, &objZ);

// ※初期化の際に GLUT_DEPTH を指定する必要あり
// glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

マウスピッキング

- 画面上の物体をマウスクリックで選択したい



- やり方は何通りかある



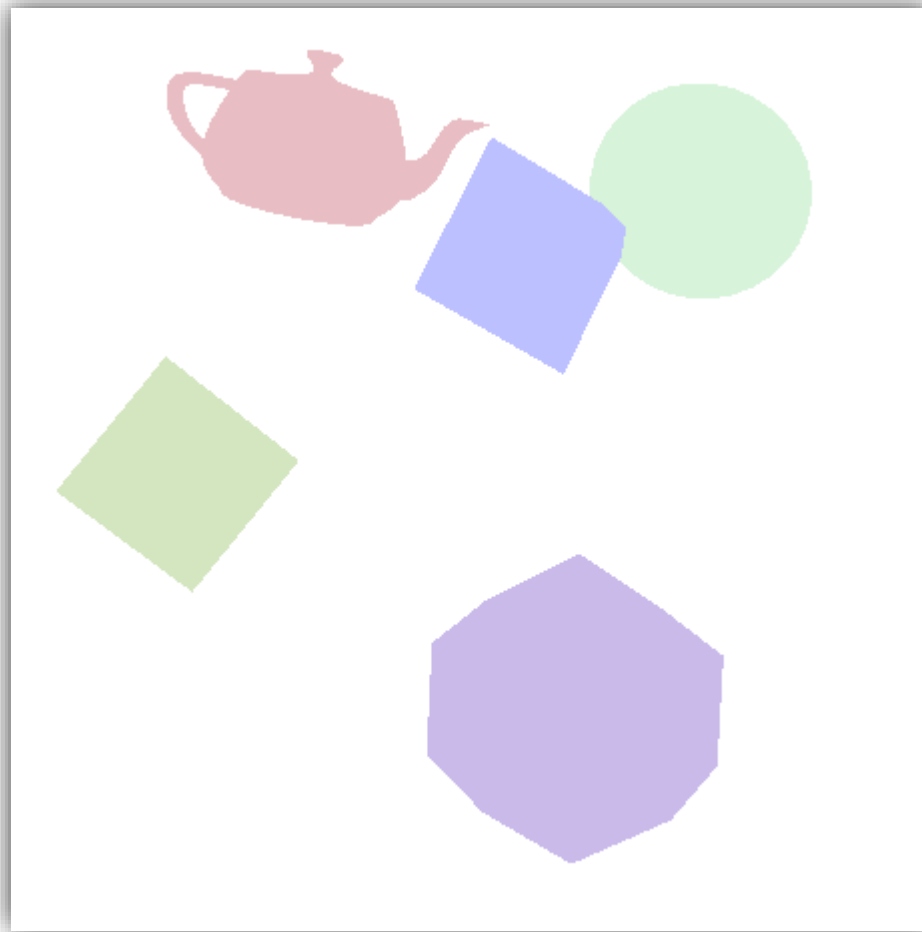
マウスピッキングの実現方法

1. gluProject で物体上の点をスクリーン座標に変換し
クリックした点と最も近い点を持つ物体を選ぶ
2. gluUnProject でクリックした点のワールド座標を
計算し、その座標と最も近い点を持つ物体を選ぶ
物体上の点の座標が必要 (GLUT の立体形状では不可)
物体の数や、各物体上の点の数が多いと時間がかかる
3. OpenGL のセレクションモードを使う
機構が複雑
4. 各物体に ID 色を割り当てて描画



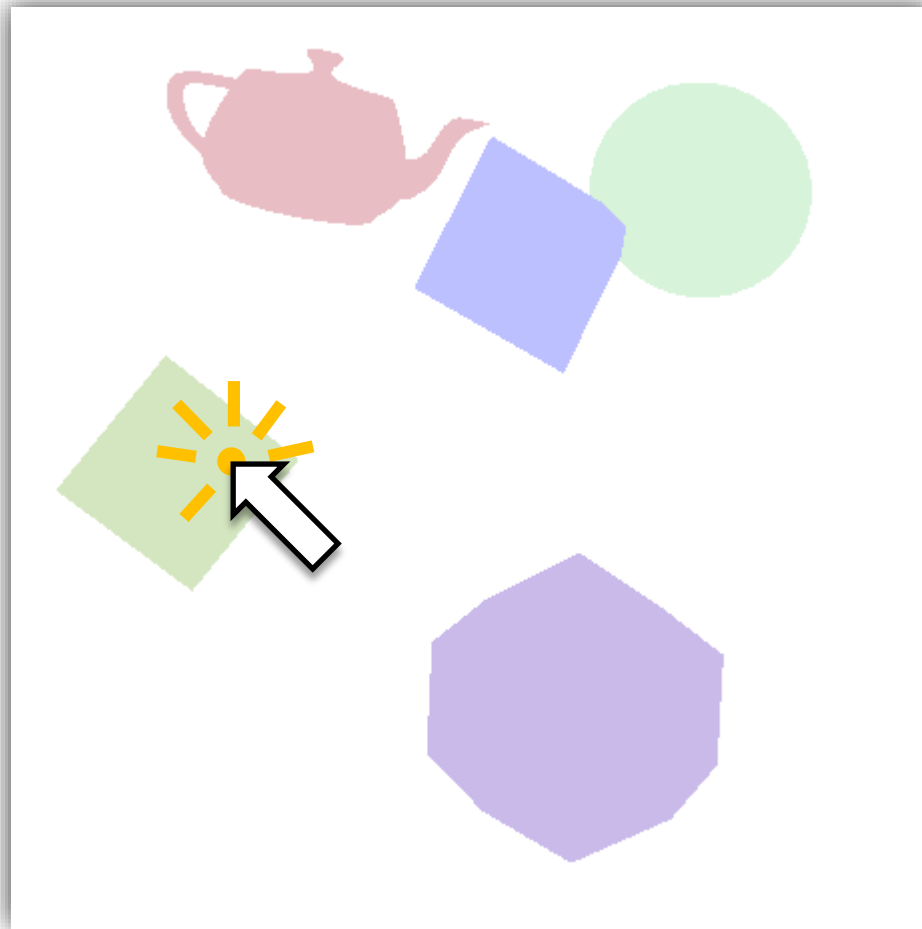
ID 色によるマウスピッキングの実現

① 物体ごとに異なる色で描画する



ID 色によるマウスピッキングの実現

② クリックした点の色を調べてどの物体か判定



ID 色によるマウスピッキングの実現

```
// モデルビュー行列と透視投影行列はすでにセットされているものとする
int pickObject(int x, int y) {
    glClearColor(1,1,1,1);    // 背景色を白に設定
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnable(GL_DEPTH_TEST); // デプステストは有効化
    glDisable(GL_LIGHTING);  // 照明は無効化

    for (int i=0; i<NUM_OBJECTS; i++) {
        // Red だけで 0 から 254 までを区別できる (255 は背景)
        glColor3ub(i, 0, 0); drawObject(i);
    }

    GLubyte c[3]; // 座標 (x,y) の色を取得
    glReadPixels(x,y,1,1, GL_RGB, GL_UNSIGNED_BYTE, c);

    return (c[0] == 255) ? -1 : (int)c[0];
} // glutSwapBuffers() がなければ物体は画面には表示されない
```



C++言語による幾何プログラムの 基礎



3 次元ベクトルの表現

- 3 要素の配列を使った表現

```
// 変数の宣言
```

```
double a[3], b[3], c[3];
```

```
// 初期化
```

```
a[0] = 1; a[1] = 2; a[2] = 3;
```

```
b[0] = 2; b[1] = 3; b[2] = 4;
```

```
// ベクトルの加算
```

```
c[0] = a[0] + b[0];
```

```
c[1] = a[1] + b[1];
```

```
c[2] = a[2] + b[2];
```

```
// 内積
```

```
double d =
```

```
a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
```

```
// 外積
```

```
c[0] = a[1] * b[2] - a[2] * b[1];
```

```
c[1] = a[2] * b[0] - a[0] * b[2];
```

```
c[2] = a[0] * b[1] - a[1] * b[0];
```

```
// 表示
```

```
printf("(%lf,%lf,%lf)", a[0], a[1], a[2]);
```

- これを毎回書くのは面倒...



3次元ベクトルの表現

- 構造体を使った表現

```
struct Vector3d  
{ double x, y, z; };
```

```
// 変数の宣言
```

```
Vector3d a, b, c;
```

```
C言語なら struct Vector3d と書く
```

```
// 初期化
```

```
a.x = 1; a.y = 2; a.z = 3;
```

```
b.x = 2; b.y = 3; b.z = 4;
```

```
// ベクトルの加算 (省略)
```

```
// 内積
```

```
double d =
```

```
    a.x*b.x + a.y*b.y + a.z*b.z;
```

```
// 外積
```

```
c.x = a.y * b.z - a.z * b.y;
```

```
c.y = a.z * b.x - a.x * b.z;
```

```
c.z = a.x * b.y - a.y * b.x;
```

```
// 表示
```

```
printf("(%.1f,%.1f,%.1f)", a.x, a.y, a.z);
```

- 配列とあまり変わらない...毎回書きたくない...



3次元ベクトルの表現

- 関数を使ったら楽になる？

```
struct Vector3d  
{ double x, y, z; };
```

```
// 変数の宣言  
Vector3d a, b, c;
```

```
// 初期化  
Vector3dInit(a, 1, 2, 3);  
Vector3dInit(b, 2, 3, 4);
```

```
// ベクトルの加算  
Vector3dAdd(c, a, b);
```

```
// 内積  
double d = Vector3dDot(a, b);
```

```
// 外積  
c = Vector3dCross(a, b);
```

```
// 表示  
Vector3dPrint(a);
```

直感的でない...
例えば加算なら $c = a + b$; と書きたい



3次元ベクトルの表現

- C++ の機能を使うと...

```
class Vector3d { // 内積
public:          double d = a * b;
    double x, y, z; // メンバ変数
    // メンバ関数を実装 // 外積 (% の演算は自分で定義)
};              c = a % b;

// 変数の宣言と初期化 // 表示
Vector3d a(1,2,3), b(2,3,4), c; a.print();

// ベクトルの加算
c = a + b;
```

- より直感的に書ける！

(演算子と演算の内容を自分で定義できる)



クラスの定義

- C++ はオブジェクト指向型言語。クラスの定義で、コンストラクタ（初期化処理）と、メンバ関数を定義できる

```
class Vector3d {  
public:  
    double x, y, z; // メンバ変数  
  
    // コンストラクタ...構造体名と同じで戻り値のない関数、初期化方法を記述  
    Vector3d() {} // 引数のない関数は「デフォルトコンストラクタ」  
    Vector3d(double _x, double _y, double _z) { x = _x; y = _y; z = _z; }  
  
    void print() { printf("(%lf,%lf,%lf)", x, y, z); } // メンバ関数のひとつ  
  
    // 他の関数については後述  
};
```



コンストラクタとメンバ関数

- コンストラクタ

```
Vector3d(double _x, double _y, double _z) { x = _x; y = _y; z = _z; }
```

を定義しておく

```
Vector3d a(1,2,3);
```

のように初期化ができる。

```
void print() { printf("(%lf,%lf,%lf)", x, y, z); }
```

を定義しておく

```
a.print();
```

という風に呼び出せる。



演算子のオーバーロード

- 既存の演算子 (+, - など) を上書きして新しい処理を定義できる
- operator というキーワードを使って定義する

```
Vector3d operator+(Vector3d v)
```

```
{
```

```
    Vector3d tmp;
```

```
    tmp.x = x + v.x; tmp.y = y + v.y; tmp.z = z + v.z; // x, y, z はメンバ変数
```

```
    return tmp; // 新しい構造体を作って返す
```

```
}
```

というメンバ関数を用意すると

```
c = a.operator+(b);
```

という風に呼び出せるが、operator と () は省略できるので

```
c = a + b;
```

と書ける！



効率性の問題

- 先ほどの定義では変数の余計なコピーが発生する

```
Vector3d operator+(Vector3d v)
{
    Vector3d tmp;
    tmp.x = x + v.x; tmp.y = y + v.y; tmp.z = z + v.z; // x, y, z はメンバ変数
    return tmp; // 新しい構造体を作って返す
}
```

① 引数をコピーしている

② 計算結果をあとでコピーする必要がある

参考 : $c = a + b$; を計算するために実際に行われること

```
Vector3d tmp1;
tmp1.x = b.x; tmp1.y = b.y; tmp1.z = b.z; // ① 引数のコピー
Vector3d tmp2;
tmp2.x = a.x + tmp1.x; tmp2.y = a.y + tmp1.y; tmp2.z = a.z + tmp1.z;
c.x = tmp2.x; c.y = tmp2.y; c.z = tmp2.z; // ② 計算結果のコピー
// ※ コンパイラが賢ければ無駄な処理は省略されるかもしれない
```



効率性の問題

- 「①引数のコピー」は解消できる

Vector3d operator+(Vector3d v)



Vector3d operator+(const Vector3d &v)

- & は「参照 (reference)」を表し、ポインタを使ったときのように値のコピーではなくアドレス渡しになる
 - 引数名に const を付けると「引数の値は変更しない」という意味になりコンパイラが最適化できるようになる
- 「②計算結果のコピー」はこのやり方だとダメ
 - operator+= を実装して `c = a; c += b;` とすれば計算結果の余計なコピーはなくなる



線形代数ライブラリ

- 3次元ベクトルと同様に、
 - 2次元ベクトル
 - 4次元ベクトル
 - 2x2 行列
 - 3x3 行列
 - 4x4 行列

などの演算やそれらとの相互の演算も用意すると便利

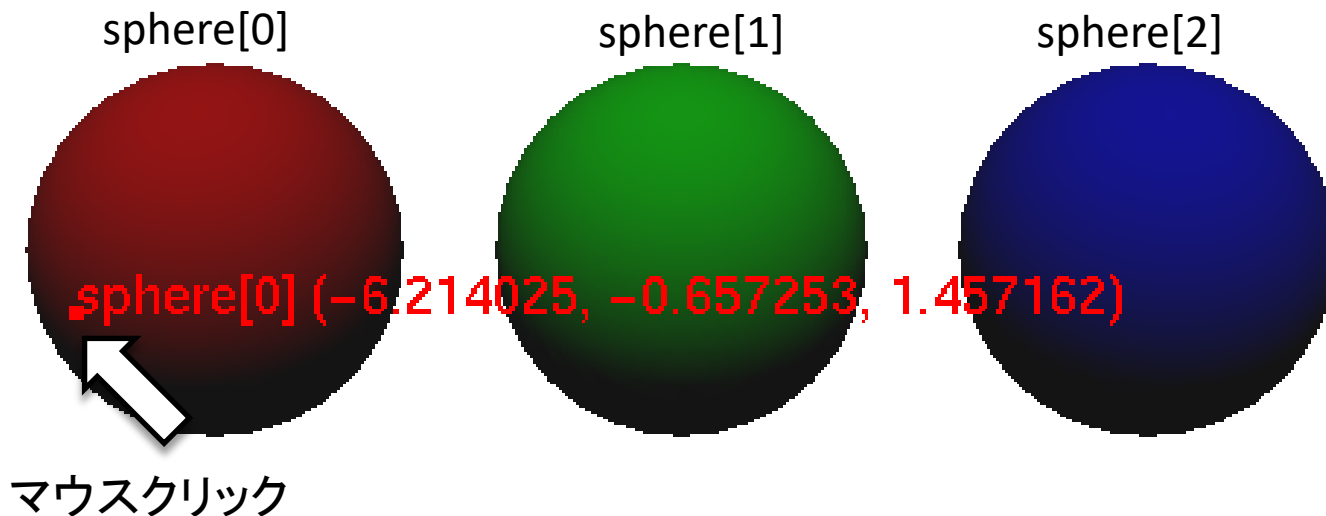
- 世の中にはたくさん線形代数ライブラリがある
 - 有名な例: Eigen <http://eigen.tuxfamily.org/>



課題の概要

(詳しくは課題用資料参照)

- 複数の物体を表示して、クリックした物体の識別と、クリックした点の座標を取得する
 - オプションとして、物体が空間を動き回るようにし、クリックすることで得点を得られるゲームを作成



コンピュータグラフィックス基礎 第4回 課題

課題の目標

- C++の初歩的な機能について理解する
- GLUT を用いた対話的なアプリケーションを作成できるようになる

課題の内容

- (1) サンプルコード `vector3d_example.cpp` で提供されている `Vector3d` クラスに対して、減算を表す演算子(-)の処理と、減算代入を表す演算子(==)の処理を追加しなさい。

また、 $\vec{a} = (2, 3, 4)$, $\vec{b} = (3, 5, -2)$, $\vec{c} = (2, -1, 1)$ のときに、次のベクトル計算を行う処理を `main` 関数に記述し、その結果を出力しなさい。×は外積を、・は内積を表すものとする。

(a) $\vec{a} - \vec{b}$ (b) $3\vec{a} - 2\vec{b}$ (c) $\vec{a} \cdot \vec{b}$

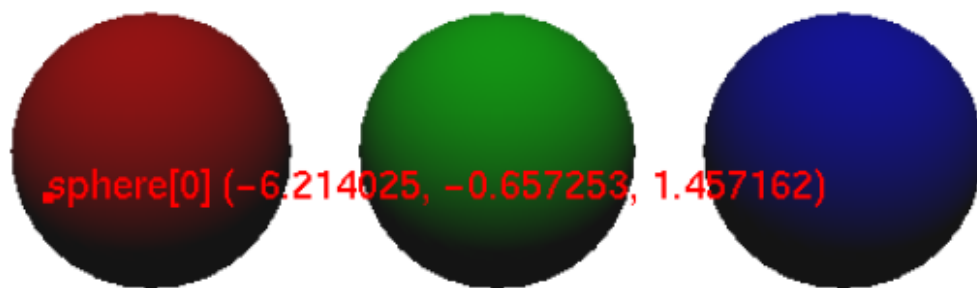
(d) $\vec{a} \times \vec{b}$ の単位ベクトル (e) $(\vec{a} + 2\vec{b}) \times \vec{c}$



- (2) サンプルプログラムコード `kadai04_pick_sphere.cpp` を実行すると、下図のように 3 つの球体が表示される。



★記号で示した箇所を埋めて、プログラムを完成させると、下図のようにマウスクリックで選択した球体の **ID** と、クリックした位置の座標値が画面に描画される。必要なコードを追加してプログラムを完成させなさい。



Mouse Picking



(発展課題：オプション)

課題 3 で学習した内容を活用し、球体が空間を動き回ったり、視点の位置がキーボードで変わるようにし、球体をクリックできた場合に得点が加算されるようなゲームアプリケーションを作りなさい。

提出するレポートに含めるもの

上記の課題のプログラムコードと、実行結果のウィンドウをキャプチャした図。課題(1),(2)のプログラムコードについては、自分で作成した部分と、元からある部分が区別できるように示すこと。

発展課題で動きのあるものは、動いている様子が分かるように、複数のキャプチャ画像を含めること。
mp4 やアニメーション GIF などの動画ファイルを作成できる場合は、動画を含めること。

