コンピュータアニメーション特論

第１回　コンピュータアニメーションの基礎

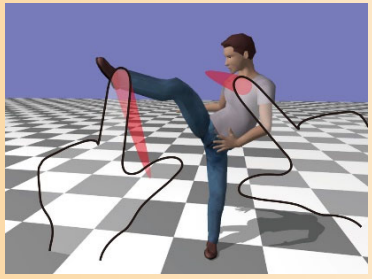
九州工業大学 情報工学研究院　尾下真樹

本日の内容

- ガイダンス
- コンピュータグラフィックスの概要と応用
- 復習：３次元グラフィックスの要素技術
- 復習：３次元グラフィックスのプログラミング
- 復習：OpenGL&GLUT プログラミング
- 復習：OpenGL&GLUT サンプルプログラム

授業担当

- 尾下 真樹 （おした まさき）
  - 九州工業大学 情報工学研究院
    知能情報工学研究系
  - e-mail: oshita@ai.kyutech.ac.jp
  - http://www.ha.ai.kyutech.ac.jp
  - 研究内容
    - コンピュータアニメーション、コンピュータグラフィックス

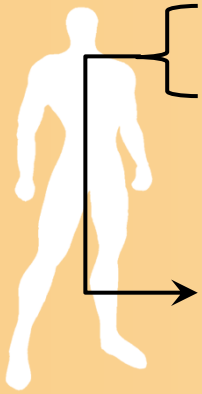


本科目の達成目標

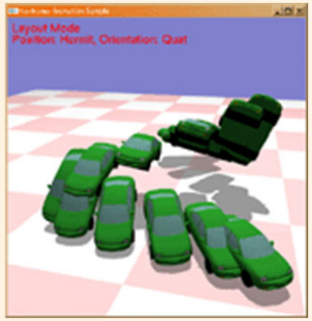
- コンピュータアニメーションの応用技術を学習する
- コンピュータアニメーションを利用したソフトウェアを開発する上で役に立つ最新技術を、実際の演習を交えながら習得する

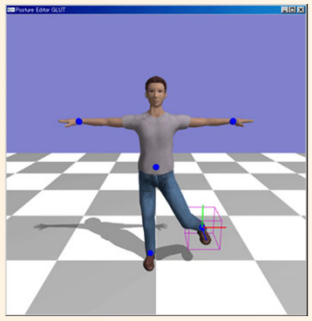
本科目の内容

- 本科目で学ぶ応用技術
  - 視点操作
  - 幾何形状データの読み込み
  - キーフレームアニメーション
  - 物理シミュレーション
  - 衝突判定、ピッキング
  - キャラクタアニメーション
  - レンダリング技術



特にキャラクタアニメーション
関連の技術を重点的に扱う





本科目の意義

- 本大学院の学生で、コンピュータグラフィックスを専門とする研究・仕事に就く人は少ない？
  - ゲームプログラマ、アニメーション制作者など
  - 本講義の内容は、これらの仕事に密接に関係
- 他の多くのソフトウェアでも、コンピュータグラフィックスが要素技術として使われる
  - ロボットのシミュレーション結果の表示
  - 仮想空間や仮想人間のアニメーション
  - 3次元物体のシミュレーション結果の可視化、等

本科目の意義（続き）

- 研究や仕事で、コンピュータグラフィックスを用いるプログラムを開発する機会があれば、本講義で扱う内容が役に立つ

想定する受講者

- コンピュータグラフィックの基礎を理解している
  - レンダリングの仕組み、座標変換などの考え方（学部のコンピュータグラフィックス科目の内容）
  - C言語＋OpenGL を使ったプログラミング
  - 今回の授業でこれらの基礎知識も簡単に復習
- C/C++を使ったプログラミングができる
  - 標準的な学部４年生程度のプログラミング能力
  - 具体的なアルゴリズムが与えられれば、自力でプログラムを作成できる程度の能力があること
  - 本科目のプログラミング演習で必要となる、やや高度なプログラミング技術も、本授業で扱う

授業の進め方

- 講義
  - 基本的に PowerPoint の資料を使用して講義
  - 教科書はなし
    - 参考書や参考文献は各テーマごとに紹介
- 演習
  - 講義の説明に従って、プログラミング演習を行う
  - 各自が利用可能な端末を使用する
    - C言語 + OpenGL + GLUT（freeglut） + vecmath
      - 詳細は本科目のウェブページの演習環境の説明を参照
    - Windows + Visual Studio を標準環境とする

遠隔授業の実施方法

- 毎回の授業の資料（講義動画、演習問題、演習課題）を Moodle に公開する
  - 指定された期間内（通常、授業の翌日まで）に、講義動画の視聴と演習問題への回答を行う
    - これらを全て行った場合のみ、出席扱いとする
  - 指定された期限までに、プログラミング演習課題を行って、レポートを提出する
- 質問には、Moodle のフォーラムや電子メール、対面授業（Moodleから案内）で対応
  - 対面授業は、質問がない人は出席は不要

講義資料

- 講義資料は、授業のウェブページで公開
- プログラミング演習・レポート課題に関する資料も、本ページで公開
- 講義動画の公開や、演習問題や演習課題レポートの提出には、Moodle を使用

本科目のウェブページ：

http://www.ha.ai.kyutech.ac.jp/lecture/aca/

成績評価

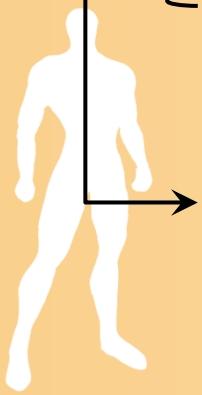
- プログラミング演習課題（80%）
  - 演習課題のプログラムを作成し、レポートを提出
  - レポート提出後に演習問題（ミニテスト）を実施
- 演習問題（20%）
  - 毎回の授業に出席して、講義内容を理解する
  - 授業中に演習問題（ミニテスト）を実施
  - 授業後に復習用の演習問題を公開する
    - 復習問題の点数も成績に考慮する

プログラミング演習課題

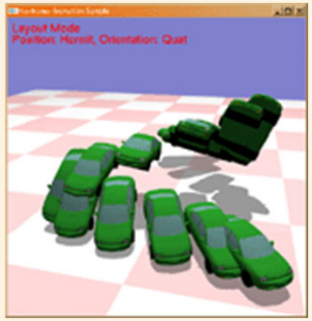
- 各テーマごとに、講義で学習した技術を応用したプログラミング課題のレポートを出す
- レポート課題の内容
  - 元になるサンプルプログラム（大部分は作成済みのプログラム）を、授業のウェブページで公開
  - 講義で説明した重要な部分の処理を、各自作成
  - 作成したプログラムをレポートとして提出
    - 文章でのレポートの作成・提出は不要
  - レポート提出後に、理解度を確認するための Moodle ミニテストを実施

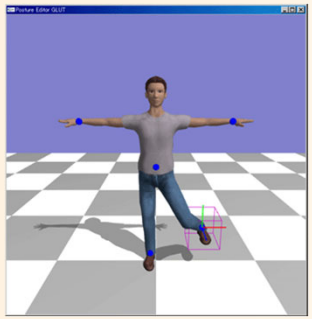
本科目の内容

- 本科目で学ぶ応用技術
  - 視点操作
  - 幾何形状データの読み込み
  - キーフレームアニメーション
  - 物理シミュレーション
  - 衝突判定、ピッキング
  - キャラクタアニメーション
  - レンダリング技術

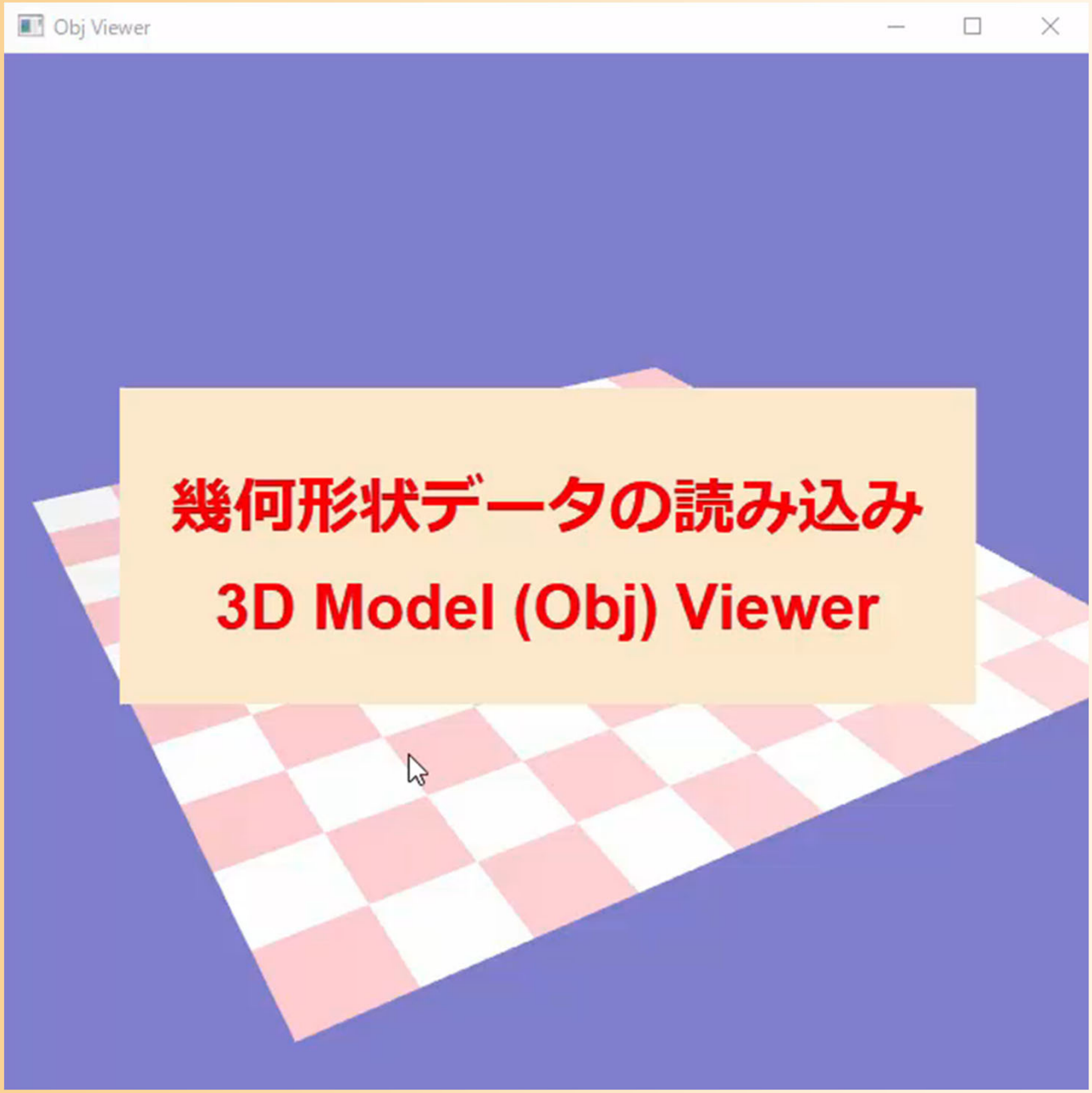


特にキャラクタアニメーション
関連の技術を重点的に扱う






View Control
Dolly Mode (Parameter)
視点操作
Camera Control


Obj Viewer
幾何形状データの読み込み
3D Model (Obj) Viewer


GLUT sample program
No Shadow
影の描画
Shadow


Keyframe Animation
Layout Mode
Position: Linear, Orientation: Eular
キーフレームアニメーション
Keyframe Animation


Simulation test environment v0.02
File Simulation Help
物理シミュレーション
Pysics Simulation (ODE)

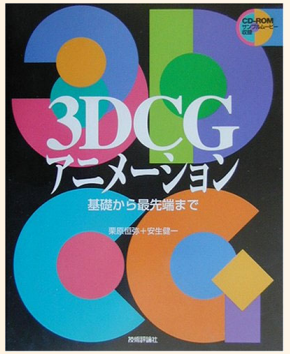

Picking
Picking on Screen
ピッキング
Picking


Human Animation Sample
Motion Playback
0.05 (5)
キャラクタ アニメーション
Human Animation

基礎知識に関する参考書

- 「コンピュータグラフィックス 改訂新版」
  CG-ARTS協会 編集・出版（3,600円）

- 「3DCGアニメーション」
  栗原恒弥 安生健一 著、技術評論社
  出版（2,980円）

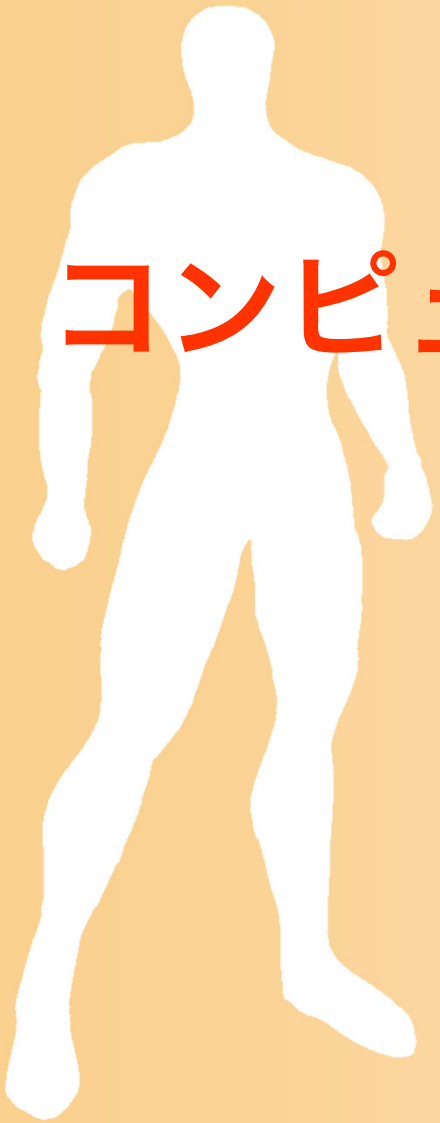


参考：CGエンジニア検定

- CGエンジニア検定（CG-ARTS協会）
  - ベーシック／エキスパート
  - 年２回実施、福岡でも受験可能
  - 学部のコンピュータグラフィックス相当の内容
    - CG-ARTS協会のテキストの内容を理解していれば、合格できる
    - エキスパートは、やや応用的な問題が出題される
  - 本学の受験者は割引を受けられる

本日の内容

- ガイダンス
- コンピュータグラフィックスの概要と応用
- 復習：３次元グラフィックスの要素技術
- 復習：３次元グラフィックスのプログラミング
- 復習：OpenGL&GLUT プログラミング
- 復習：OpenGL&GLUT サンプルプログラム



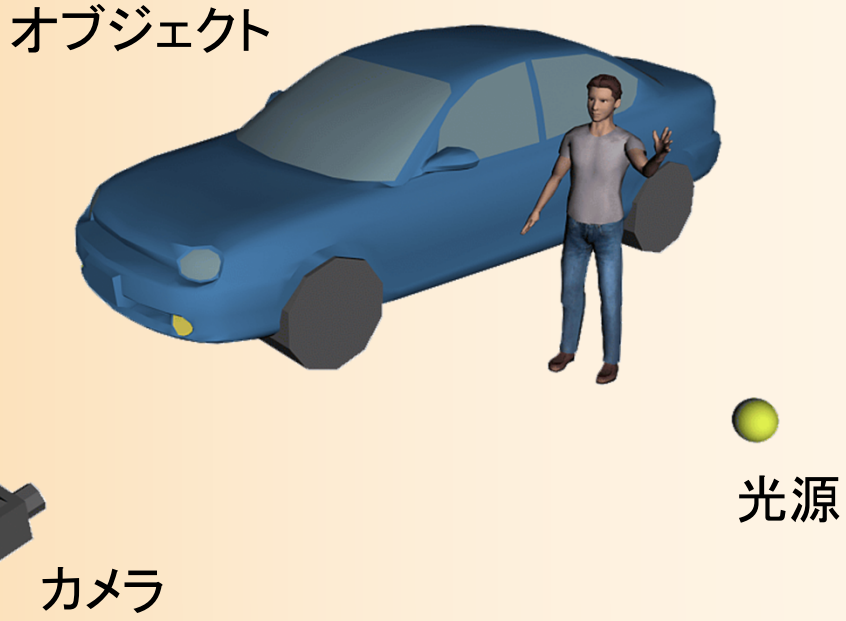
コンピュータグラフィックスの概要と応用

コンピュータ・グラフィックス

- 広い意味でのコンピュータ・グラフィックス
  - コンピュータを使って視覚データを扱う技術
  - 画像・動画、ユーザインターフェースなど、2次元グラフィックスを含む
- 狭い意味でのコンピュータ・グラフィックス
  - 3次元の形状・空間データを扱い、計算によって画像を生成する技術
    - ただし、最終的なアウトプットとしては、2次元の画像データになることが多い
  - この講義では、こちらを主に扱う

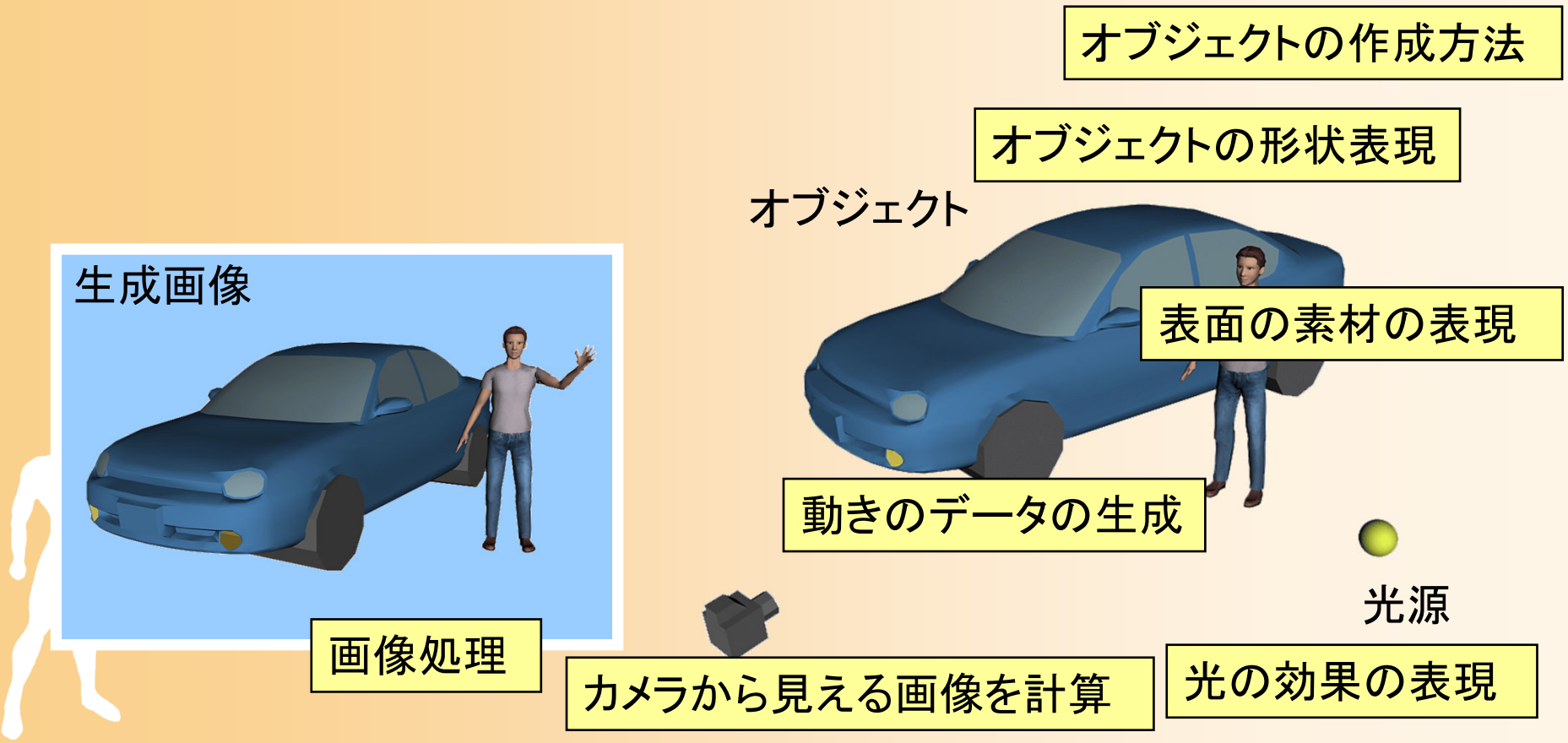
グラフィックスの要素技術

- ＣＧ画像を生成するためのしくみ
  - 仮想空間にオブジェクトを配置
  - 仮想的なカメラから見える映像を計算で生成
  - オブジェクトやカメラを動かすことでアニメーション



グラフィックスの要素技術

- コンピュータグラフィックスの主な技術



コンピュータグラフィックスの技術

- コンピュータグラフィックスの主な技術

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト

表面の素材の表現

動作データの生成

光源

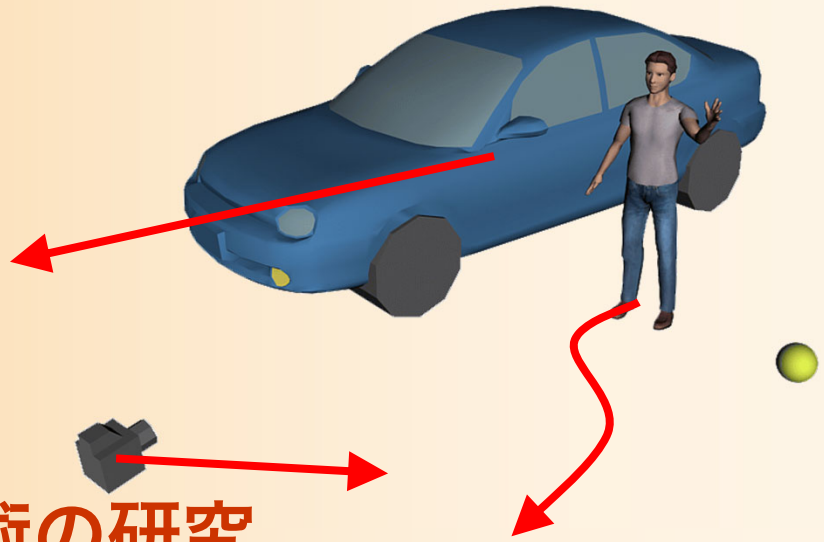
光の効果の表現

画像処理

カメラから見える画像を計算

アニメーションの技術

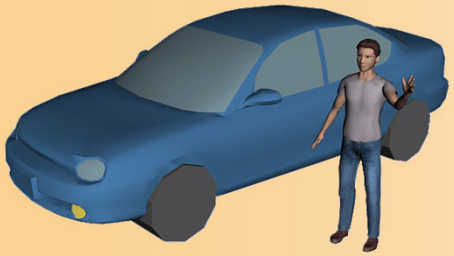
- アニメーション ＝ 動作の生成 ＋ 画像生成
  - ３次元空間の中で
    オブジェクトに動作
    のデータを与える
  - 動作のデータを
    どう生成するか？



- 動作データを扱う技術の研究

他のメディア処理技術との関係

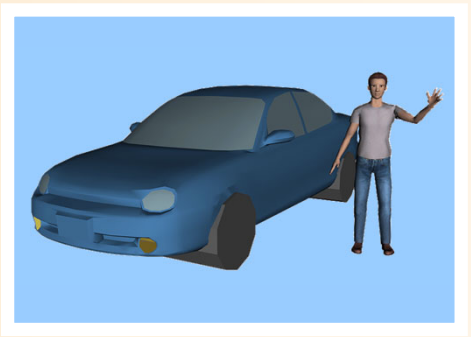
- 音声（1次元）
- 画像・動画（2次元）
- コンピュータグラフィックス（3次元）
  - コンピュータビジョンとは逆方向の技術



3次元 空間・形状データ

コンピュータ
グラフィックス

コンピュータ
ビジョン



2次元 画像・動画

グラフィックス応用の分類

- オフライン画像生成
  - 静止画、映画など
  - アニメータらが多くの時間をかけて制作
- オンライン画像生成
  - コンピュータゲーム、シミュレーション、VR など
  - ユーザの操作などに応じて、インタラクティブに画像を生成する必要がある
    - ただし、物体の形状データやモーションデータは、前もって作成されていることが多い

グラフィックス応用による違い

- 基本的な原理や考え方は共通
- オフライン画像生成のための技術
  - 多少時間がかかっても良いので、高画質の画像を生成する必要がある
  - 基本的には既存のソフトウェアが利用可能なので、各技術の概要を大体理解していれば、実用には十分
- オンライン画像生成のための技術
  - 多少画質を犠牲にしても、リアルタイムに画像を生成する必要がある
  - 自分でプログラムを作成しなければならないことが多いため、技術を具体的に理解しておく必要がある
  - 本授業では、こちらの応用のための技術を扱う

市販ソフトウェアの利用

- 市販のアニメーション制作ソフトウェアが広く使われている
  - Maya, 3ds max , Softimage, LightWave など
- 既存のソフトウェアがサポートする技術は、自分でわざわざプログラムを開発しなくとも良い場合もある
  - 形状データやモーションデータの作成など
  - うまく組み合わせることが重要

必要となるプログラミング

- 市販ソフトウェアと組合わせたプログラムの例
  - 市販ソフトウェアを使って形状データをモデリング
    → 自作のプログラムで有限要素法シミュレーションを行い、解析結果をCGで描画
  - 市販ソフトウェアを使い、キャラクタの骨格・形状データや動作データを作成
    → 自作のプログラムでは、作成したデータを読み込み、ユーザの操作に応じてキャラクタを動かして描画
- オンライン・アプリケーションでは、描画処理や物体制御などは、自分で作成する必要がある
  - 最近は、既存のミドルウェアやゲームエンジン（Unity、Unreal等）を用いる場合もある

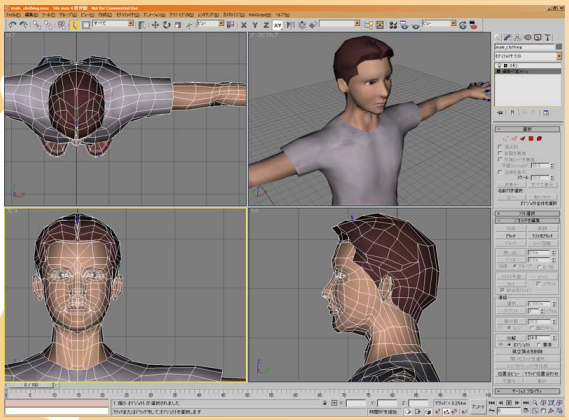
ミドルウェアの利用

- ミドルウェアやゲームエンジンの利用
  - Unity、Unreal 等
  - 近年、コンピュータゲーム以外のアプリケーション開発にも広く使われている
  - 基本的な処理は、自分でプログラムを作成しなくとも、ミドルウェアの機能により実現できる
    - 原理はきちんと理解している必要がある
    - 必要に応じて機能を拡張することは難しい
  - 本講義では、ミドルウェアの利用方法は扱わないが、一部のテーマで、最近のミドルウェアの機能との関連についても紹介する

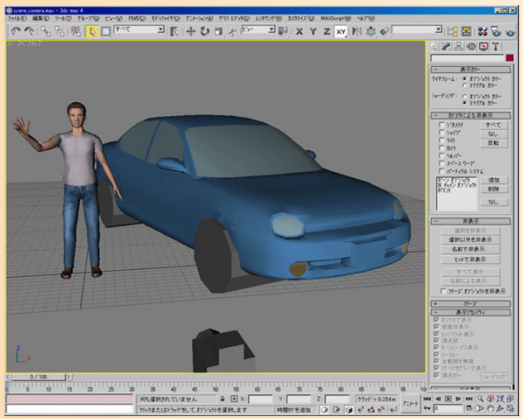
市販ソフトウェアの利用

- 一般的なCG作品制作の流れ
  - モデリング・・・各物体ごとの形状データを作成
  - レイアウト・・・空間に物体を配置、動きを設定
  - レンダリング・・・静止画像orアニメーション生成

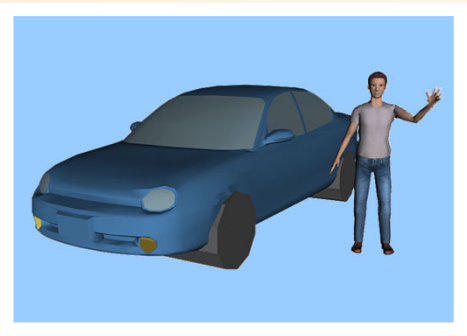
モデリング



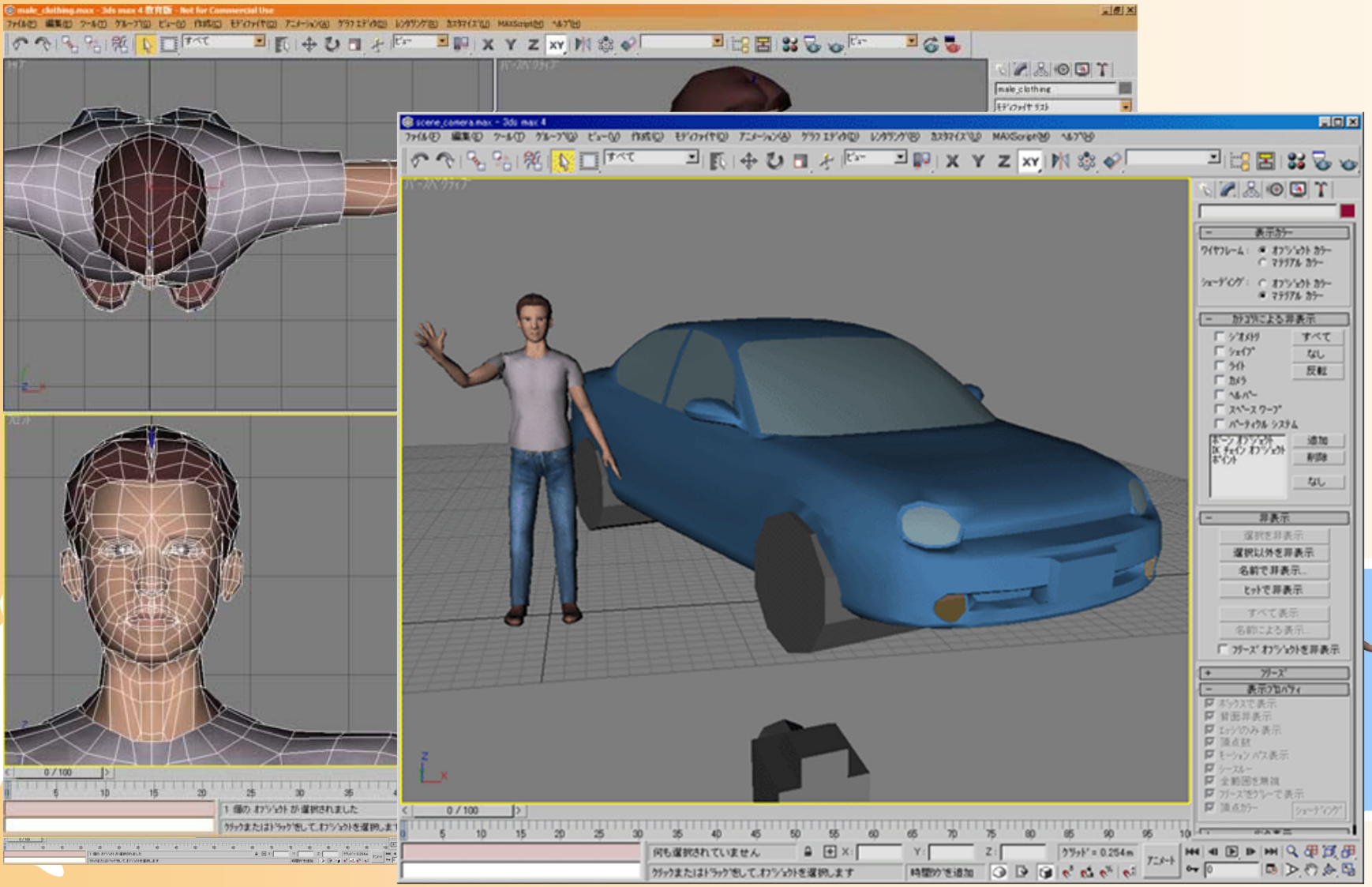
レイアウト



レンダリング



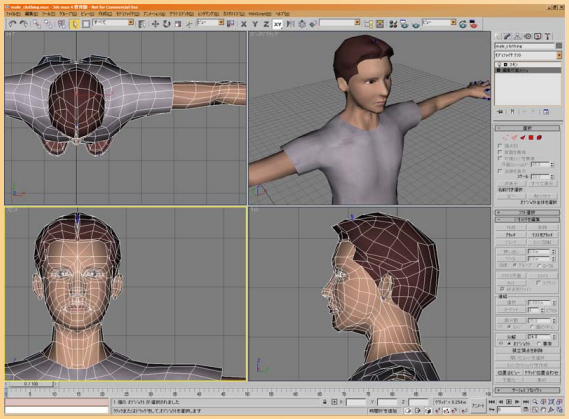
市販ソフトウェアの利用



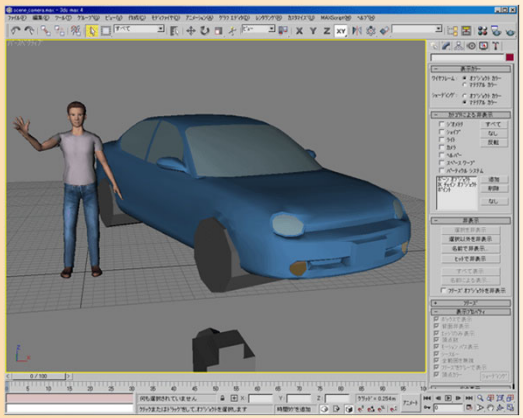
市販ソフトウェアの利用

- 既存ソフトウェアと組み合わせたプログラミング

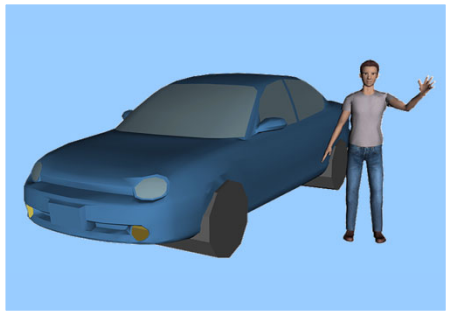
モデリング



レイアウト

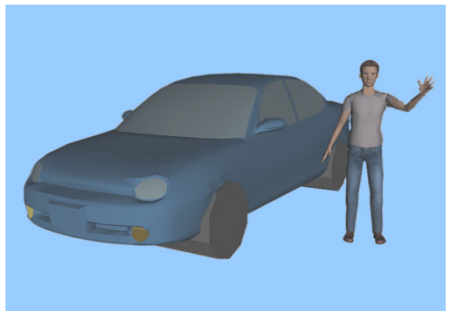


レンダリング



高品質な描画

高速な描画



形状データ

シーンデータ
動作データ

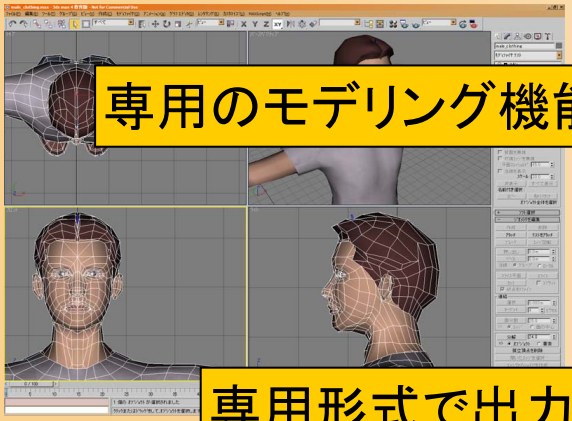
プログラム

ファイルからデータを読み込み、必要に応じて動きを生成しながら、リアルタイムにレンダリング

市販ソフトウェアの利用

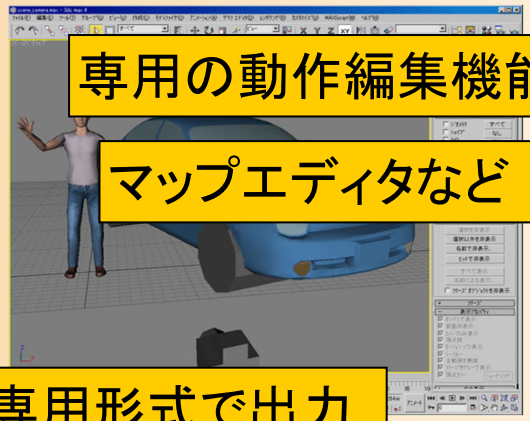
- プラグインによる拡張が可能

モデリング



形状データ

レイアウト

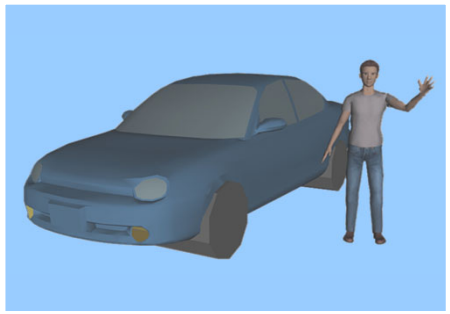


シーンデータ
動作データ

使い慣れたソフトウェアに、必要な機能だけを追加することができるので、効率的

高品質な描画

高速な描画

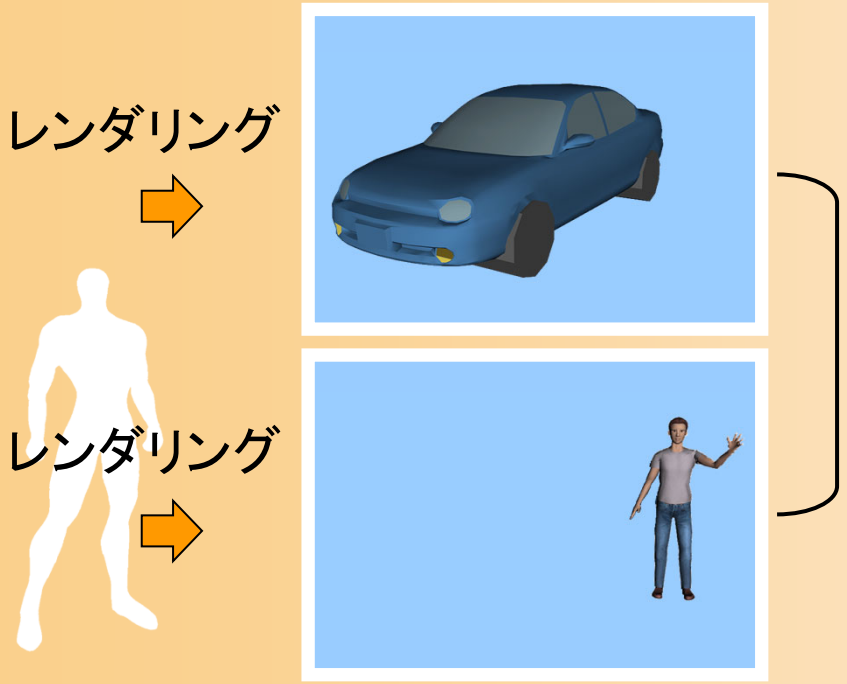


プログラム

映画制作・ゲーム制作などでは、各プロダクションごとに、独自のプラグインを多数使用

オフライン・アニメーション

- コンポジションも重要な作業のひとつ
  - 実際の制作では、各オブジェクトや効果を個別にレンダリングして、後でまとめて合成・編集することが多い （専用の編集ソフトを使用）



コンポジションの利用

- 映画などでは、フルCGよりも、実写＋ＣＧの合成が多い
  - 実写では実現できないような映像のみをＣＧで表現
  - 我々の身近にある物、特に人間などはＣＧで再現することが難しい
    - 少しでも不自然なところがあるとすぐに目立つ
  - あまり身近にないような物、実写では絶対に撮影できないような物をＣＧで作り出す
  - 実写で撮影可能なものにはＣＧは使わず、実写とＣＧを合成

3Dグラフィックスと実写の関係

- 3Dグラフィックス
  - 制作には労力がかかる
  - 存在しないものも表現可
  - 人間などの再現は難しい
- 実写
  - 実物をそのまま撮影できる
  - 人間などは実写の方が向いている

Jurassic Park III
Universal Pictures

- 両者をうまく使い分けて撮影・生成し、最終的に合成して映像を作る方法が一般的

グラフィックス分野の課題（1）

- 写実的な画像を高速に生成
  - 光の働きは物理現象であるため、理論的には、精確な３次元データと十分な計算時間があれば、現実世界と同一の現象（画像）を再現できる
  - 実際には、準備できる３次元データや計算機の能力には限りがあるため、実用的な品質・速度を実現するための技術が必要となる
  - 複数の要素技術の組み合わせが必要であるため、各要素技術の工夫や組み合わせ方の工夫が必要となる

グラフィックス分野の課題（2）

- 生成画像・映像の品質の評価
  - 定量的な比較・評価を行うことが難しい
    - 他の工学・情報工学分野との違い
  - 写実的な画像生成を目的としない場合もある（ノンフォトリアリスティックレンダリングなど）
- 人間（クリエータ）による入力が重要
  - 元データを容易に作成するための技術や、対話的なインターフェースの技術なども必要
- 他の分野と同様、まだ多くの課題がある

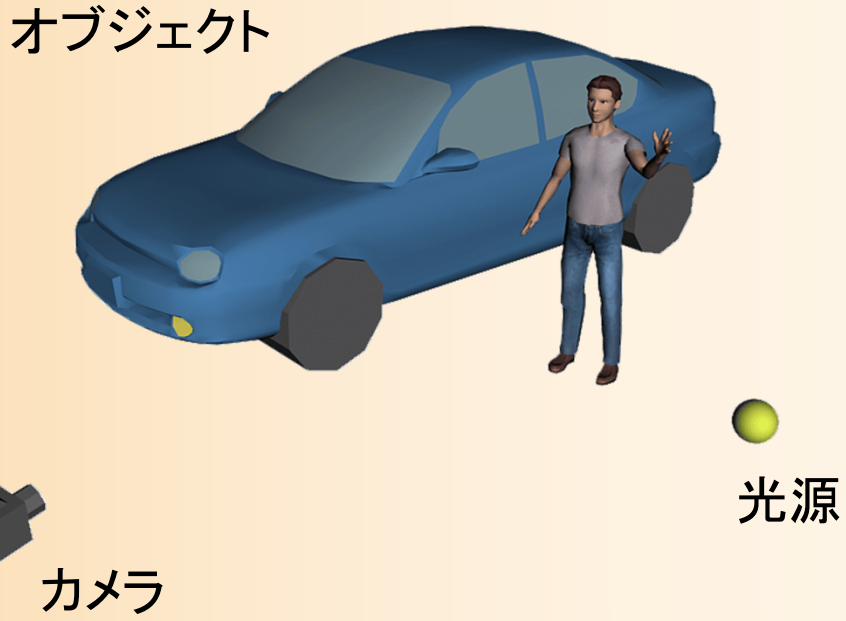
本日の内容

- ガイダンス
- コンピュータグラフィックスの概要と応用
- 復習：３次元グラフィックスの要素技術
- 復習：３次元グラフィックスのプログラミング
- 復習：OpenGL&GLUT プログラミング
- 復習：OpenGL&GLUT サンプルプログラム

３次元グラフィックスの要素技術

３次元グラフィックスのしくみ（復習）

- ＣＧ画像を生成するための方法
  - 仮想空間にオブジェクトを配置
  - 仮想的なカメラから見える映像を計算で生成
  - オブジェクトやカメラを動かすことでアニメーション



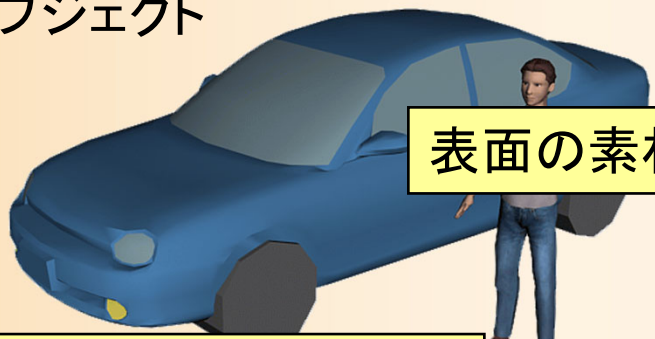
３次元グラフィックスの要素技術

- コンピュータグラフィックスの主な技術

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト



表面の素材の表現

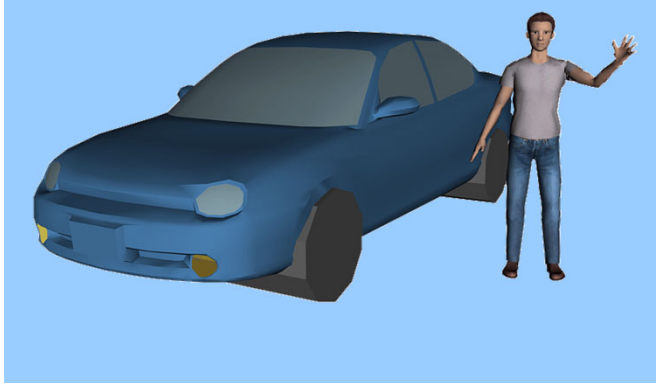
動きのデータの生成

光源

光の効果の表現

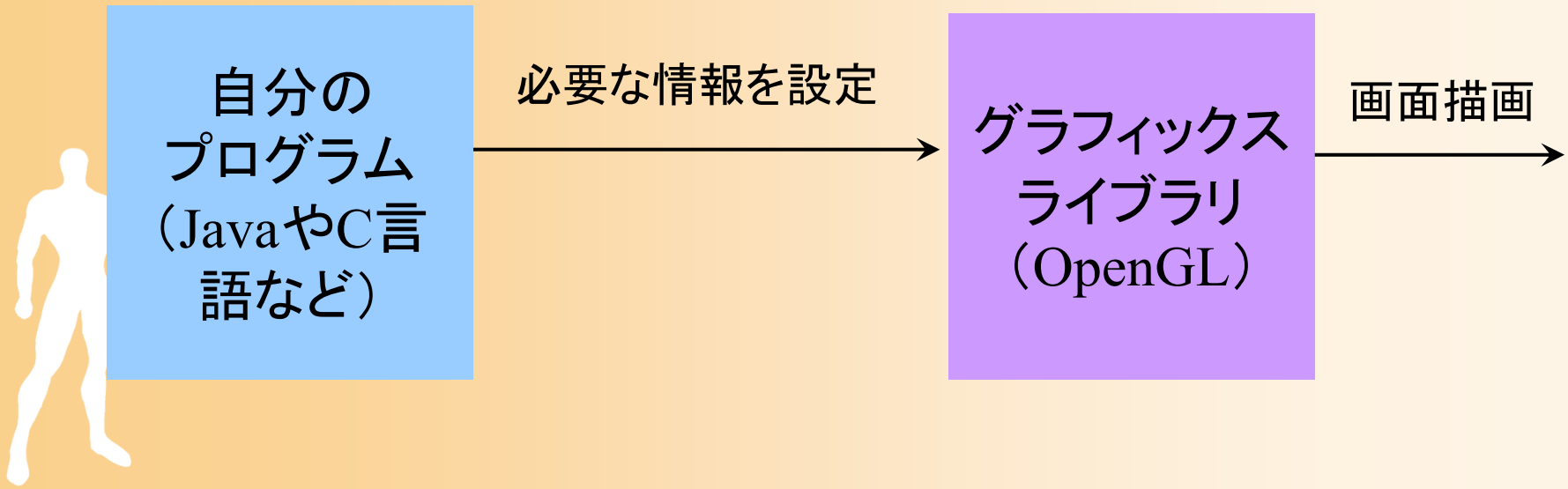
カメラから見える画像を計算

生成画像



グラフィックスライブラリの利用

- グラフィックスライブラリ（OpenGLなど）
  - 要素技術を簡単に利用できる
  - 要素技術の仕組みは理解する必要がある



モデリング

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト

表面の素材の表現

動きのデータの生成

光源

カメラから見える画像を計算

光の効果の表現

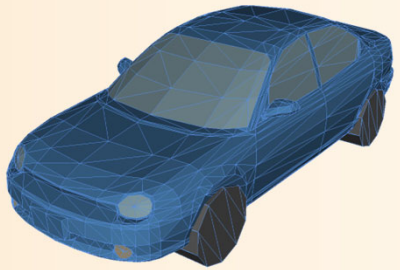
モデリング

- モデリング（Modeling）
  - コンピュータ上で、物体の形状のデータを扱うための技術
  - 形状の種類や用途によって、さまざまな表現方法がある
  - 形状データの表現方法だけでなく、どのようにしてデータを作るかという、作成方法も重要になる

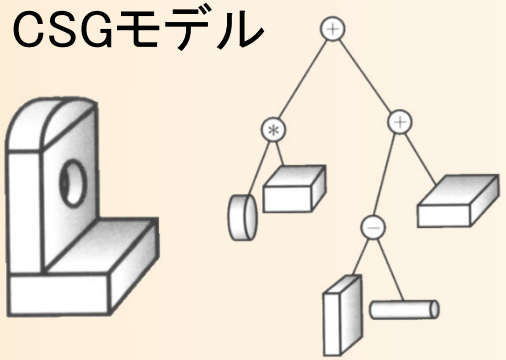
各種モデリング技術

- サーフェスモデル
  - ポリゴンモデル
  - 曲面パッチ
  - サブディビジョンサーフェス
- ソリッドモデル
  - 境界表現
  - CSGモデル
- その他のモデル

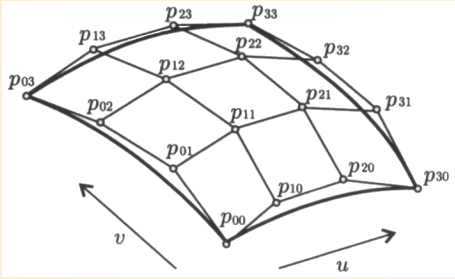
ポリゴンモデル



CSGモデル

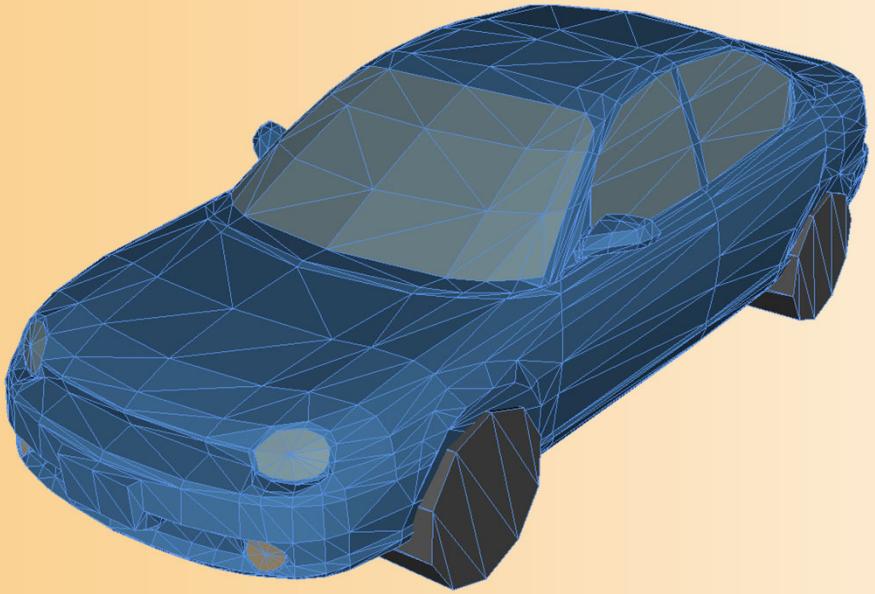


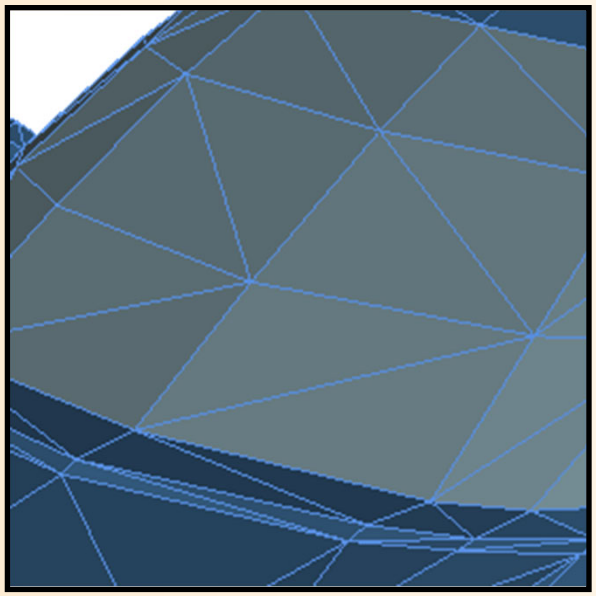
曲面パッチ



ポリゴンモデル

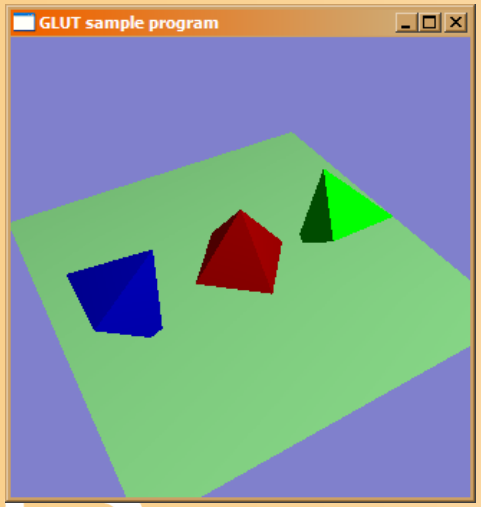
- 物体の表面の形状を、多角形（ポリゴン）の集まりによって表現する方法
  - 最も一般的なモデリング技術
  - 本講義の演習でも、ポリゴンモデルを扱う

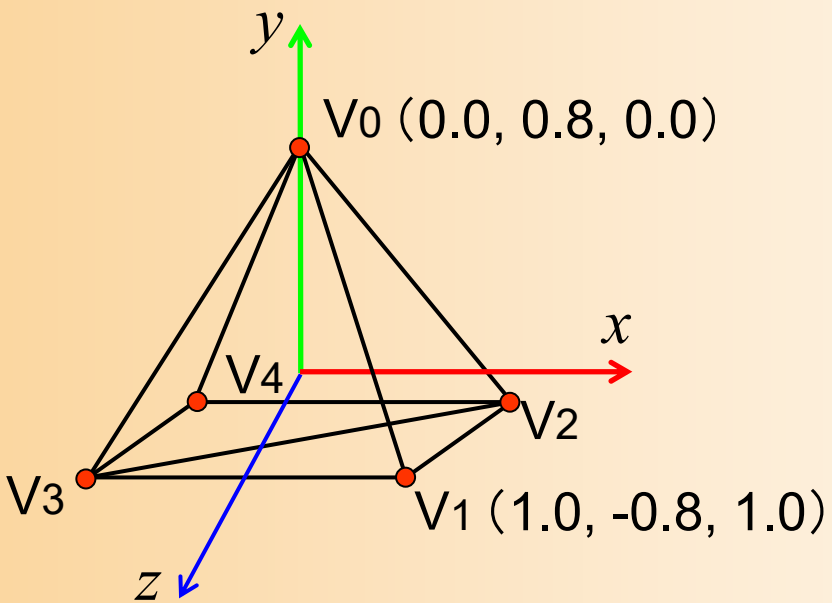




ポリゴンモデルの表現例

- 四角すいの例
  - ５個の頂点 と ６枚の三角面（ポリゴン） によって表現できる
    - 各三角面は、どの頂点により構成されるかという情報を持つ





三角面

{ V0, V3, V1 }

{ V0, V2, V4 }

{ V0, V1, V2 }

{ V0, V4, V3 }

{ V1, V3, V2 }

{ V4, V2, V3 }

ポリゴンモデルの表現例（続き）

- プログラムでの表現例（配列による表現）
  - 頂点座標の配列
  - ポリゴンを構成する頂点番号の配列

```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 },
          {-1.0,-0.8, 1.0 }, {-1.0,-0.8,-1.0 }
};

// 三角面インデックス（各三角面を構成する頂点の頂点番号）の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};
```

レンダリング

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト

表面の素材の表現

動きのデータの生成

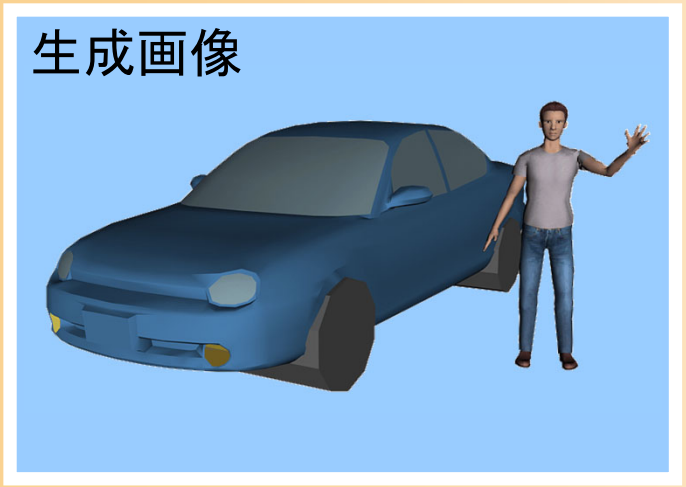
光源

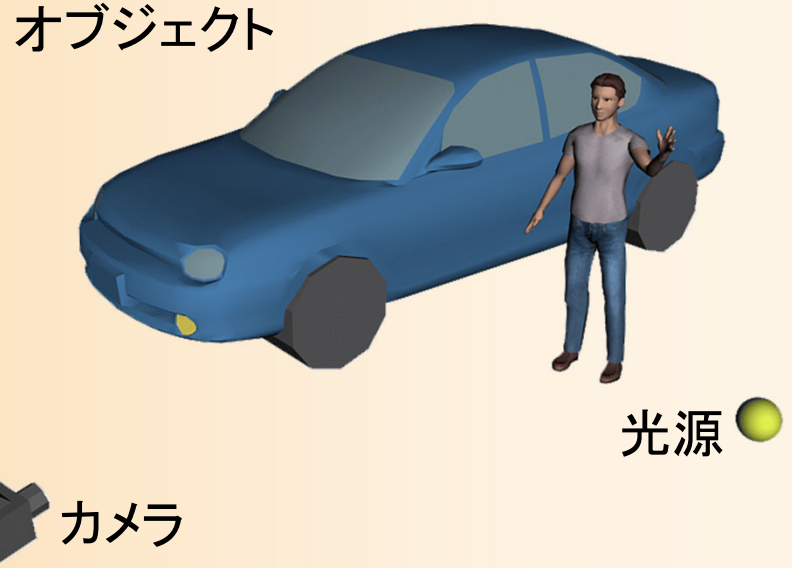
カメラから見える画像を計算

光の効果の表現

レンダリング

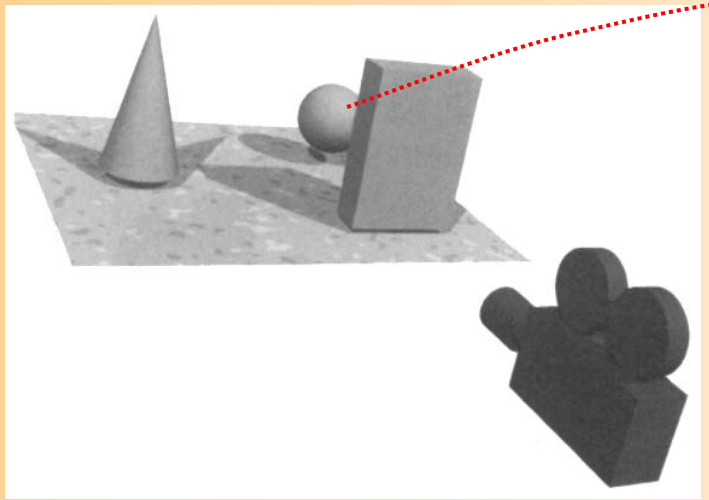
- レンダリング（Rendering）
  - カメラから見える画像を計算するための方法
  - 使用するレンダリングの方法によって、生成画像の品質、画像生成にかかる時間は変わる



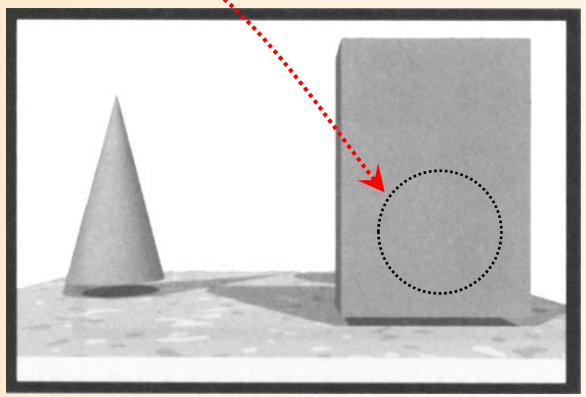


レンダリングの重要なポイント

- 隠面消去をどのようにして実現するか？
  - 見えるはずのない範囲を描画しない処理
  - 存在する面を全て描いたら、見えるはずのない面まで表示されてしまう



この球は手前の直方体で隠れるため描画しない



参考書「コンピュータグラフィックスの基礎知識」図2-21

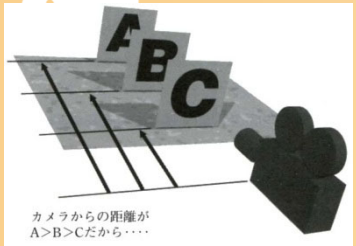
各種レンダリング手法

- 主なレンダリング手法
  - Zソート法
  - Zバッファ法
  - スキャンライン法
  - レイトレーシング法
    - 隠面消去の実現方法が異なる

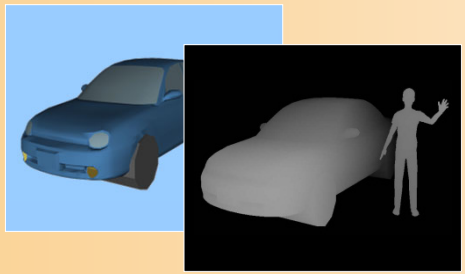
低画質、高速度

高画質、低速度

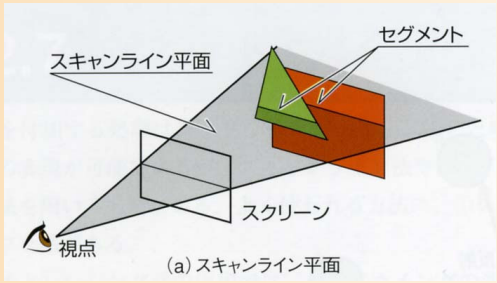
Zソート法



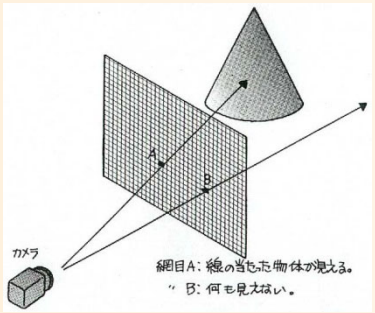
Zバッファ法



スキャンライン法

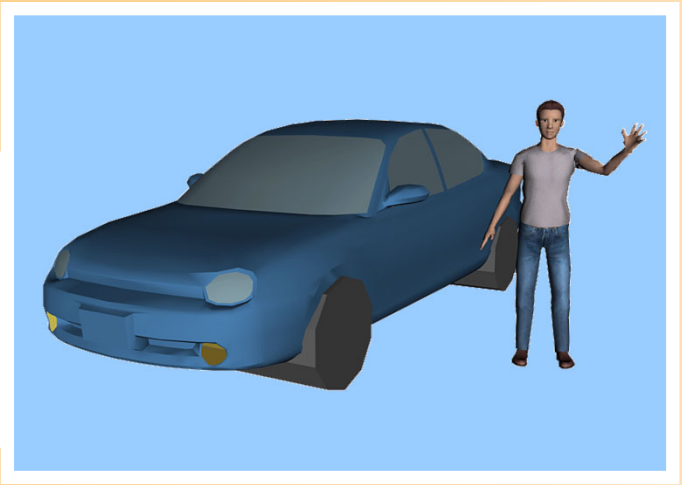


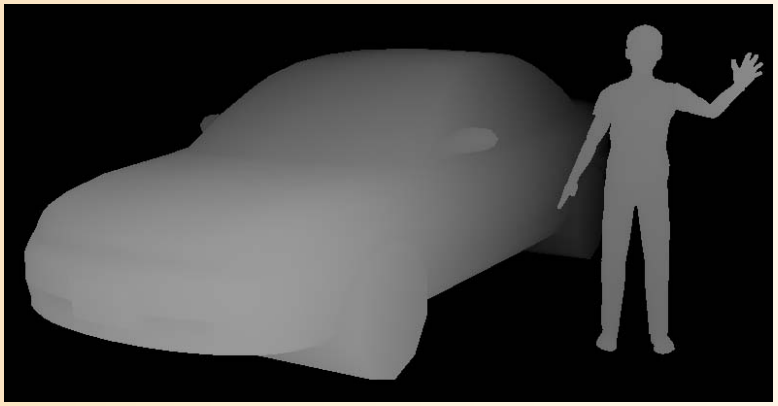
レイトレーシング法



Ｚバッファ法

- Ｚバッファ法
  - 描画を行う画像とは別に、画像の各ピクセルの奥行き情報を持つ Ｚバッファ を使用する
  - コンピュータゲームなどのリアルタイム描画で、最も一般的な方法（本講義の演習でも使用）

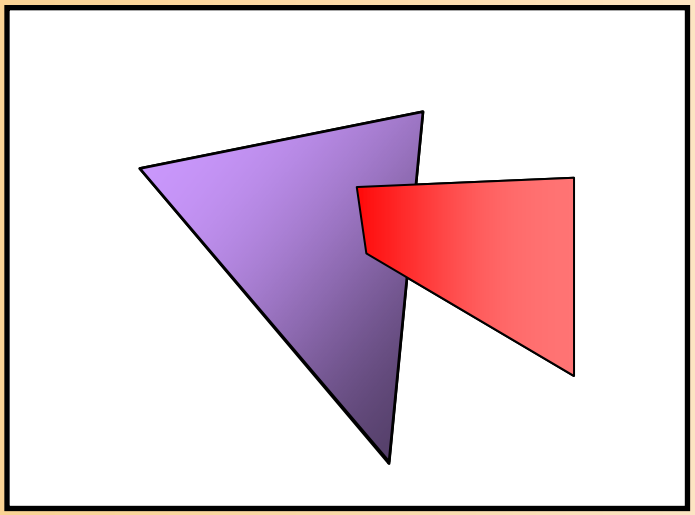


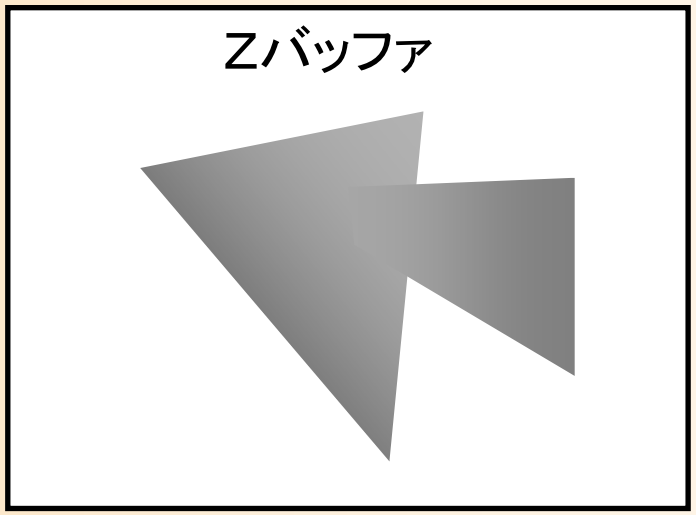


Ｚバッファの値（手前にあるほど明るく描画）

Ｚバッファ法による隠面消去

- Ｚバッファ法による面の描画手順
  - 面を描画するとき、各ピクセルの奥行き値（カメラからの距離）を計算して、Ｚバッファに描画
  - 同じ場所に別の面を描画するときは、すでに描画されている面より手前のピクセルのみを描画

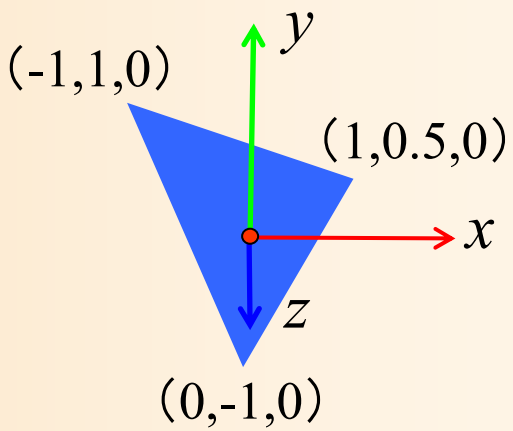




プログラムの例

- １枚の三角形を描画するプログラムの例
  - 各頂点の頂点座標、法線、色を指定して描画

```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );
        glVertex3f( 0.0,-1.0, 0.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```



座標変換

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト

表面の素材の表現

動きのデータの生成

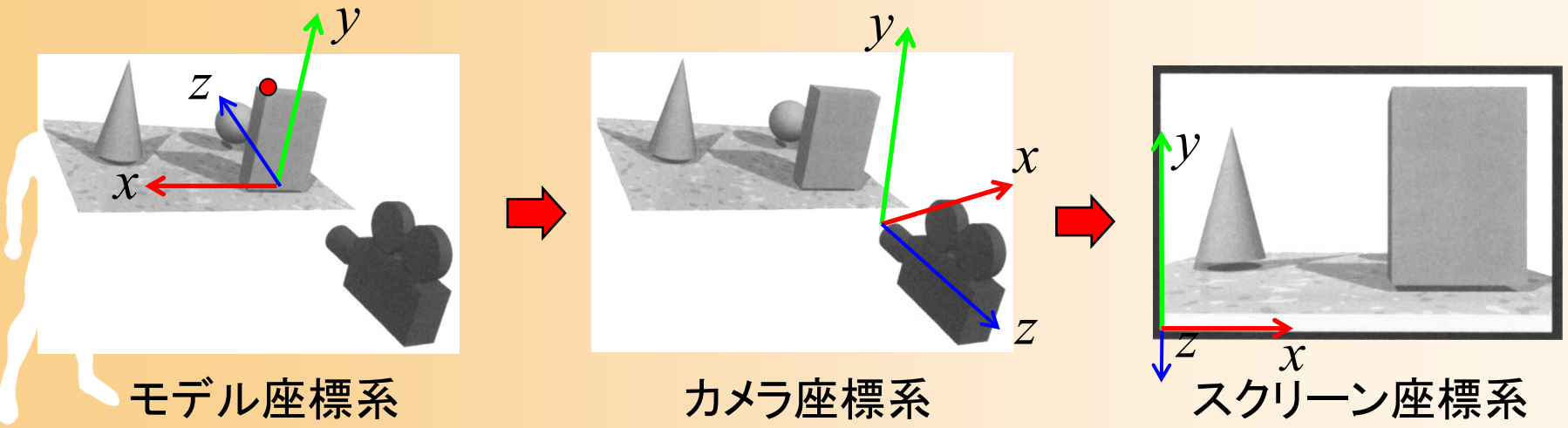
光源

光の効果の表現

カメラから見える画像を計算

座標変換

- 座標変換（Transformation）
  - 行列演算を用いて、ある座標系から、別の座標系に、頂点座標やベクトルを変換する技術
    - カメラから見た画面を描画するためには、モデルの頂点座標をカメラ座標系（最終的にはスクリーン座標系）に変換する必要がある



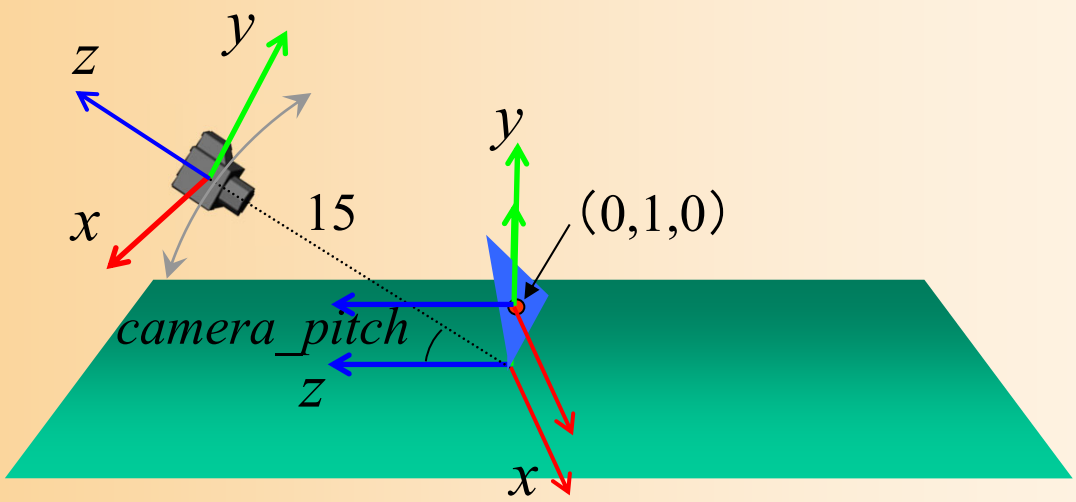
同次座標変換

- 同次座標変換によるアフィン変換
  - ４×４行列の演算によって、３次元空間における回転・平行移動・拡大縮小などのアフィン変換を計算
  - 同次座標系
    - （*x*, *y*, *z*, w）の４次元座標値によって扱う
    - ３次元座標値は（x/w, y/w, z/w）で計算（通常は w = 1）

$$\begin{pmatrix} R_{00}S_x & R_{01} & R_{02} & T_x \\ R_{10} & R_{11}S_y & R_{12} & T_y \\ R_{20} & R_{21} & R_{22}S_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

プログラムの例

- カメラや物体の配置に応じて変換行列を設定



ポリゴンを基準とする座標系での頂点座標

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

カメラから見た頂点座標（描画に使う頂点座標）

プログラムの例

- カメラや物体の配置に応じて変換行列を設定

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

```
// 変換行列を設定（ワールド座標系→カメラ座標系）
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( 0.0,  0.0, - 15.0 );
glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );
glTranslatef( 0.0, 1.0, 0.0 );
```

シェーディング

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト

表面の素材の表現

動きのデータの生成

光源

カメラから見える画像を計算

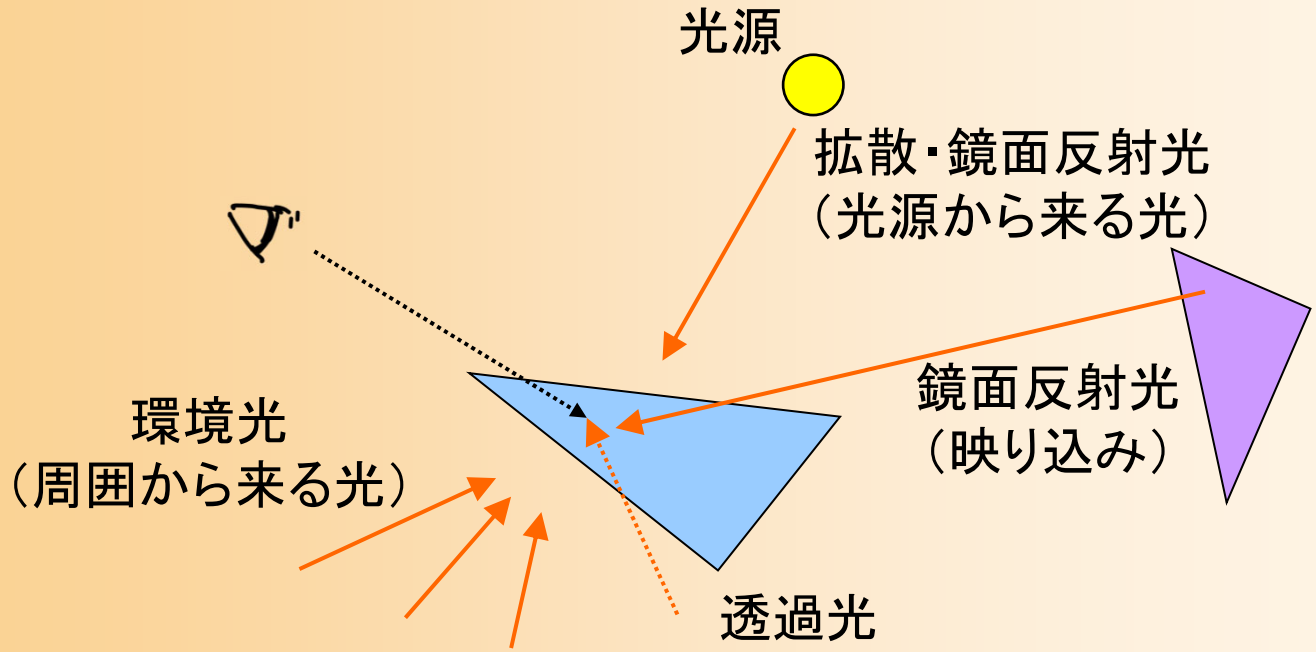
光の効果の表現

シェーディング

- シェーディング（Shading）
  - 光による効果を考慮して、物体を描く色を決めるための技術
    - 現実世界では、同じ素材の物体でも、光の当たり方によって見え方は異なる
    - コンピュータグラフィックスでも、このような効果を再現する必要がある

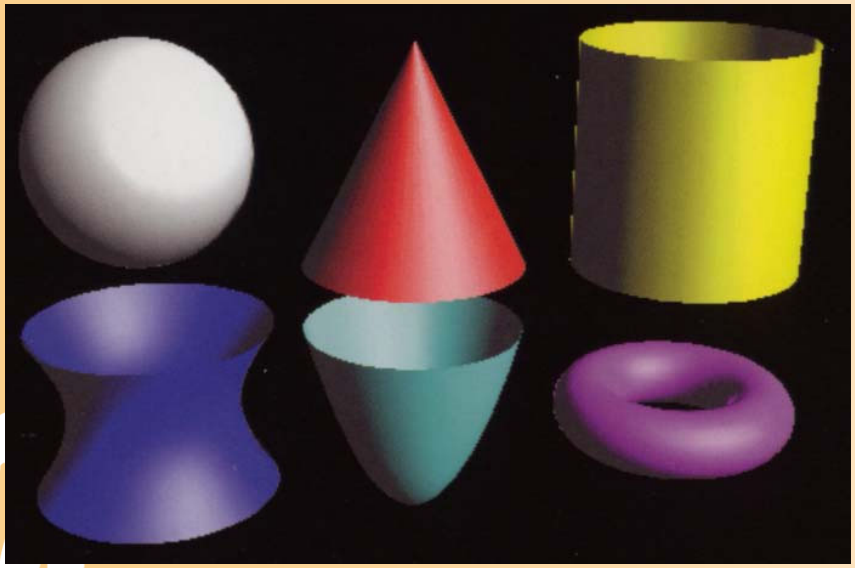
光のモデル

- 光を影響をいくつかの要素に分けて計算
  - 局所照明（光源からの拡散・鏡面反射光）
  - 大域照明（環境光、映り込み、透過光）



大域照明の効果の例

- 大域照明を考慮して描画することで、より写実的な画像を得ることができる



映り込み（大域照明）を考慮
基礎と応用 図8.9

環境光（大域照明）を考慮
基礎と応用 図9.1, 9.2

大域照明の効果の例

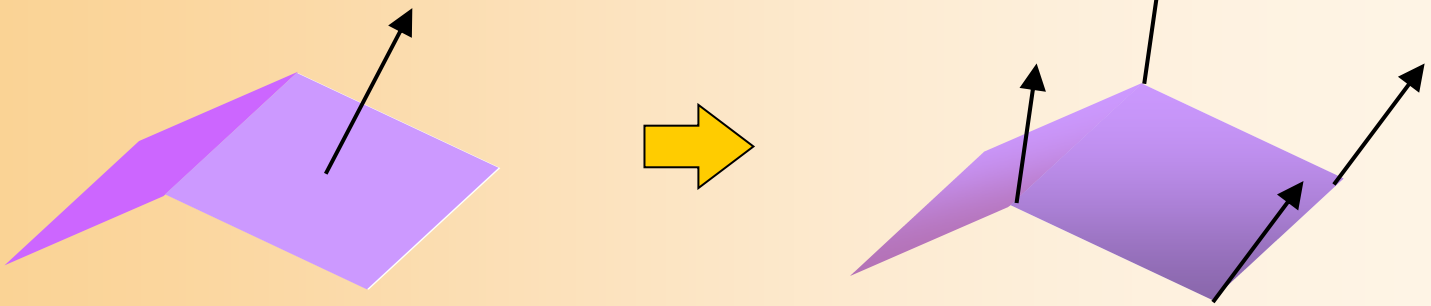
- 大域照明を考慮して描画することで、より写実的な画像を得ることができる

映り込み（大域照明）を考慮
基礎と応用 図8.9

環境光（大域照明）を考慮
基礎と応用 図9.1, 9.2

スムーズシェーディング

- 法線を変化させることで、ポリゴンモデルの表面で滑らかに描画
  - 局所照明モデルでは、法線ベクトルにもとづいて色が決まるため、各面は同じ色で描画される
    - 面の境界（辺）で色が大きく変わる
  - 頂点の法線にもとづいて、各ピクセルの色を計算することで、色の変化を滑らかにできる



プログラムの例

- 素材の色や光源の位置・色を設定すると、自動的に局所照明にもとづく色を計算

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );
glEnable( GL_LIGHT0 );
glEnalbe( GL_LIGHTING );
```

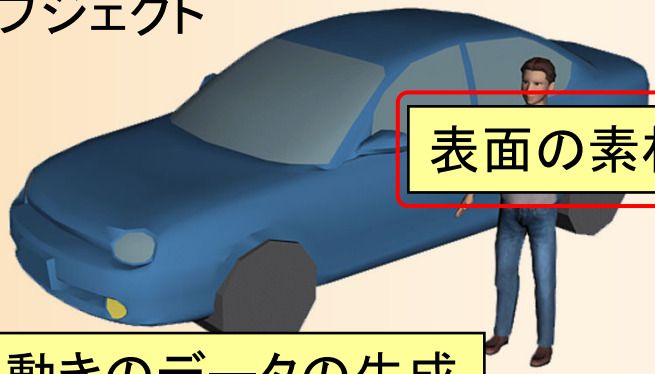
マッピング

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト



表面の素材の表現

動きのデータの生成

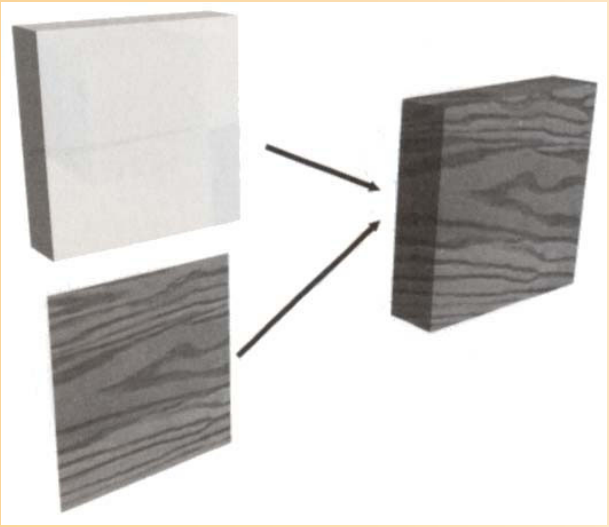
光源

カメラから見える画像を計算

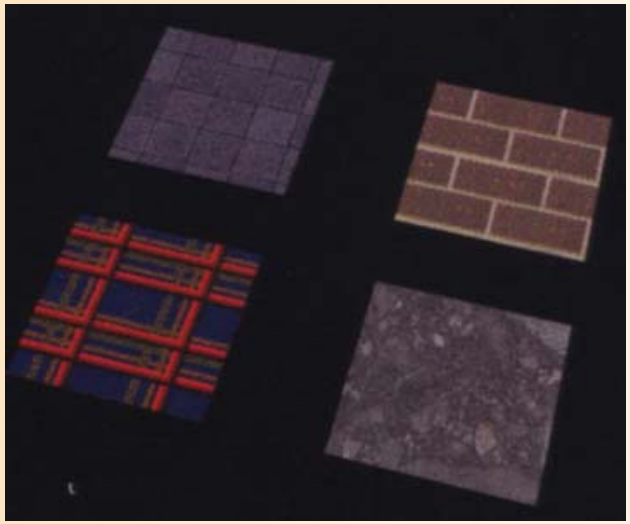
光の効果の表現

マッピング

- マッピング（Mapping）
  - 物体を描画する時に、表面に画像を貼り付けて描画する技術
  - 複雑な形状データを作成することなく、細かい模様などを表現できる



基礎知識 図3-19



基礎と応用
図5.2

高度なマッピング

- 凹凸のマッピング（バンプマッピング）

基礎と応用 図5.9

- 周囲の風景のマッピング（環境マッピング）

CG制作独習事典p.17

アニメーション

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト

表面の素材の表現

動きのデータの生成

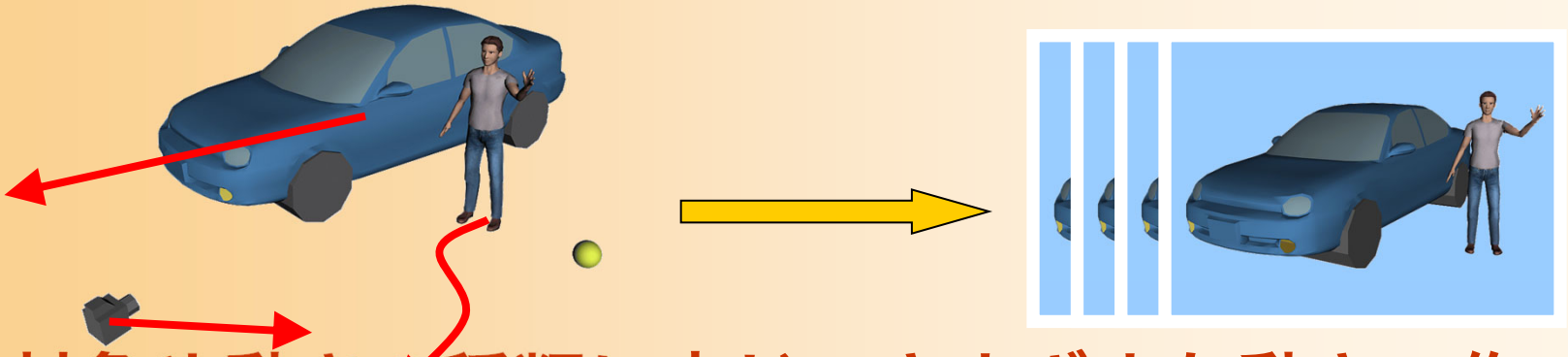
光源

カメラから見える画像を計算

光の効果の表現

アニメーション

- 動きのデータをもとに、アニメーションを生成



- 対象や動きの種類に応じてさまざまな動きの作成方法がある
  - キーフレームアニメーション、物理シミュレーション、モーションキャプチャ、など

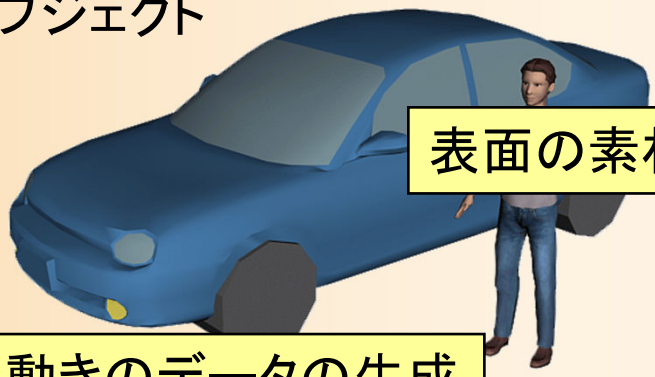
３次元グラフィックスの要素技術

- モデリング
- レンダリング
- 座標変換
- シェーディング
- マッピング
- アニメーション

オブジェクトの作成方法

オブジェクトの形状表現

オブジェクト



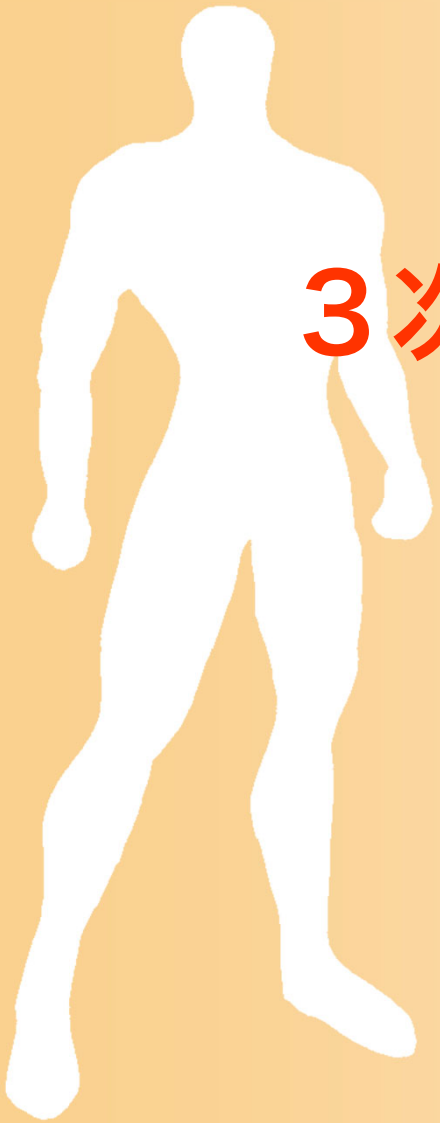
表面の素材の表現

動きのデータの生成

光源

光の効果の表現

カメラから見える画像を計算



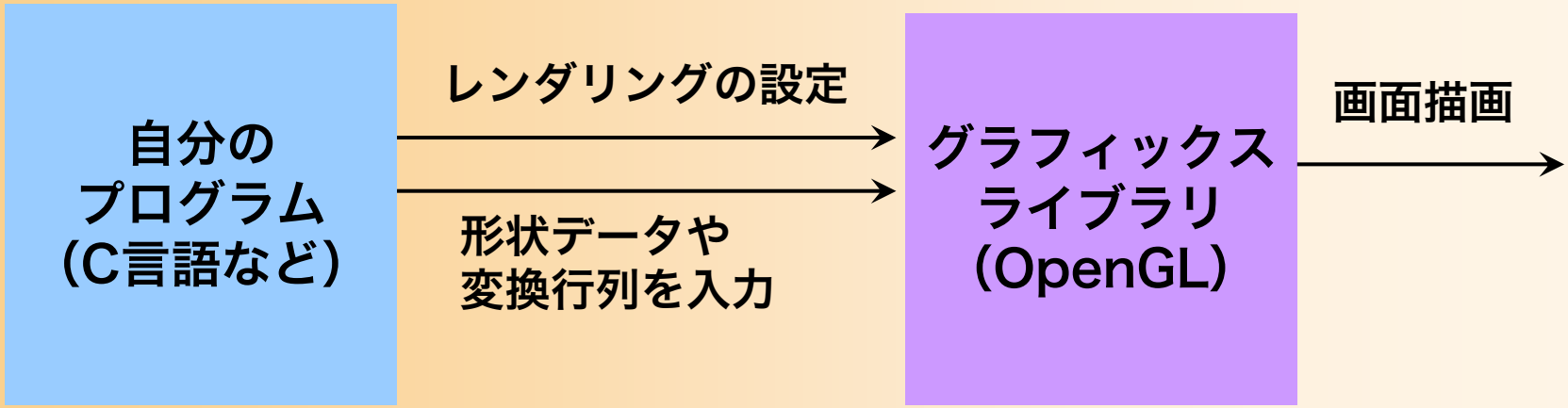
３次元グラフィックスのプログラミング

３次元グラフィックス・プログラミング

- ここまでに説明した技術を実現するようなプログラムを作成することで、３次元グラフィックスを描画できる
  - 全てを自分で実現しようとすると、非常に多くのプログラムを書く必要がある
  - 現在は、OpenGL のような、３次元グラフィックスライブラリが存在するので、これらのライブラリを利用することで、３次元グラフィックスを扱うプログラムを、比較的簡単に作成できる

グラフィックスライブラリの利用

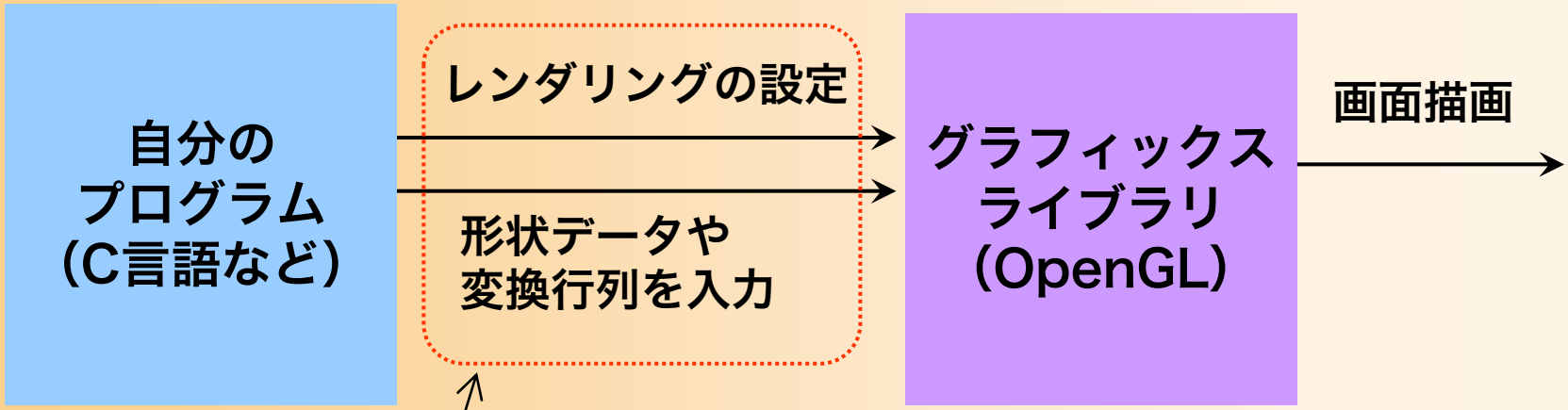
- 自分のプログラム と OpenGL の関係



レンダリング（＋座標変換、
シェーディング、マッピング）
などの処理を行ってくれる

グラフィックスライブラリの利用

- 自分のプログラム と OpenGL の関係



最低限、これらの方法だけ学べば、プログラムを作れる

これらの処理は、自分でプログラムを作る必要はないが、仕組みは理解しておく必要がある

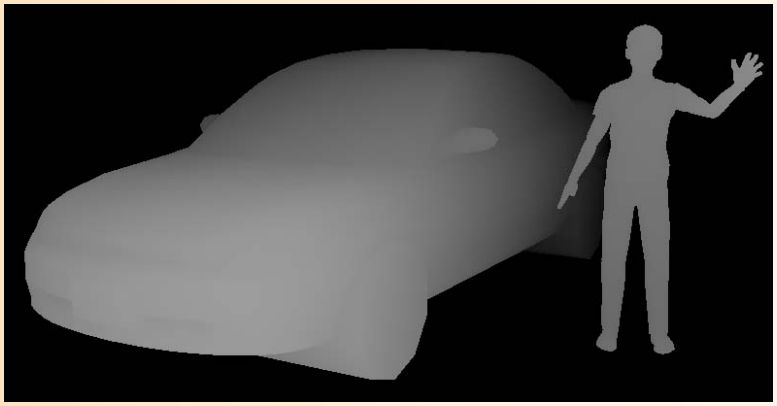
レンダリング（＋座標変換、シェーディング、マッピング）などの処理を行ってくれる

グラフィックスライブラリの機能

- 基本的なレンダリングの機能を提供
  - ポリゴンモデルによるモデリング
  - Ｚバッファ法によるレンダリング
  - スクリーン座標系への座標変換の適用
  - 局所照明モデルによるシェーディング
  - テクスチャマッピングの適用
- プログラマブルシェーダ（GPUプログラミング）により、座標変換やシェーディングの処理を拡張可能（本講義では説明は省略）

Ｚバッファ法（復習）

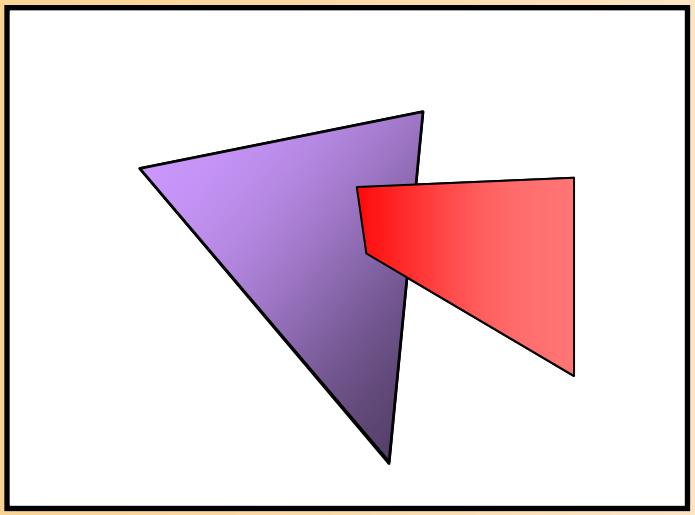
- 画像とは別に、それぞれのピクセルの奥行き情報であるZバッファを持つ
- Ｚバッファを使うことで隠面消去を実現
  - すでに書かれているピクセルのＺ座標と比較して、手前にある時のみピクセルを描画

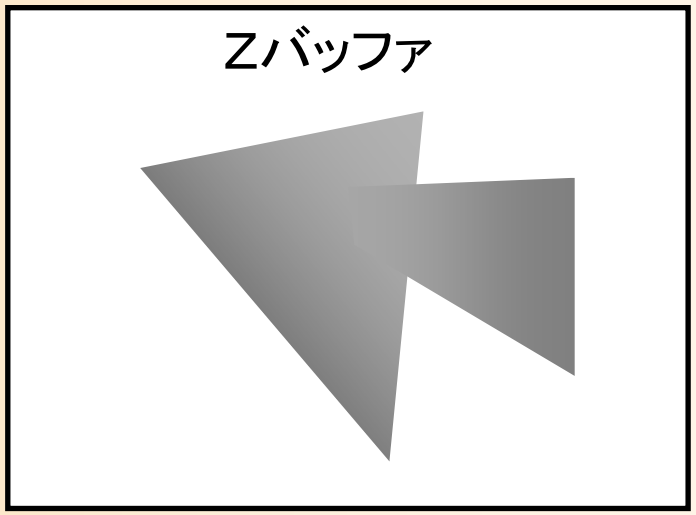


Zバッファの値（手前にあるほど明るく描画）

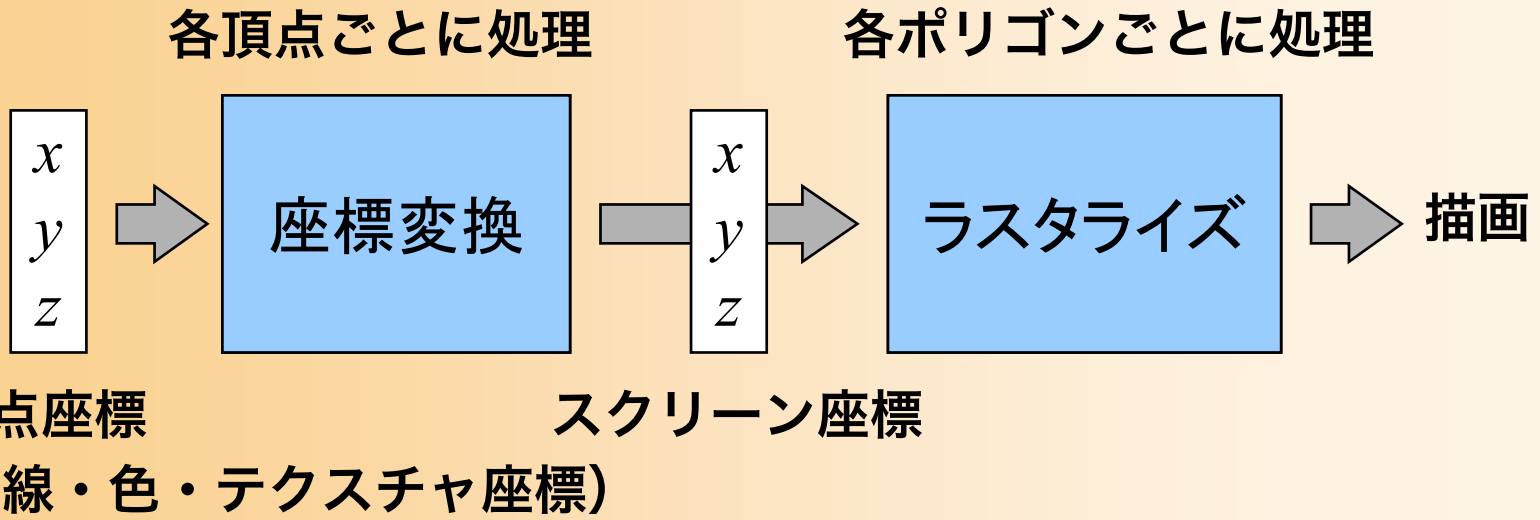
Ｚバッファ法による隠面消去（復習）

- Ｚバッファ法による面の描画
  - 面を描画するとき、各ピクセルの奥行き値（カメラからの距離）を計算して、Ｚバッファに描画
  - 同じ場所に別の面を描画するときは、すでに描画されている面より手前のピクセルのみを描画



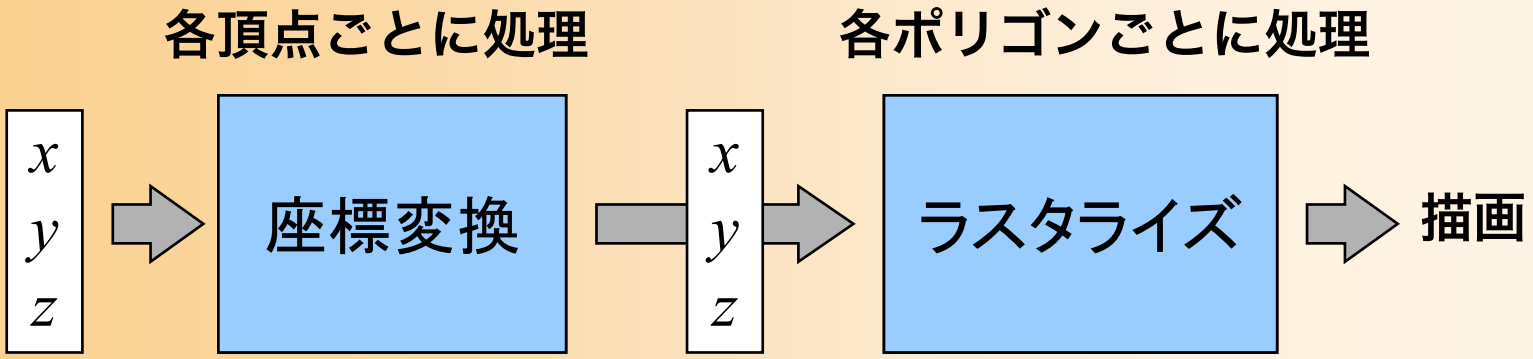


レンダリング・パイプライン



- レンダリング・パイプライン（ビューイング・パイプライン、グラフィックス・パイプライン）
  - 入力されたデータを、流れ作業（パイプライン）で処理し、順次、画面に描画
  - ポリゴンのデータ（頂点データの配列）を入力
  - いくつかの処理を経て、画面上に描画される

入出力の例（サンプルプログラム）

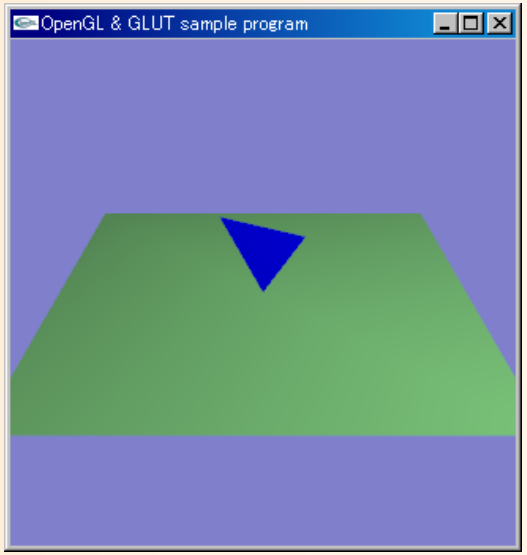


OpenGLにポリゴンの頂点情報を入力

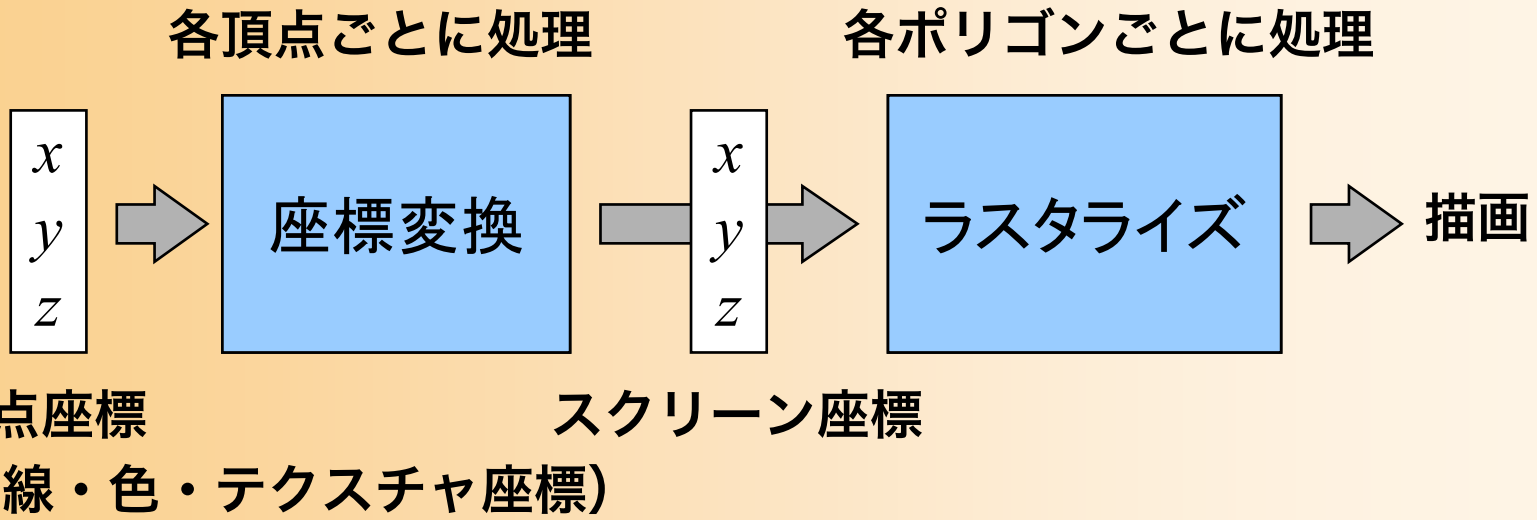
```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );
        glVertex3f( 0.0,-1.0, 0.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```

座標変換 &
ラスタライズ

ポリゴンが描画される



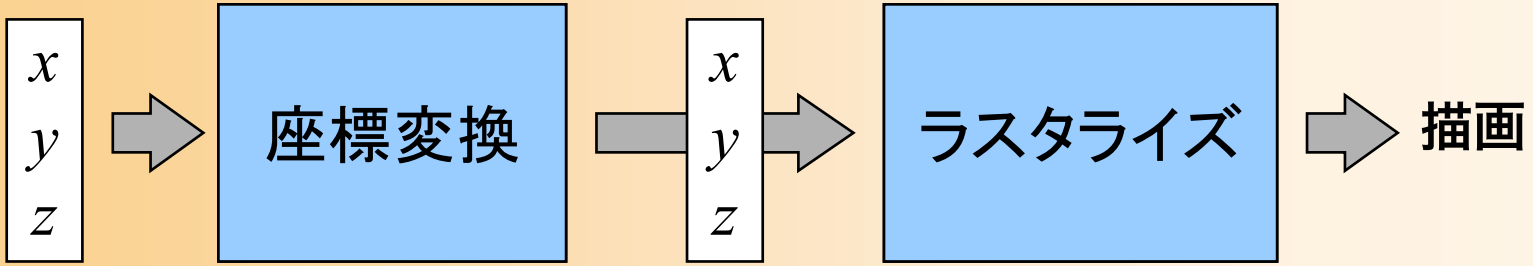
処理の流れ



- レンダリング時のデータ処理の流れ
  1. ポリゴンを構成する頂点の座標、法線、色、テクスチャ座標などを入力
  2. スクリーン座標に変換（座標変換）
  3. ポリゴンをスクリーン上に描画（ラスタライズ）

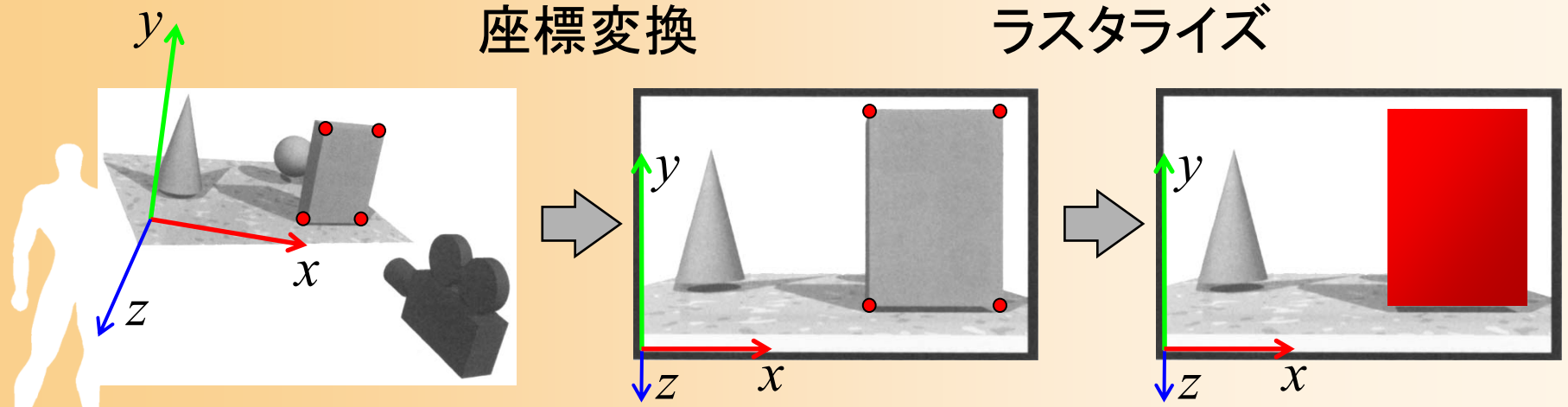
処理の流れ

各頂点ごとに処理　　各ポリゴンごとに処理



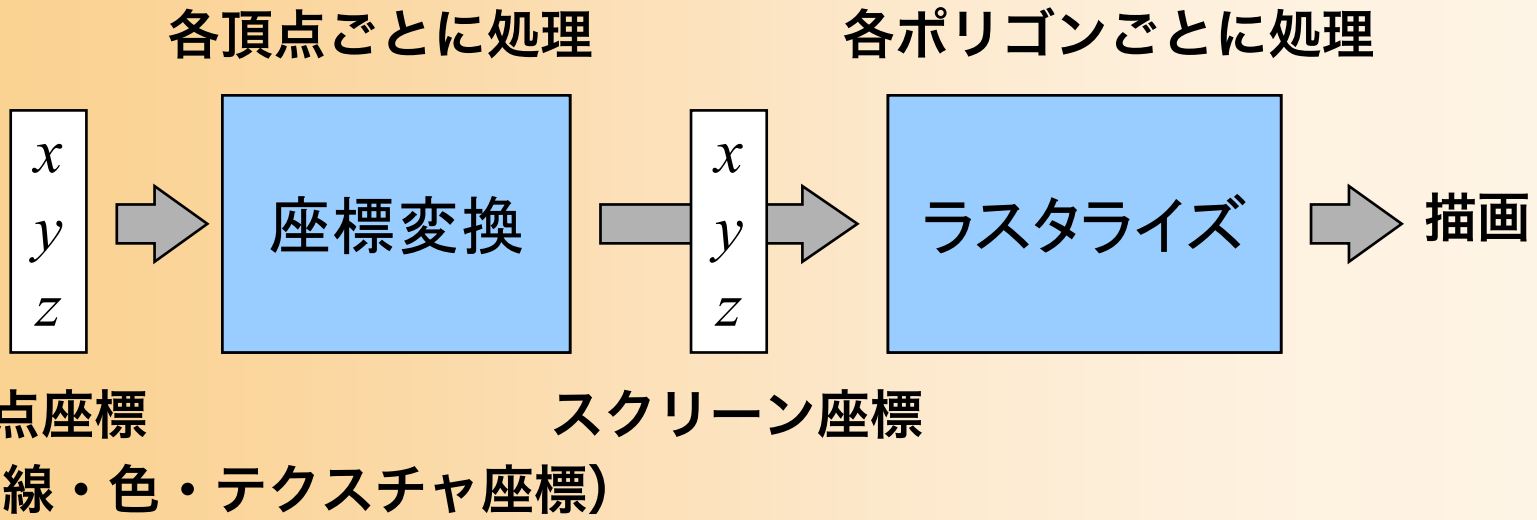
頂点座標
(法線・色・テクスチャ座標)

スクリーン座標



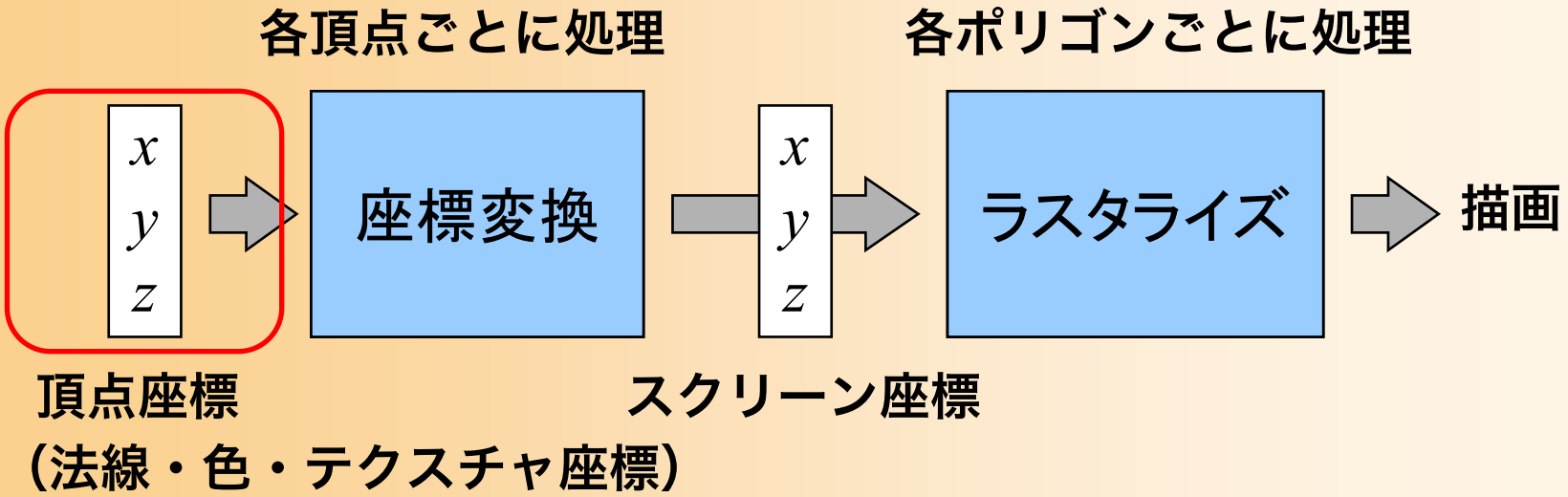
教科書 基礎知識 図2-21

描画前に行なう設定



- カメラの位置・向き（変換行列）の設定
- 光源の情報（位置・向き・色など）を設定
- テクスチャの情報を設定
- これらの情報は次に更新されるまで記録される

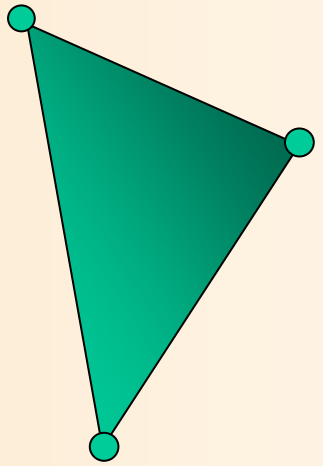
描画データの入力



- 物体の情報を入力
  - ポリゴンを構成する頂点の座標・法線・色・テクスチャ座標などを入力
- 表面の素材などを途中で変える場合は、適宜設定を変更

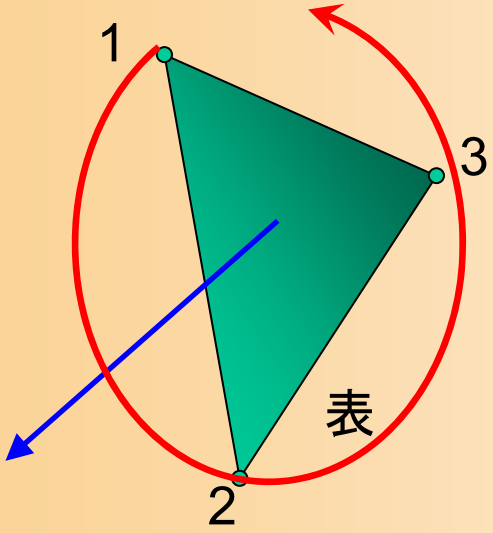
ポリゴンデータ

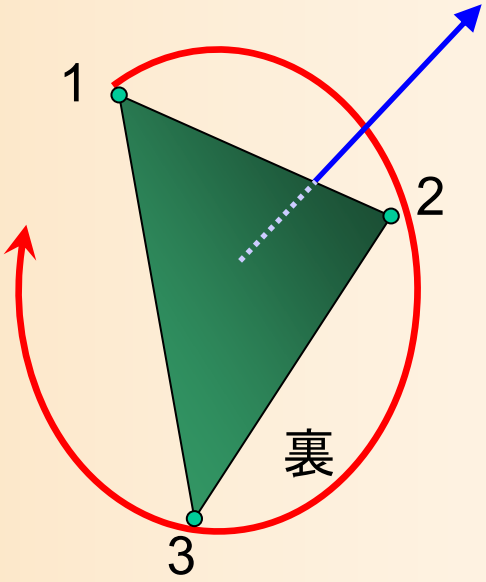
- ポリゴンの持つ情報
  - 各頂点の情報
    - 座標
    - 法線
    - 色
    - テクスチャ座標
  - 面の向き
    - 頂点の順番によって面の向きを表す
    - 視点と反対向きの面は描画しない（背面除去）
    - 法線と面の向きは別なので注意が必要



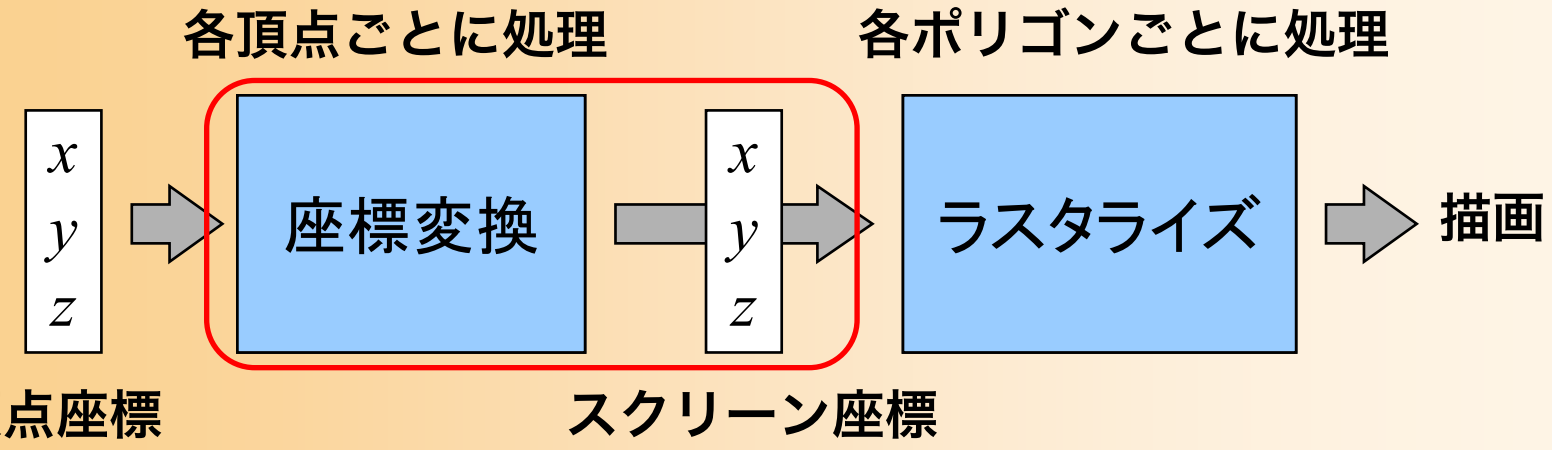
ポリゴンの向き

- 頂点の順番により、ポリゴンの向きを決定
  - 表から見て反時計回りの順序で頂点を与える
    - どちらの順序を表とするかは設定で変更できる
  - 視点と反対向きの面は描画しない（背面除去）
    - 頂点の順序を間違えると描画されないため要注意



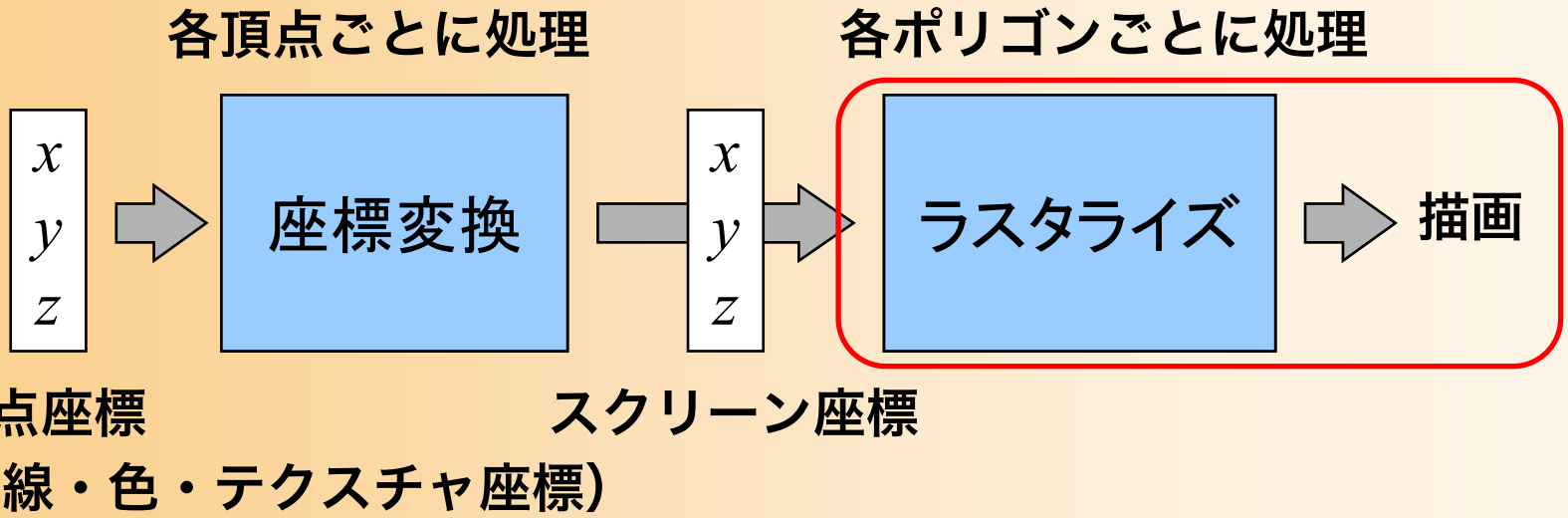


座標変換



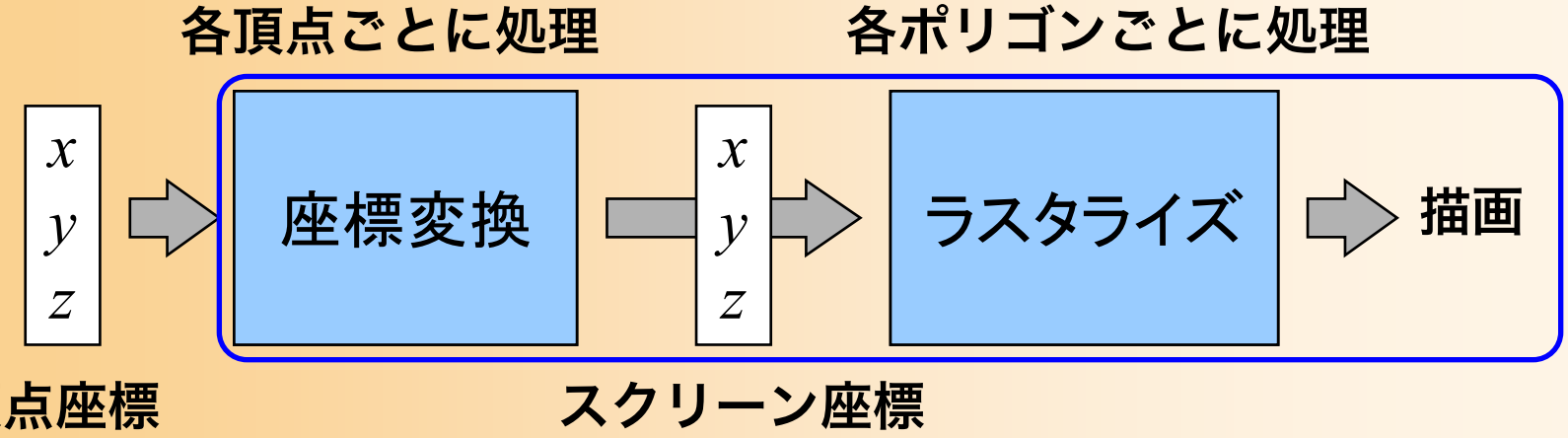
- 座標変換
  - 各頂点のスクリーン座標を計算
  - 法線と光源情報から、頂点の色を計算
  - 面の向きをもとに背面除去、視界外の面も除去

ラスタライズ



- ラスタライズ（ポリゴンを画面上に描画）
  - Zバッファ法による描画
  - スムーズシェーディング（グローシェーディング）
  - テクスチャマッピング

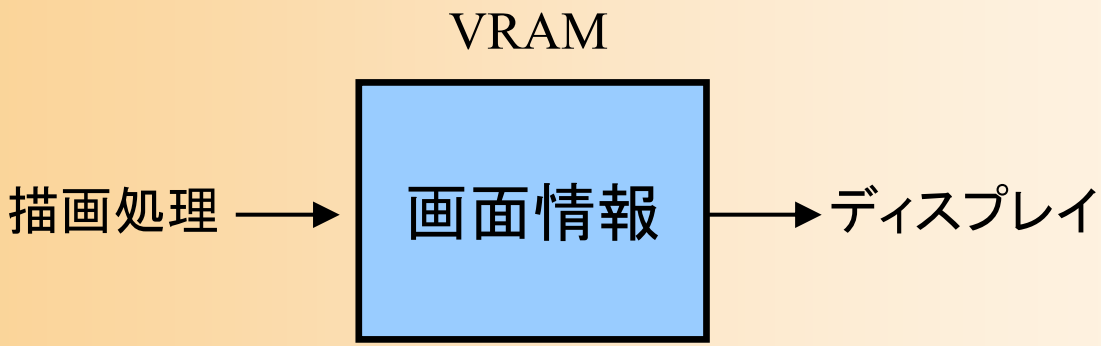
ハードウェアサポート



- ハードウェアによる処理
  - 最近のハードウェアでは、レンダリング・パイプラインの全ての計算を、グラフィックスハードウェア（GPU）で処理可能
  - GPU用のプログラムを記述することで、座標変換やラスタライズの処理方法を変更できる（VertexShader, PixelShader）

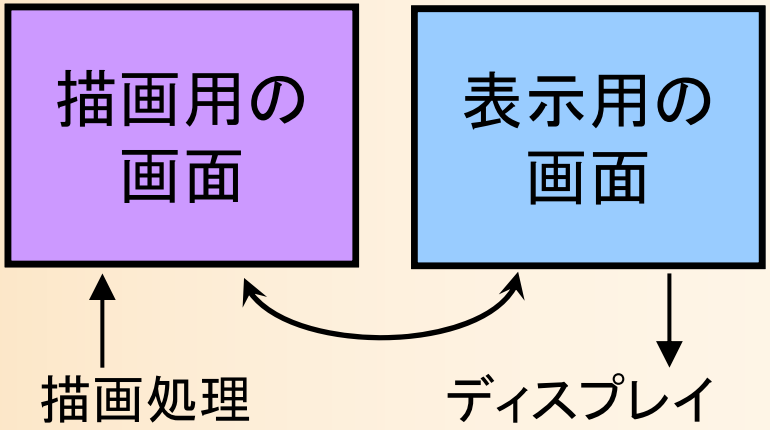
ダブルバッファリング

- 画面表示の仕組み
  - ビデオメモリ（VRAM）上の画面データをディスプレイ上に表示
  - 描画途中の画面を表示するとちらついてしまう
    - 描画量が少ない場合は垂直同期（VSYNC）中に描画すればちらつかない



ダブルバッファリング

- ２枚の画面を使用
  - 表示用
  - 描画用（＋Ｚバッファ）



- ダブルバッファリング
  - 描画用の画面に対して描画
  - 描画が完了したら、描画用の画面と表示用の画面を切り替える
    - 自分のプログラムから明示的にこの切り替えの処理を行う必要がある

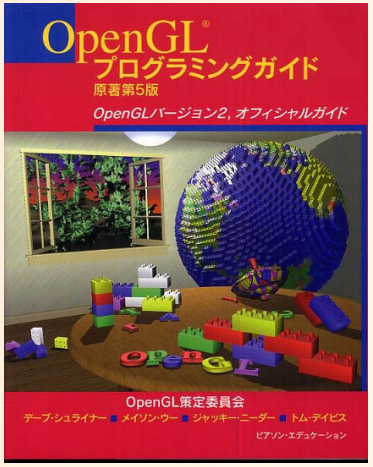
本日の内容

- ガイダンス
- コンピュータグラフィックスの概要と応用
- 復習：３次元グラフィックスの要素技術
- 復習：３次元グラフィックスのプログラミング
- 復習：OpenGL&GLUT プログラミング
- 復習：OpenGL&GLUT サンプルプログラム

OpenGL&GLUT プログラミング

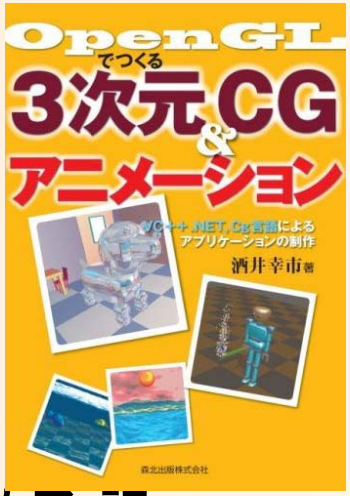
参考書

- OpenGLの定番の参考書
  - OpenGLプログラミングガイド（赤本），12,000円
  - OpenGLリファレンスマニュアル（青本），8,300円
    - ピアソン・エデュケーション出版
    - 中・上級者向け



参考書（続き）

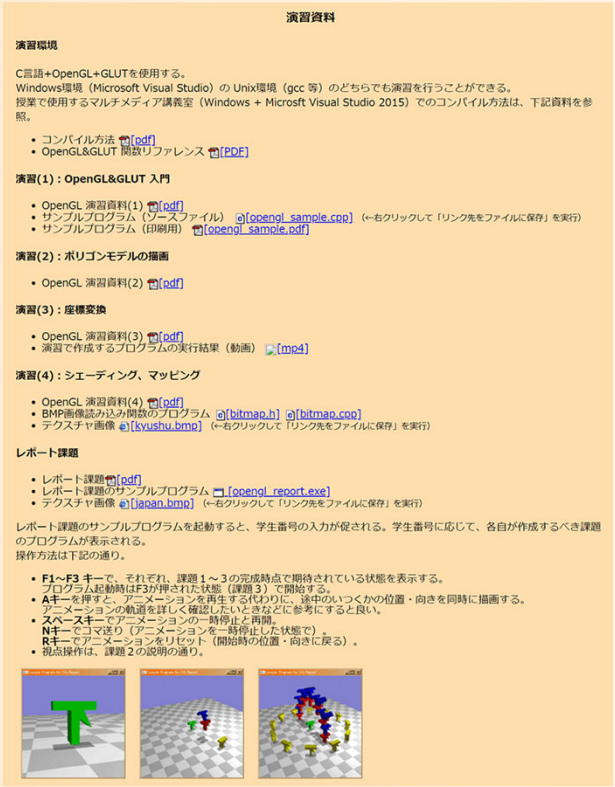
- 他にもOpenGLの入門書は多数ある
  - OpenGLでつくる 3次元CG & アニメーション （3600円）
    - 酒井 幸市 著
    - OpenGL・GLUTの使い方 ＋ 最新技術
    - 興味がある人は、買ってみると良い
  - OpenGL入門 （3,000円）
    - エドワード・エンジェル 著、 滝沢 徹・牧野 祐子 訳
    - ピアソン・エデュケーション出版
    - OpenGL・GLUTの使い方



プログラミング演習資料

- 学部の授業の演習課題・レポート課題
  - 基礎の理解が不十分な人は、各自で取り組む
  - システム創成情報工学科「コンピュータグラフィックスＳ」の講義・演習資料

    http://www.cg.ces.kyutech.ac.jp/lecture/cg/
  - 演習課題
    1. ポリゴンモデルの描画
    2. 座標変換によるアニメーション
    3. テクスチャマッピング
  - レポート課題



演習資料

演習環境

C言語+OpenGL+GLUTを使用する。
Windows環境（Microsoft Visual Studio）の Unix環境（gcc 等）のどちらでも演習を行うことができる。
授業で使用するマルチメディア講義室（Windows + Microsft Visual Studio 2015）でのコンパイル方法は、下記資料を参照。

- コンパイル方法 [pdf]
- OpenGL&GLUT 関数リファレンス [PDF]

演習(1)：OpenGL&GLUT 入門

- OpenGL 演習資料(1) [pdf]
- サンプルプログラム（ソースファイル） [opengl_sample.cpp] （←右クリックして「リンク先をファイルに保存」を実行）
- サンプルプログラム（印刷用） [opengl_sample.pdf]

演習(2)：ポリゴンモデルの描画

- OpenGL 演習資料(2) [pdf]

演習(3)：座標変換

- OpenGL 演習資料(3) [pdf]
- 演習で作成するプログラムの実行結果（動画） [mp4]

演習(4)：シェーディング、マッピング

- OpenGL 演習資料(4) [pdf]
- BMP画像読み込み関数のプログラム [bitmap.h] [bitmap.cpp]
- テクスチャ画像 [kyushu.bmp] （←右クリックして「リンク先をファイルに保存」を実行）

レポート課題

- レポート課題 [pdf]
- レポート課題のサンプルプログラム [opengl_report.exe]
- テクスチャ画像 [japan.bmp] （←右クリックして「リンク先をファイルに保存」を実行）

レポート課題のサンプルプログラムを起動すると、学生番号の入力が促される。学生番号に応じて、各自が作成するべき課題のプログラムが表示される。
操作方法は下記の通り。

- **F1～F3 キー**で、それぞれ、課題１～３の完成時点で期待されている状態を表示する。
  プログラム起動時はF3が押された状態（課題３）で開始する。
- **Aキー**を押すと、アニメーションを再生する代わりに、途中のいくつかの位置・向きを同時に描画する。
  アニメーションの軌道を詳しく確認したいときなどに参考にすると良い。
- **スペースキー**でアニメーションの一時停止と再開。
  Nキーでコマ送り（アニメーションを一時停止した状態で）。
  Rキーでアニメーションをリセット（開始時の位置・向きに戻る）。
- 視点操作は、課題２の説明の通り。

OpenGL & GLUT

- OpenGL
  - 現在、最も広く使われている３次元API
    - C言語を始め、さまざまな言語から使える
  - ポリゴンの描画、Ｚバッファなどの３次元描画に必要な機能を提供
  - ウィンドウ生成やマウス・キーボード入力などの処理の機能は持たない
    - これらは、OSやウィンドウシステム固有の機能なので、各環境に応じたAPIを使って記述する必要がある
    - 実装が大変、環境ごとに実装する必要がある

GLUT

- OpenGL Utility Toolkit （GLUT)
  - ウィンドウ生成やイベント処理などの環境依存の部分を共通化したライブラリ
    - OpenGL標準ではないがかなり広く普及している
  - 内部に各OS用のコードを含んでいるため、一度プログラムを作ればいろんな環境で動く
  - 機能が限定されている代わりに非常にシンプル
  - とりあえずOpenGLを使いたい場合に適している

DirectXとの比較

- DirectX
  - Windowsのみでしか動かない
  - Windowsと密接に関連している
    - WindowsやCOMなどの仕組みを理解する必要がある
    - プログラミングが必要以上に面倒
  - 最新のハードウェアの機能を使えるという利点もある
  - 他のマルチメディア機能も持っている
    - DirectSound, DirectPlay, DirectInput
  - 基本な考え方はOpenGLと同じ

Java3Dとの比較

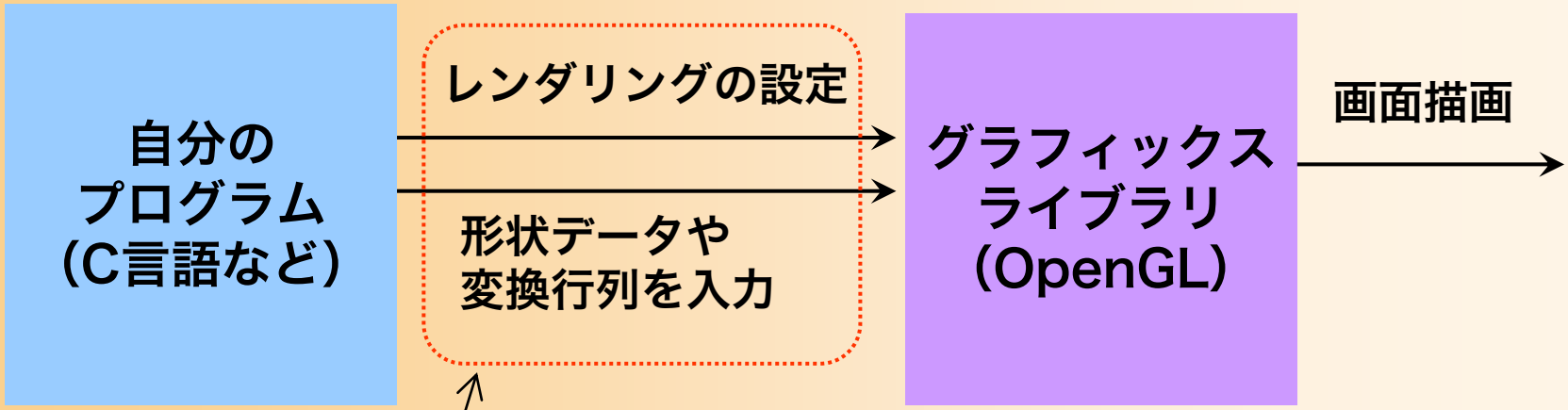
- Java3D
  - シーングラフ API （高レベルAPI）
    - カメラや物体などのシーンの階層構造を設定してやると、細かい描画は自動的に行ってくれる
    - 高機能で便利、ＣＧの原理をよく知らなくても使える
  - デメリット
    - 独自のライブラリなので、Java3Dの使い方だけ覚えても使い回しが利かない
    - クラスライブラリになっているので、本当にきちんと理解しようとすると全体像を把握する必要があり、大変

他のライブラリとの比較

- 携帯端末（iOS/Androidなど）用アプリ
  - OpenGL ES が採用されている
    - OpenGL のサブセット（機能限定版）
    - 描画処理の基本は OpenGL と基本的に同様
- ゲームエンジン
  - Unity, Unreal など
  - Java 3D 同様、シーン情報やアニメーションを設定すれば、細かい描画処理は自動的に行ってくれる

OpenGLの利用

- 自分のプログラム と OpenGL の関係



最低限、これらの方法だけ学べば、
プログラムを作れる

これらの処理は、自分でプログラ
ムを作る必要はないが、仕組みは
理解しておく必要がある

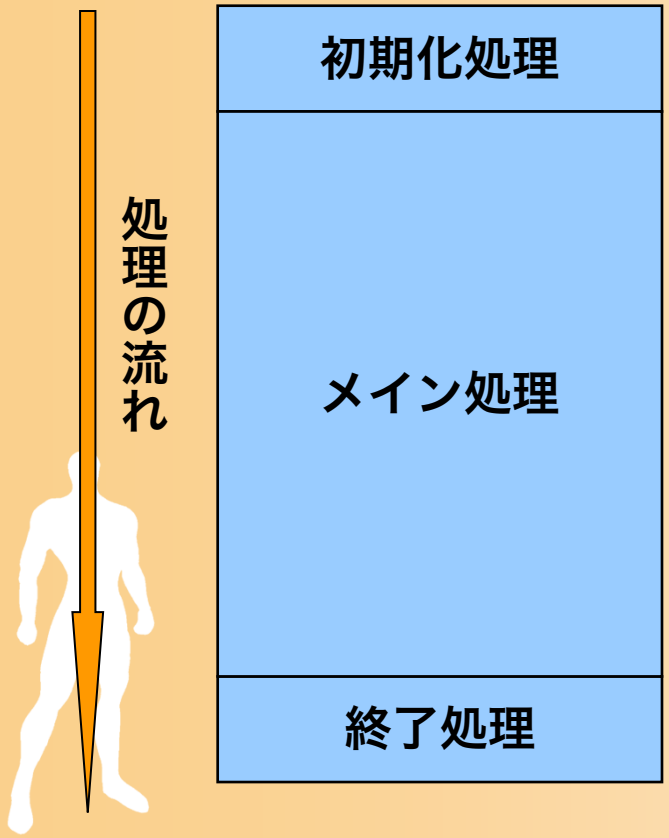
レンダリング（＋座標変換、
シェーディング、マッピング）
などの処理を行ってくれる

GLUTのイベントモデル

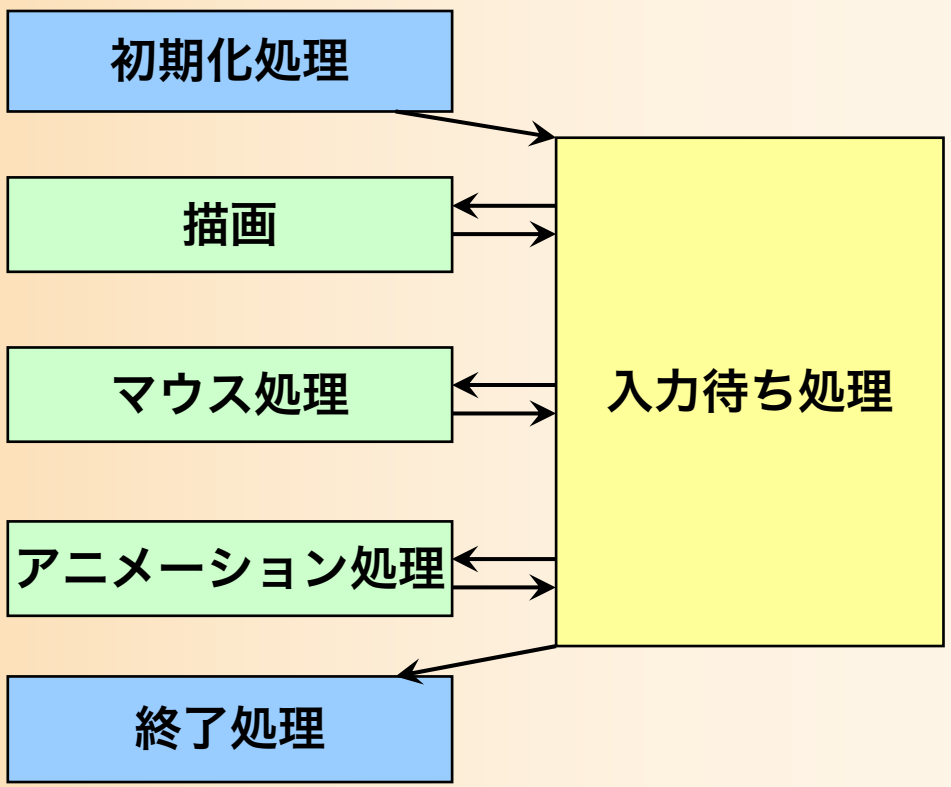
- ウィンドウシステムでのプログラミング
  - Windows や X Window などの一般的なウィンドウシステム
  - ウィンドウ管理やマウス操作などはシステムがまとめて処理するため、ユーザプログラムは扱わない
  - ユーザプログラムは初期化処理を行った後は処理をウィンドウシステムに移す
  - ウィンドウシステムは、画面の再描画やマウスの操作などのイベントが起こるたびにユーザプログラムに処理を一時的に戻す

イベントドリブン型プログラム

コンソール・プログラム



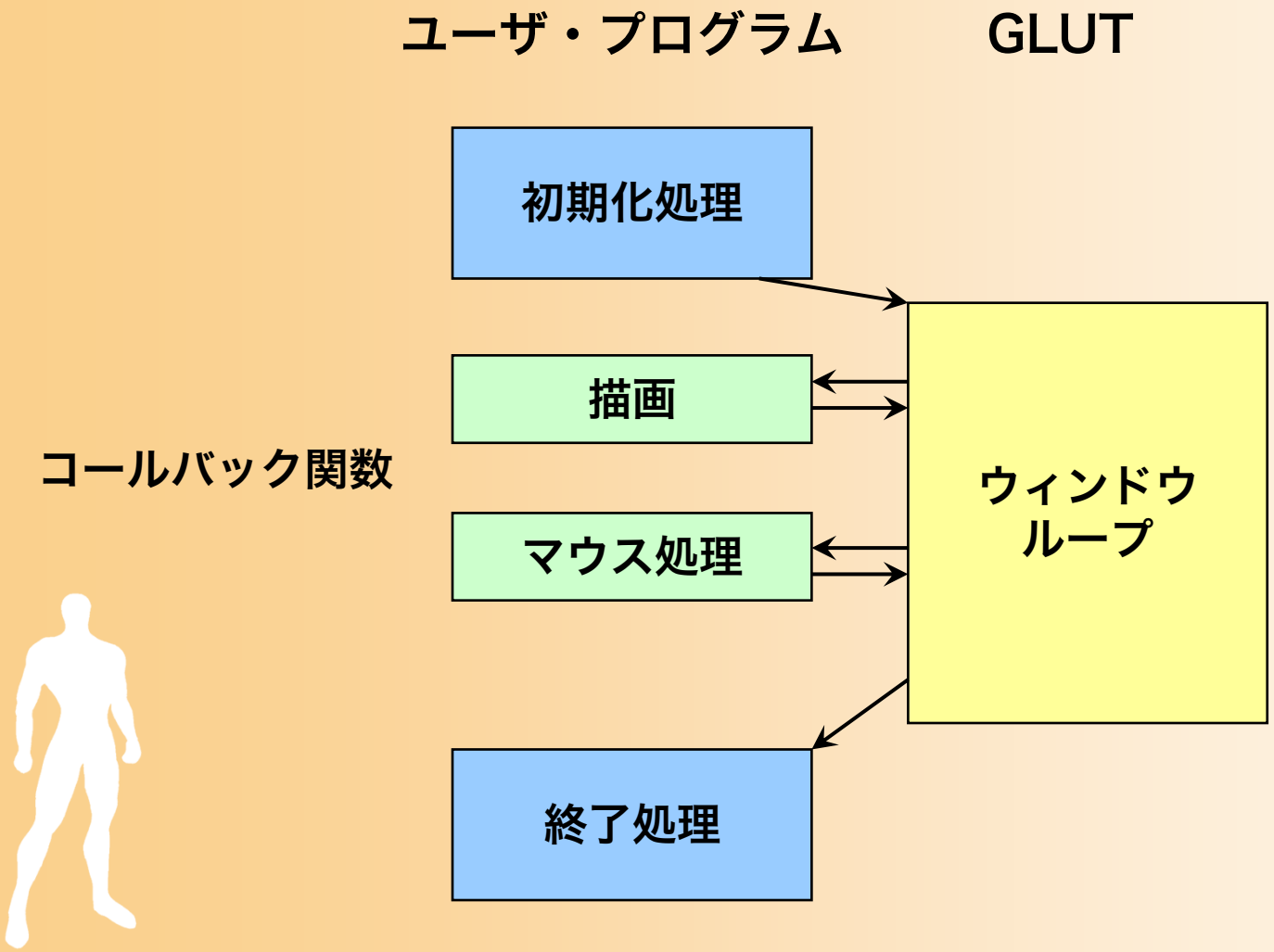
ウィンドウ・プログラム（イベントドリブン）



GLUTのイベントモデル

- イベントループとコールバック
  - イベントが起こった時にそのイベントを処理する関数をあらかじめ登録しておく
  - プログラムは初期化が終わったら、GLUTに処理を移す
  - マウス操作などのイベントが起こったらあらかじめ登録した関数が呼ばれる（コールバック）

GLUTのイベントモデル



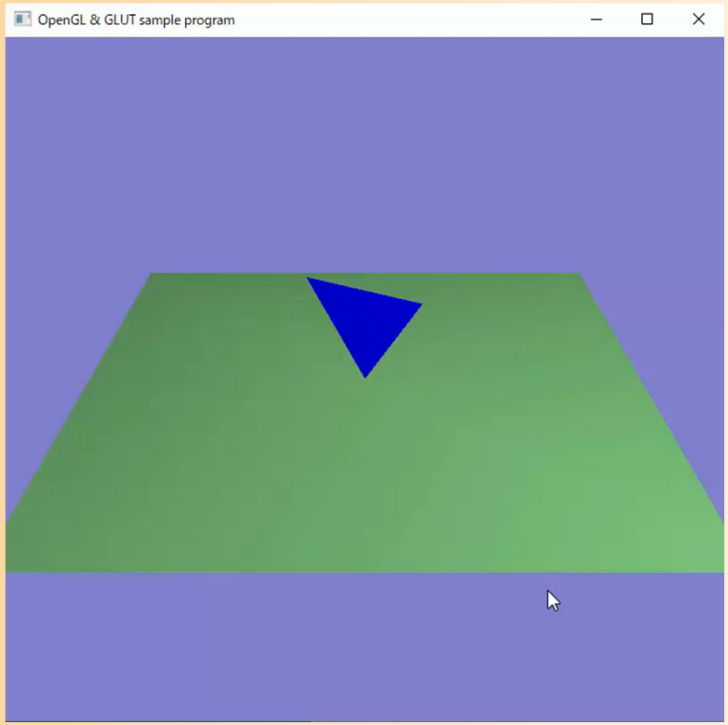
GLUTのコールバック関数の種類

- 描画コールバック関数
  - 描画が必要な時に呼ばれる
- サイズ変更コールバック関数
  - ウィンドウサイズ変更時に呼ばれる
- マウスクリック・コールバック関数
  - マウスのボタンが押されたとき、離されたときに呼ばれる
- マウスドラッグ・コールバック関数
  - マウスがウィンドウ上でドラッグされたときに呼ばれる
- キーボード・コールバック関数
  - キーボードのキーが押されたときに呼ばれる
- アイドル・コールバック関数
  - 処理が空いた時に定期的に呼ばれる

OpenGL&GLUT
サンプルプログラム

サンプルプログラム

- opengl_sample.cpp
  - 地面と１枚の青い三角形が表示される
  - マウスの右ボタンドラッグで、視点を上下に回転



サンプルプログラム

- opengl_sample.cpp

```
1 //
2 //  コンピュータグラフィックスＳ
3 //  OpenGLによる３次元グラフィックス演習 サンプルプログラム
4 //
5
6
7 // 基本的なヘッダファイルのインクルード
8 #ifdef _WIN32
9     #include <windows.h>
10 #endif
11 #include <stdio.h>
12 #include <math.h>
13
14 // GLUTヘッダファイルのインクルード
15 #include <GL/glut.h>
16
17
18 // 視点操作のための変数
19 float  camera_pitch = -30.0;  // Ｘ軸を軸とするカメラの回転角度
20
21 // マウスのドラッグのための変数
22 int  drag_mouse_r = 0;  // 右ボタンをドラッグ中かどうかのフラグ（0:非ドラッグ中,1:ドラッグ中）
23 int  last_mouse_x;      // 最後に記録されたマウスカーソルのＸ座標
24 int  last_mouse_y;      // 最後に記録されたマウスカーソルのＹ座標
25
26
27 //
28 //  画面描画時に呼ばれるコールバック関数
29 //
30 void  display( void )
31 {
32     // 画面をクリア（ピクセルデータとＺバッファの両方をクリア）
33     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
34
35     // 変換行列を設定（ワールド座標系→カメラ座標系）
36     glMatrixMode( GL_MODELVIEW );
37     glLoadIdentity();
38     glTranslatef( 0.0,  0.0, - 15.0 );
39     glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );
40
41     // 光源位置を設定（モデルビュー行列の変更にあわせて再設定）
42     float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
43     glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
44
```

サンプルプログラムの解説

- ここでは、プログラム全体を眺めて、大まかに、各部分でどのような処理を行っているかを確認する
- 各自、実際にコンパイルをしてみて、動作を確認する

OpenGLの関数

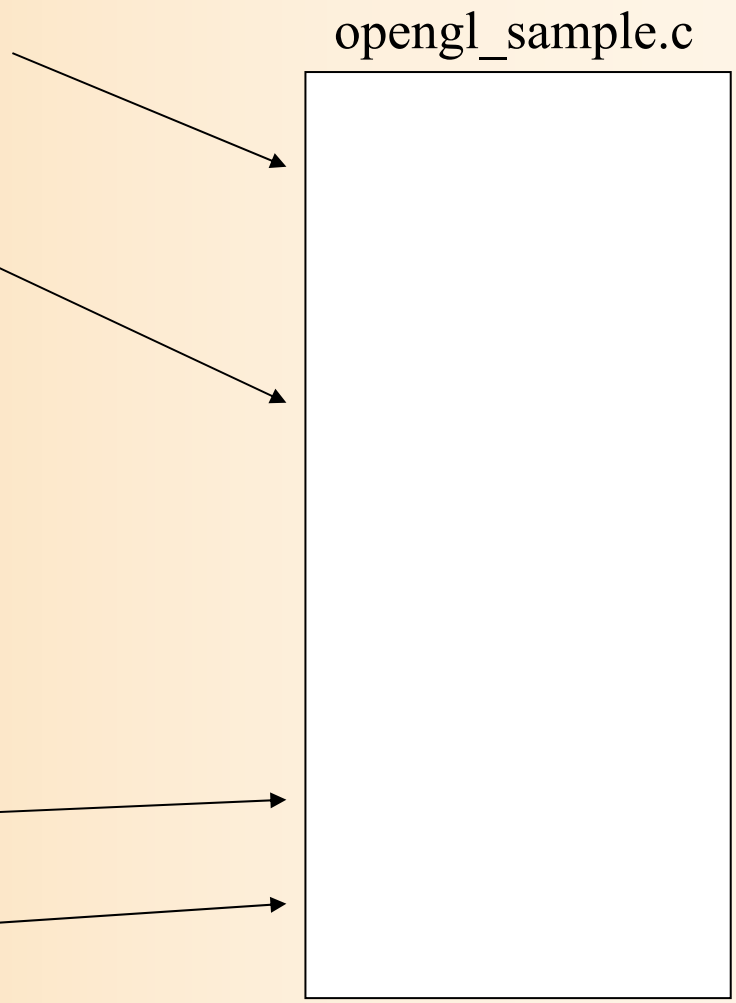
- gl～ で始まる関数
  - OpenGLの標準関数
- glu～ で始まる関数
  - OpenGL Utility Library の関数
  - OpenGLの関数を内部で呼んだり、引数を変換したりすることで、使いやすくした補助関数
- glut～ で始まる関数
  - GLUT（OpenGL Utility Toolkit）の関数
  - 正式にはOpenGL標準ではない

OpenGLの関数名

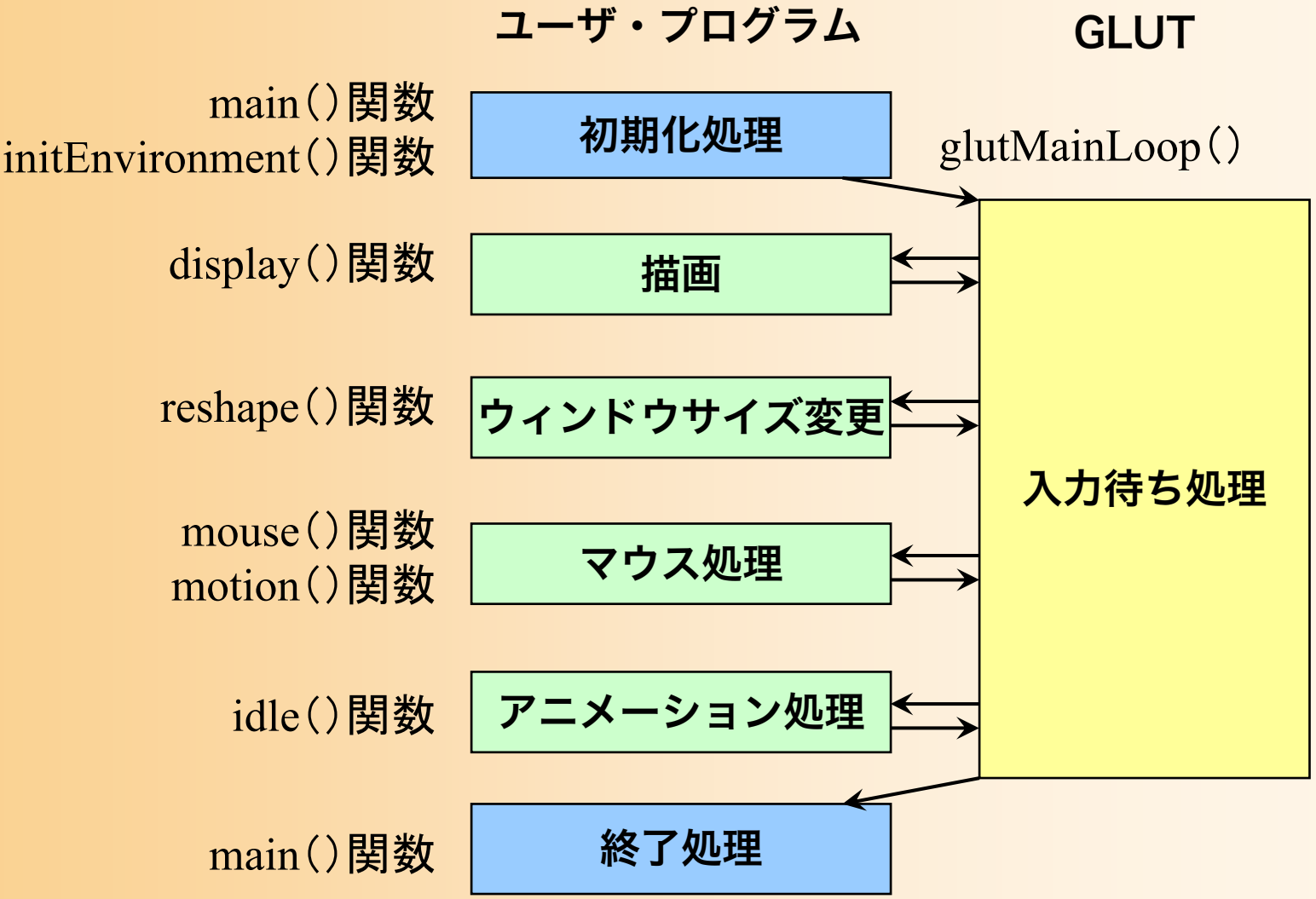
- 同じ機能で、微妙に違う名前の関数がある
  - 例： glVertex3f（*x, y, z*）, glVertex3d（*x, y, z*）
    - f は引数が float 型であることを表す
    - d は引数が double 型であることを表す
      - C言語なので、関数のオーバーロード（同じ名前で引数が異なる関数）はサポートしていない
    - 必要に応じて使い分ける

サンプルプログラムの構成

- グローバル変数の定義
- コールバック関数
  - display（）
  - reshape（）
  - mouse（）
  - motion（）
  - idle（）
- initEnvironment()
- main()



サンプルプログラムの構成

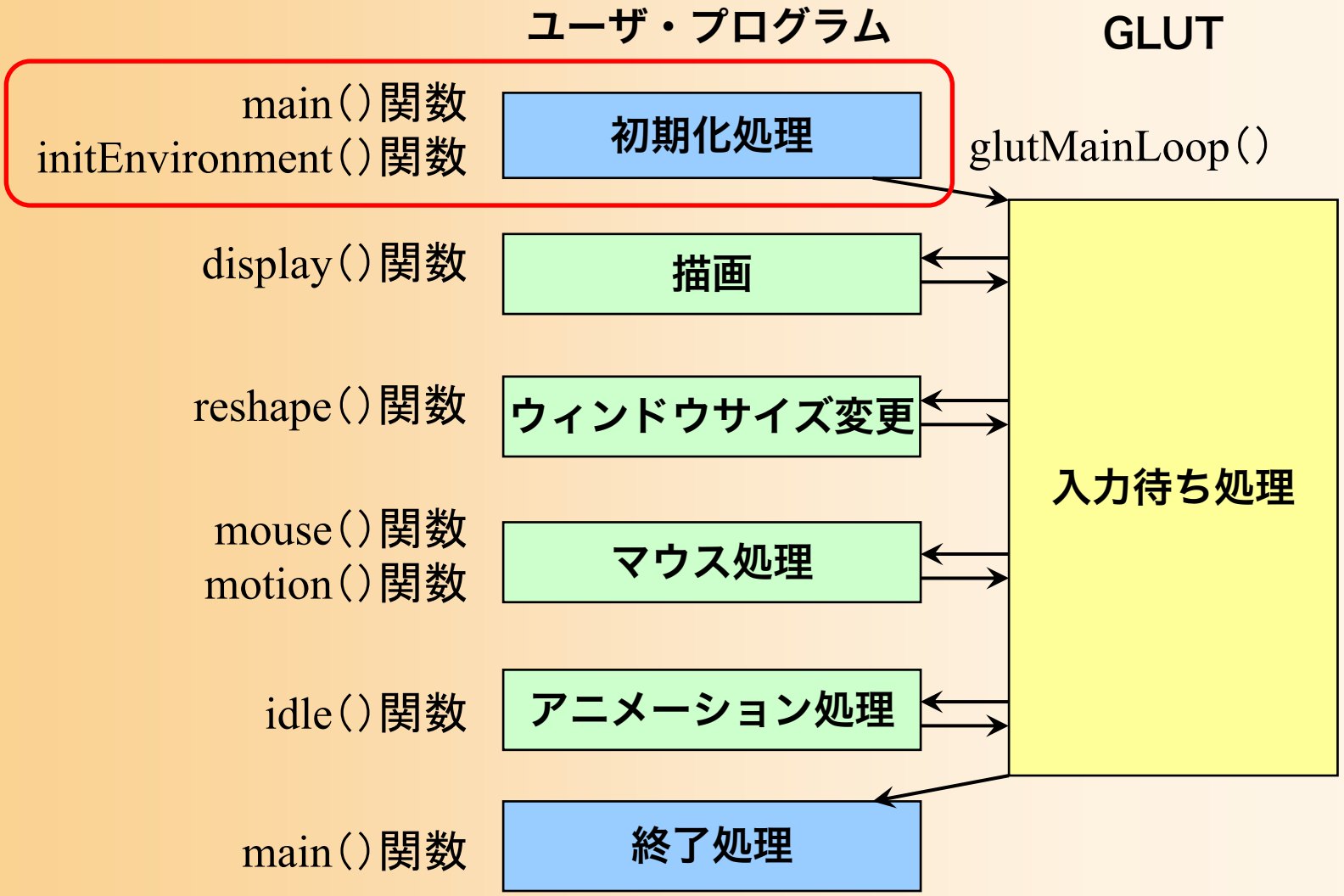


グローバル変数の定義

- グローバル変数
  - 全ての関数からアクセスできる変数
  - ここでは視点操作に関する変数を定義
    - 詳細は後で説明

```
// 視点操作のための変数
float  camera_pitch = -30.0;  // X軸を軸とするカメラの回転角度

// マウスのドラッグのための変数
int  drag_mouse_r = 0;  // 右ボタンをドラッグ中かどうかのフラグ
                                    （0:非ドラッグ中,1:ドラッグ中）
int  last_mouse_x;      // 最後に記録されたマウスカーソルのX座標
int  last_mouse_y;      // 最後に記録されたマウスカーソルのY座標
```

サンプルプログラムの構成



開始・初期化処理

- main関数
  - GLUTの初期化（メイン関数）
  - コールバック関数の設定
  - initEnvironment関数の呼び出し
  - GLUTのメインループの開始
- initEnvironment関数
  - レンダリングの設定
  - 光源の設定

1. GLUTの初期化

```
int  main( int argc, char ** argv )
{
        // GLUTの初期化
        glutInit( &argc, argv );
        glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
        glutInitWindowSize( 320, 320 );
        glutInitWindowPosition( 0, 0 );
        glutCreateWindow(“OpenGL & GLUT sample program");

        ......
}
```

2. コールバック関数の設定

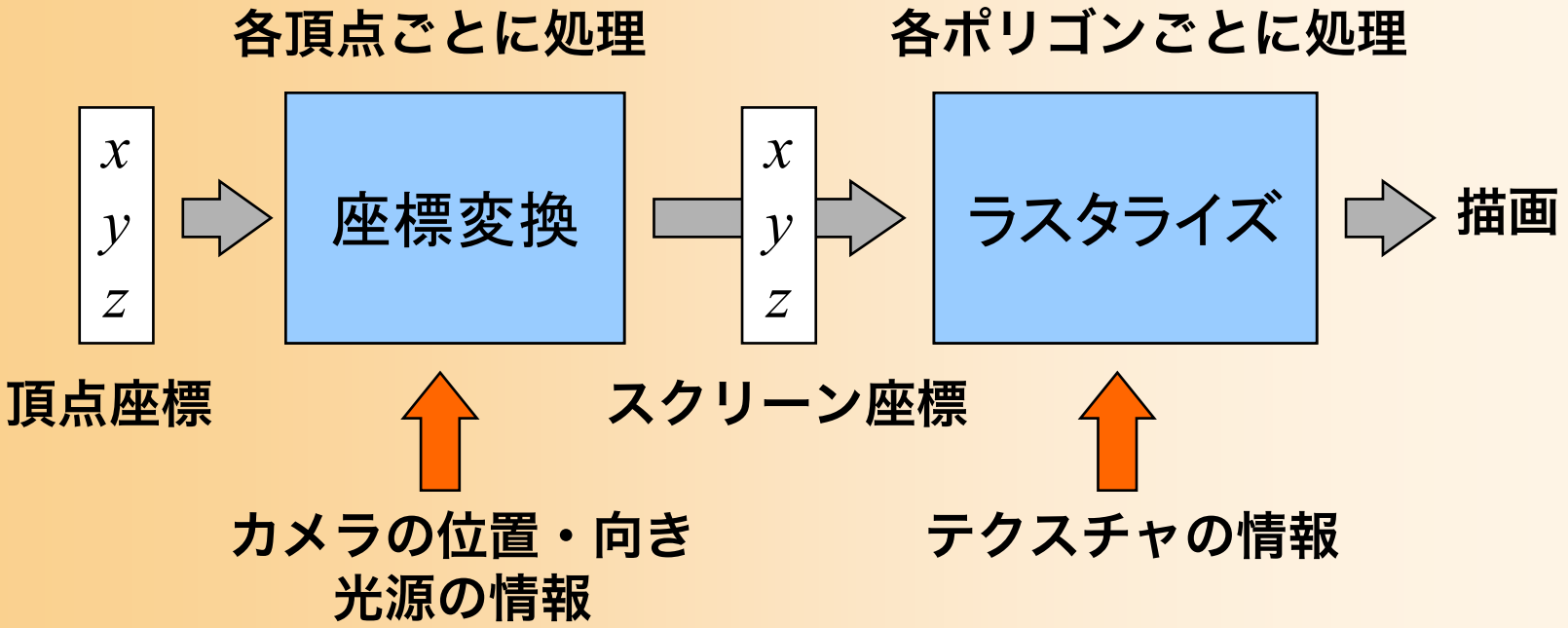
```
int  main( int argc, char ** argv )
{
        ......
        // コールバック関数の登録
        glutDisplayFunc( display );
        glutReshapeFunc( reshape );
        glutMouseFunc( mouse );
        glutMotionFunc( motion );
        glutIdleFunc( idle );

        // 環境初期化
        initEnvironment();

        // GLUTのメインループに処理を移す
        glutMainLoop();
        return 0;
}
```

3. レンダリングの設定

- Ｚバッファ法によるレンダリングの各種設定
  - 標準的な描画機能を設定（詳しい内容は後日説明）

```
void  initEnvironment( void )
{
        ......
        // 光源計算を有効にする
        glEnable( GL_LIGHTING );

        // 物体の色情報を有効にする
        glEnable( GL_COLOR_MATERIAL );

        // Zテストを有効にする
        glEnable( GL_DEPTH_TEST );

        // 背面除去を有効にする
        glCullFace( GL_BACK );
        glEnable( GL_CULL_FACE );

        // 背景色を設定
        glClearColor( 0.5, 0.5, 0.8, 0.0 );
}
```

レンダリング・パイプラインの設定



- 描画を行う前に、さまざまな設定を行える
- 各機能の使用の有無（Zバッファ、背面除去等）
- カメラの位置・向き（変換行列）の設定
- 光源の情報（位置・向き・色など）を設定

描画機能の設定

- さまざまな描画機能のオン・オフを設定
  - 不必要な処理はオフにすることで、高速できる
  - 初期状態ではオフになっている機能が多いので、必要な機能はオンに設定する必要がある
- glEnable（機能の種類）, glDisable（・・・）
  - 各機能のオン・オフを変更する
    - GL_LIGHTING, GL_COLOR_MATERIAL, GL_DEPTH_TEST, CL_CULL_FACE, etc
  - 各機能の動作はそれぞれ別の関数で設定

サンプルプログラムの描画機能の設定

- 標準的な描画の設定（最初に一度だけ設定）

```
void  initEnvironment( void )
{
        ......
        // 光源計算を有効にする
        glEnable( GL_LIGHTING );

        // 物体の色情報を有効にする
        glEnable( GL_COLOR_MATERIAL );

        // Zテストを有効にする
        glEnable( GL_DEPTH_TEST );

        // 背面除去を有効にする
        glCullFace( GL_BACK );
        glEnable( GL_CULL_FACE );

        // 背景色を設定
        glClearColor( 0.5, 0.5, 0.8, 0.0 );
}
```

描画機能の設定（その他）

- 背面除去の設定
  - glCullFace（ GL_BACK ）
  - 表面・背面のどちらを描画しないかを設定
- 背景色の設定
  - glClearColor（ r, g, b, a ）
  - 画面をクリアしたときの色を設定

4. 光源の設定

- シェーディングのための光源情報の設定
  - １つの点光源を設定（詳細は後述）

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );

glEnable( GL_LIGHT0 );
glEnalbe( GL_LIGHTING );
```

OpenGLの光源処理の概要

- 光源と物体の素材（頂点の色）・法線によって、描画される頂点（ポリゴン）の色が決まる
- OpenGLの光源処理
  - OpenGLの関数を使って、光源や物体の素材・法線の情報を指定
  - OpenGLは、各頂点ごとに、自動的に光源処理を行い、各頂点の色を決定
    グローシェーディングにより、各頂点の色をもとに、ポリゴンが描画される

光のモデル（復習）

- 輝度の計算式
  - 全ての光による影響を足し合わせることで、物体上の点の輝度が求まる

$$I = I_a k_a + \sum_{i=1}^{n_L} I_i \left[k_d \left(N \cdot L \right) + k_s \left(R \cdot V \right)^n \right] + k_r I_r + k_t I_t$$

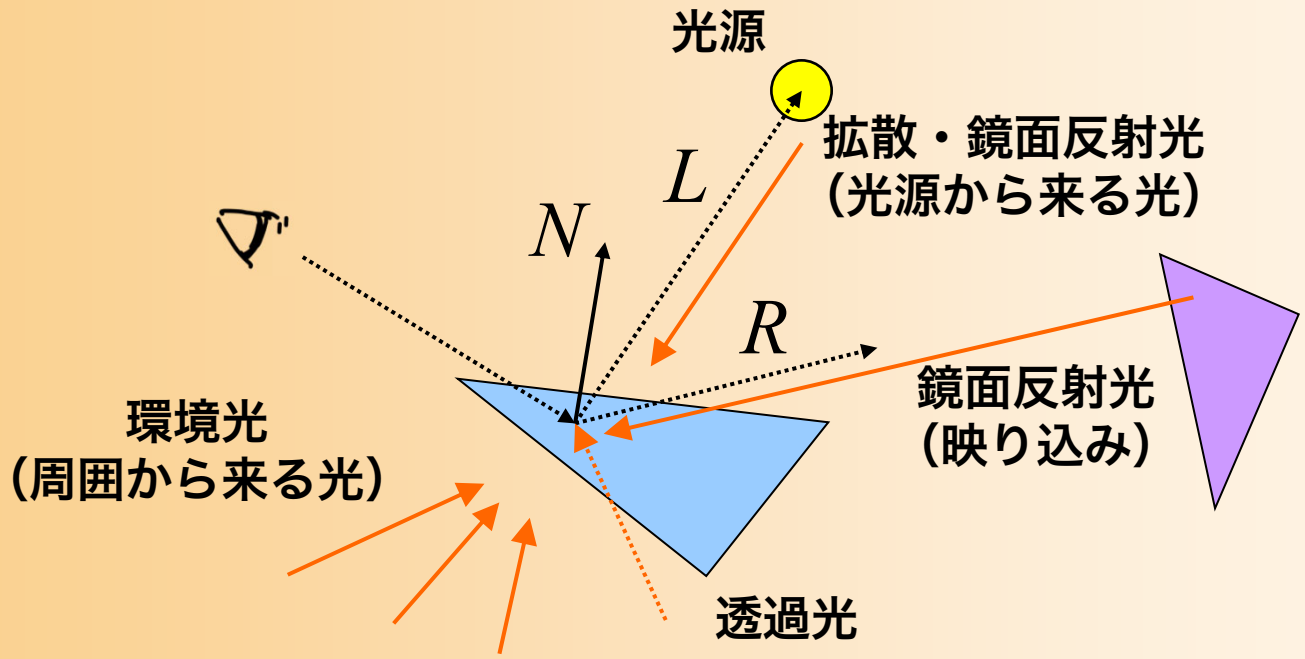
環境光　拡散反射光　鏡面反射光（局所照明）　鏡面反射光（大域照明）　透過光

それぞれの光源からの光（局所照明）　大域照明

$$k_a + n_L \left(k_d + k_s \right) + k_r + k_t = 1$$

各係数の和は１

光のモデル（復習）



$$I = I_a k_a + \sum_{i=1}^{n_L} I_i \left[k_d \left(N \cdot L \right) + k_s \left(R \cdot V \right)^n \right] + k_r I_r + k_t I_t$$

環境光　拡散反射光　鏡面反射光（局所照明）　鏡面反射光（大域照明）　透過光

それぞれの光源からの光（局所照明）　大域照明

OpenGLの光源処理

- 光のモデルにもとづき、各光源による輝度を、RGBごとに次式で計算して加算

$$Color = L_{\text{ambient}} \cdot M_{\text{ambient}} + \max\{\ \mathbf{l}\cdot\mathbf{n},\ 0\ \}\ L_{\text{diffuse}} \cdot M_{\text{diffuse}}$$

$$+\max\{\ \mathbf{s}\cdot\mathbf{n},\ 0\ \}^{M_{\text{specular\_factor}}}\ L_{\text{specular}} \cdot M_{\text{specular}}$$

  - max{A, B}は、A, B のうち大きい値を使用
    内積が負の場合は、その項は０になる
  - 全ての値を足し合わせた結果は、0.0～1.0の範囲に丸められる

- $L_{\text{ambient}}, L_{\text{diffuse}}, L_{\text{specular}}$ は光の輝度
- $M_{\text{ambient}}, M_{\text{diffuse}}, M_{\text{specular}}, M_{\text{specular\_factor}}$ は素材の特性

光源情報の設定

- 光源情報の設定
  - glLight(), glLightv() 関数 を使用
    - 光源番号、設定パラメタの種類、設定する値、を指定
    - glLight() 関数はスカラ値を設定
    - glLightv() 関数はベクトル値を設定
- 光源処理を有効にする
  - 光源処理を有効にする glEnable(GL_LIGHTING)
  - 各光源の影響を有効にする glEnable(GL_LIGHT0)

光源情報の設定の例（1）

- 初期化処理での設定

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );

glEnable( GL_LIGHT0 );
glEnalbe( GL_LIGHTING );
```

詳細は、後ほど説明

光源情報の設定の例（2）

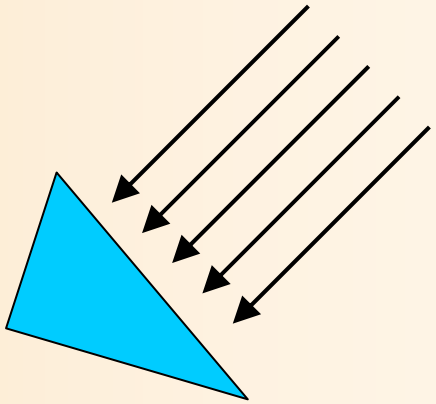
- 変換行列の変更後に、光源位置を再設定
  - 光源計算は、カメラ座標系で適用されるため

```
void  display( void )
{
        ・・・・・・
        // 変換行列を設定（ワールド座標系→カメラ座標系）
        glMatrixMode( GL_MODELVIEW );
        ・・・・・・

        // 光源位置を設定（変換行列の変更にあわせて再設定）
        float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
        glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
        ・・・・・・
```

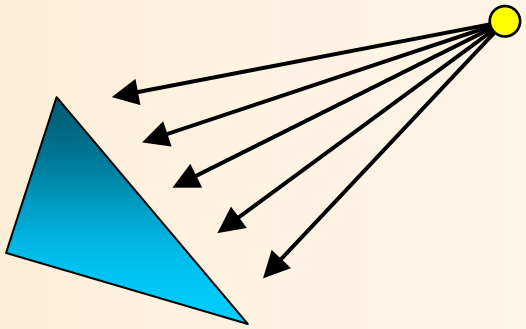
光源の種類と設定方法（1）

- 平行光源
  - （x,y,z）の方向から平行に光が来る
  - 光源位置のw座標を0.0に設定

無限遠に光源があると見なせる



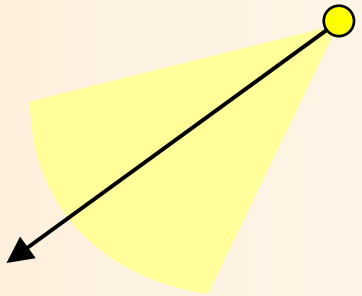
- 点光源
  - （x,y,z）の位置に光源がある
  - 光源位置のw座標を1.0に設定

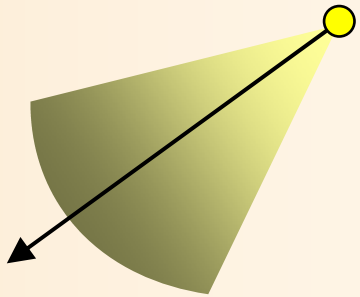


光源の種類と設定方法（2）

- スポットライト光源
  - 点光源にさらに、スポットライトの向き・角度範囲などの情報を設定したもの
- 光源の減衰も設定可能
  - 点光源・スポットライト光源から距離が離れるほど暗くなるような効果を加える
- 設定方法の説明は省略

指定した方向・角度にのみ有効な点光源





光源情報の設定の例

- サンプルプログラムの例

```
float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };

glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );

glEnable( GL_LIGHT0 );
glEnalbe( GL_LIGHTING );
```

光源位置のw座標が1.0なので、点光源となる

LIGHT0の
・光源の位置・種類
・拡散反射成分の色
・鏡面反射成分の色
を設定

LIGHT0の
・環境光成分の色
を設定

一般的な光源の設定方針

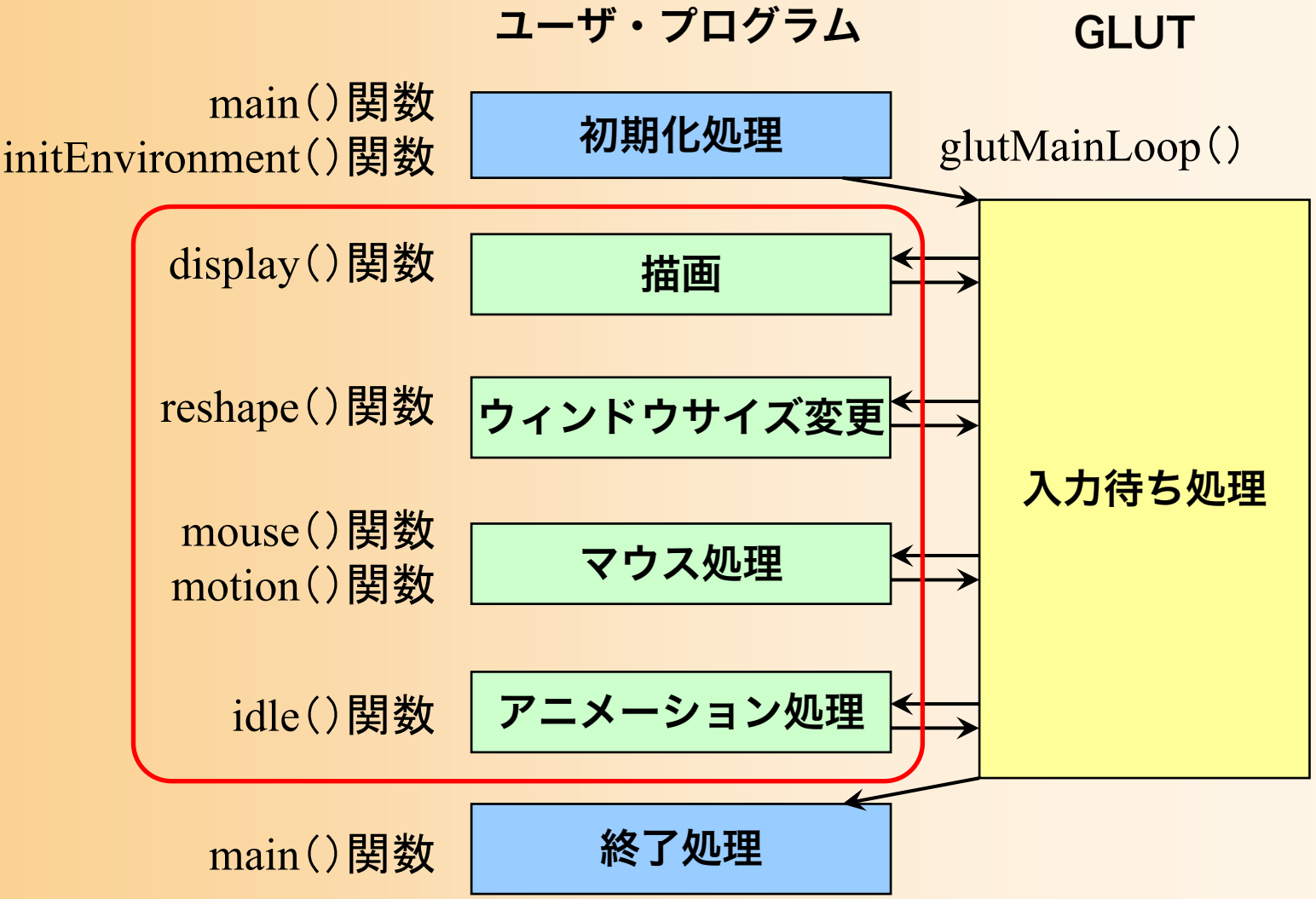
- LIGHT0を使って環境の主な光源を設定
  - その環境の明るさに応じて環境光を設定
  - 全体の明るさを決めるような、平行光源or点光源を設定
- LIGHT1以降を使って追加の光を設定
  - 電灯や車など、空間中にあるオブジェクトが周囲のオブジェクトを照らすような場合に、点光源やスポットライトを追加する
  - ２番目以降の光源では、環境光はあまり大きくしないことが多い

素材の設定

- 頂点の色の設定
  - glColor()関数
    - デフォルトでは、頂点の環境特性と拡散反射特性を同時に設定 （個別に設定することも可能）
- その他の素材特性を個別に設定（詳細は省略）
  - glMaterial()関数
  - 環境特性、拡散反射特性、鏡面反射特性、鏡面反射係数など

$$Color = L_{\text{ambient}} \cdot M_{\text{ambient}} + \max\{\ \mathbf{l}\cdot\mathbf{n},\ 0\ \}\ L_{\text{diffuse}} \cdot M_{\text{diffuse}}$$
$$+\max\{\ \mathbf{s}\cdot\mathbf{n},\ 0\ \}^{M_{\text{specular\_factor}}}\ L_{\text{specular}} \cdot M_{\text{specular}}$$

サンプルプログラムの構成



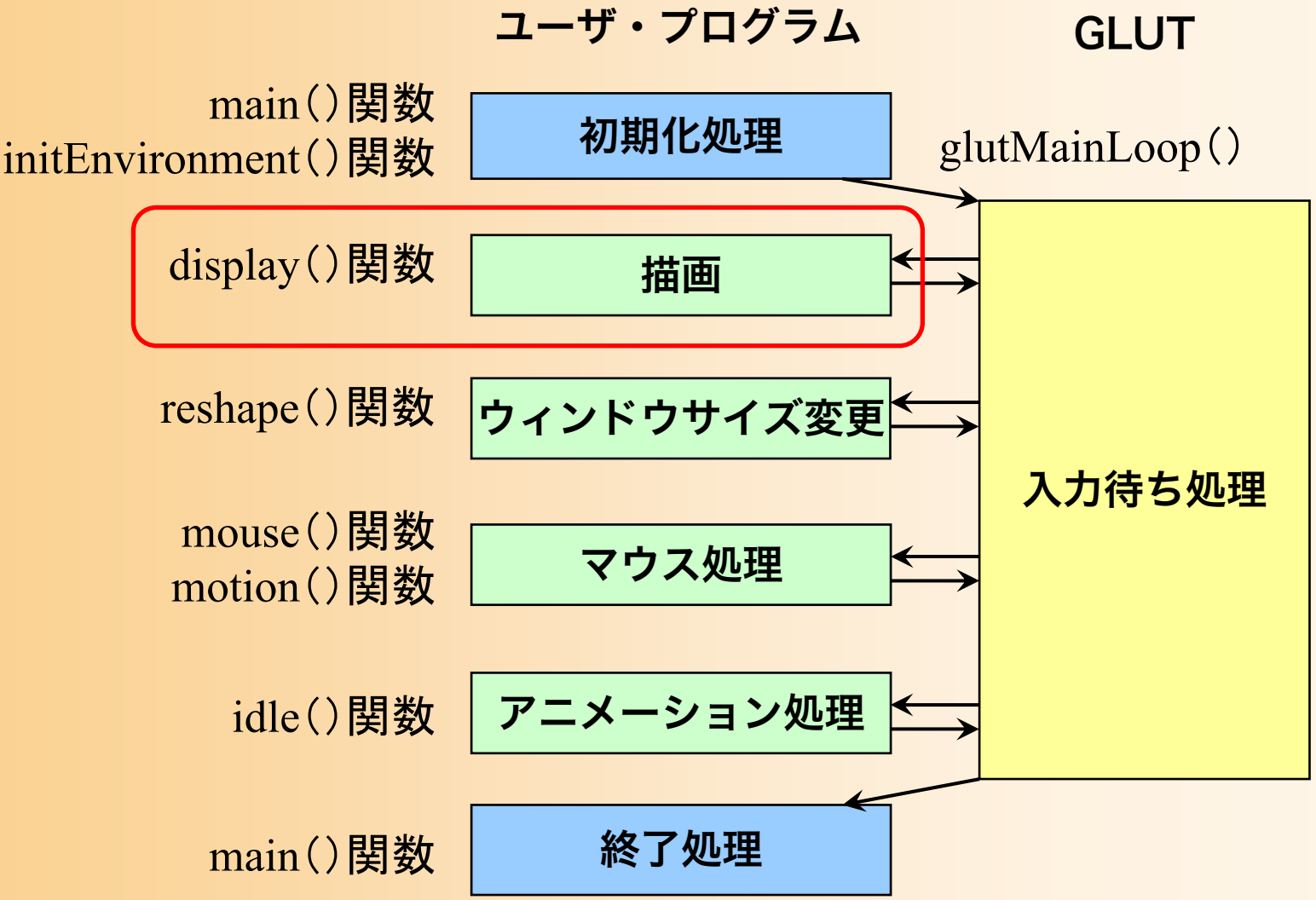
コールバック関数（1）

- 描画コールバック関数 display()
  - 再描画が必要な時に呼ばれる
  - 本プログラムでは、変換行列の設定、地面と 1枚のポリゴンの描画、を行っている
- サイズ変更コールバック関数 reshape()
  - ウィンドウサイズ変更時に呼ばれる
  - 本プログラムでは、視界の設定、ビューポート変換の設定、を行っている

コールバック関数（2）

- マウスクリック・コールバック関数 mouse()
  - マウスのボタンが押されたとき、離されたときに呼ばれる
  - 本プログラムでは、右ボタンの押下状態を記録
- マウスドラッグ・コールバック関数 motion（）
  - マウスがウィンドウ上でドラッグされたときに呼ばれる
  - 本プログラムでは、右ドラッグされたときに、視点の回転角度を変更
- アイドル・コールバック関数 idle()
  - 処理が空いた時に定期的に呼ばれる
  - 本プログラムでは、現在は何の処理も行っていない

サンプルプログラムの構成



描画関数の流れ

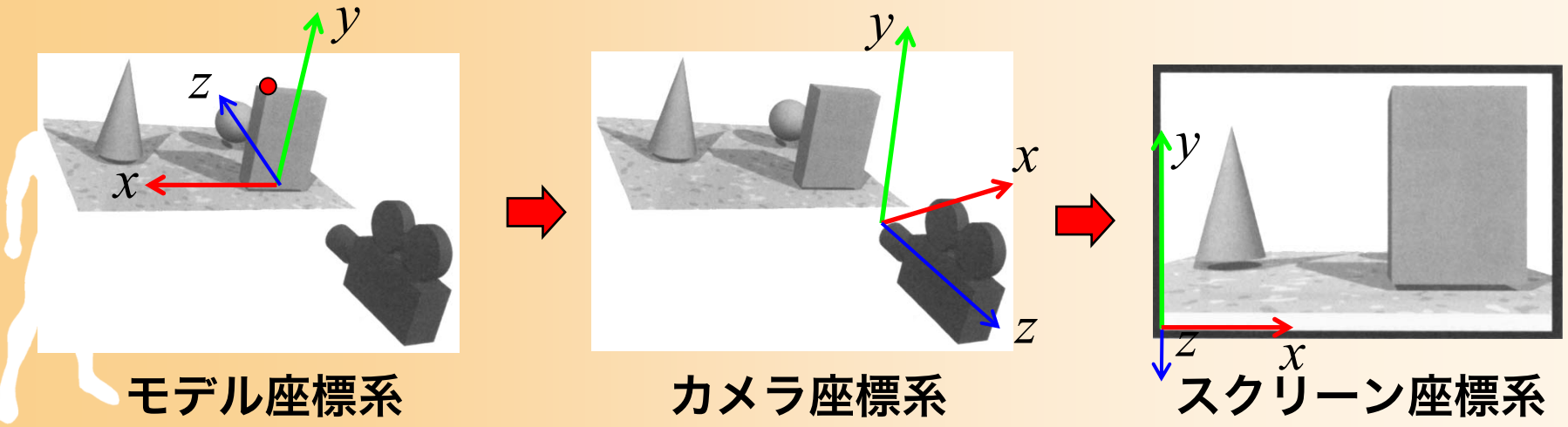
```
//
//  ウィンドウ再描画時に呼ばれるコールバック関数
//
void  display( void )
{
        // 画面をクリア（ピクセルデータとZバッファの両方をクリア）
        // 変換行列を設定（ワールド座標系→カメラ座標系）
        // 光源位置を設定（モデルビュー行列の変更にあわせて再設定）
        // 地面を描画
        // 変換行列を設定（物体のモデル座標系→カメラ座標系）
        // 物体（１枚のポリゴン）を描画
        // バックバッファに描画した画面をフロントバッファに表示
}
```

描画関数（1/4）

```
void  display( void )
{
        // 画面をクリア（ピクセルデータとZバッファの両方をクリア）
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        // 変換行列を設定（ワールド座標系→カメラ座標系）
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity();
        glTranslatef( 0.0,  0.0, - 15.0 );
        glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

        // 光源位置を設定（モデルビュー行列の変更にあわせて再設定）
        float  light0_position[] = { 10.0, 10.0, 10.0, 1.0 };
        glLightfv( GL_LIGHT0, GL_POSITION, light0_position );

        ......
}
```

座標変換（復習）

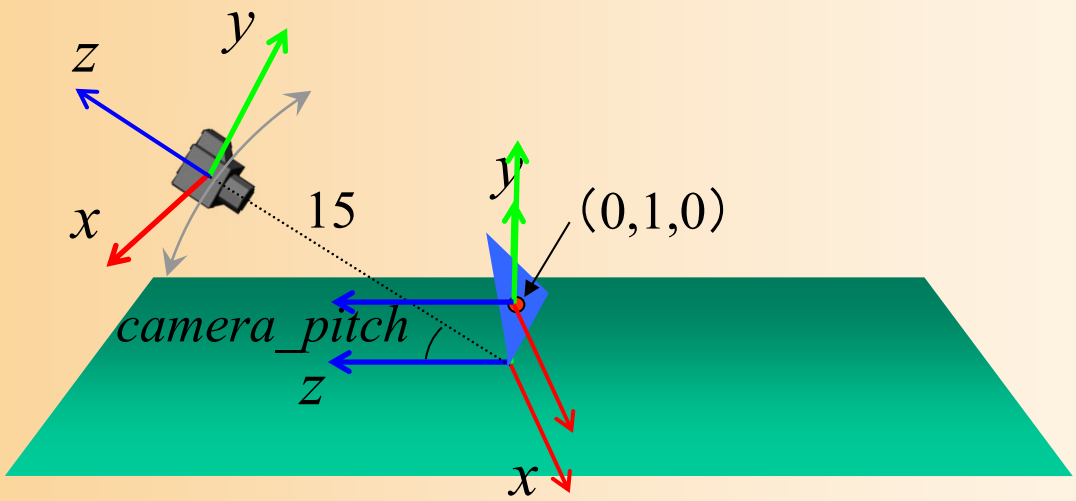
- 座標変換（Transformation）
  - 行列演算を用いて、ある座標系から、別の座標系に、頂点座標やベクトルを変換する技術
    - カメラから見た画面を描画するためには、モデルの頂点座標をカメラ座標系（最終的にはスクリーン座標系）に変換する必要がある



モデル座標系　　カメラ座標系　　スクリーン座標系

変換行列の設定

- サンプルプログラムでのカメラ位置の設定



- 以下の変換行列により表せる

ポリゴンを基準とする座標系での頂点座標

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

カメラから見た頂点座標（描画に使う頂点座標）

変換行列の設定

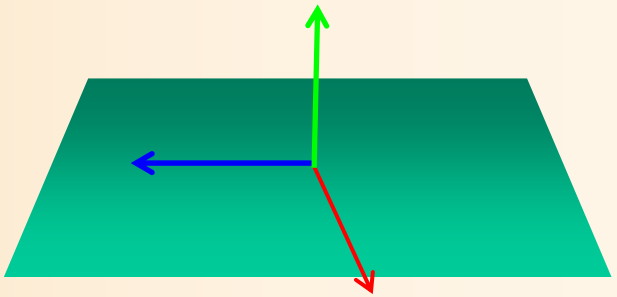
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -15 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-camera\_pitch) & -\sin(-camera\_pitch) & 0 \\ 0 & \sin(-camera\_pitch) & \cos(-camera\_pitch) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

```
// 変換行列を設定（ワールド座標系→カメラ座標系）
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( 0.0,  0.0, - 15.0 );
glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );

// 地面を描画
・・・・・・

// 変換行列を設定（物体のモデル座標系→カメラ座標系）
glTranslatef( 0.0, 1.0, 0.0 );

// 物体（１枚のポリゴン）を描画
・・・・・・
```

描画関数（2/4）

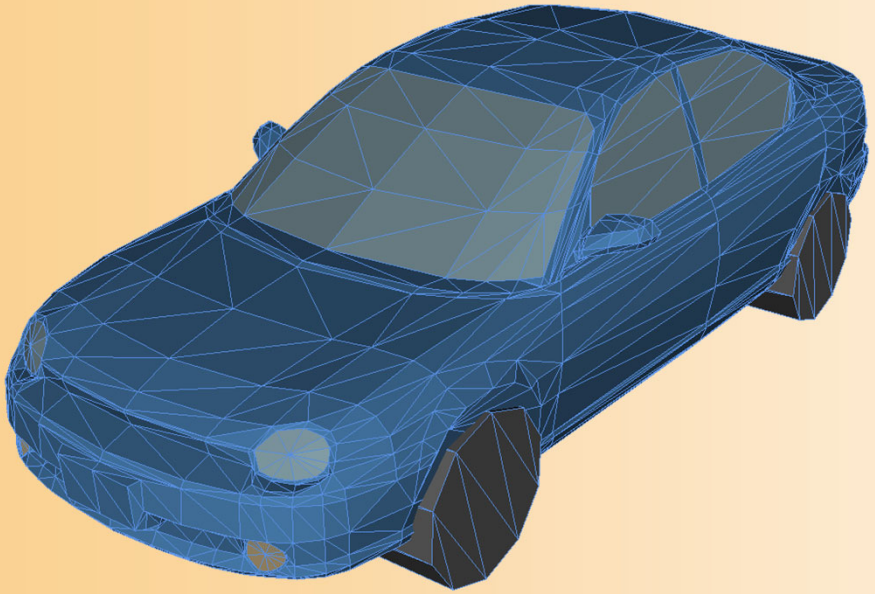
- １枚の四角形として地面を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - 真上（0,1,0）を向き、水平方向の長さ10の四角形

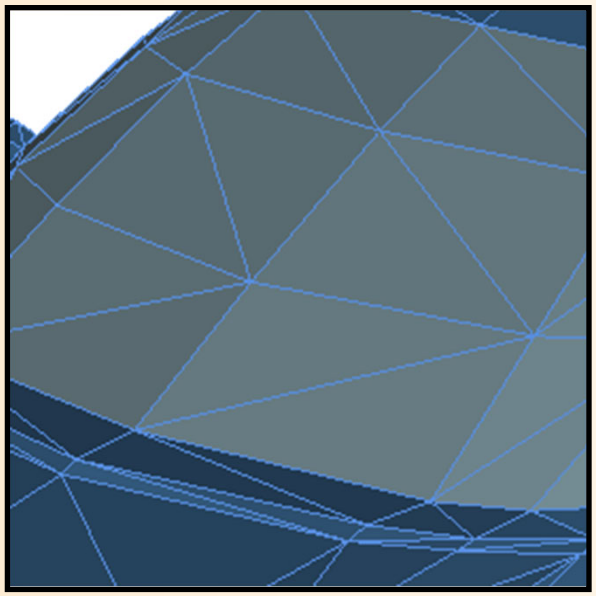
```
// 地面を描画
glBegin( GL_POLYGON );
        glNormal3f( 0.0, 1.0, 0.0 );
        glColor3f( 0.5, 0.8, 0.5 );

        glVertex3f( 5.0, 0.0, 5.0 );
        glVertex3f( 5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0, 5.0 );
glEnd();
```



ポリゴンモデル（復習）

- 物体の表面の形状を、多角形（ポリゴン）の集まりによって表現する方法
  - 最も一般的なモデリング技術
  - 本講義の演習でも、ポリゴンモデルを扱う

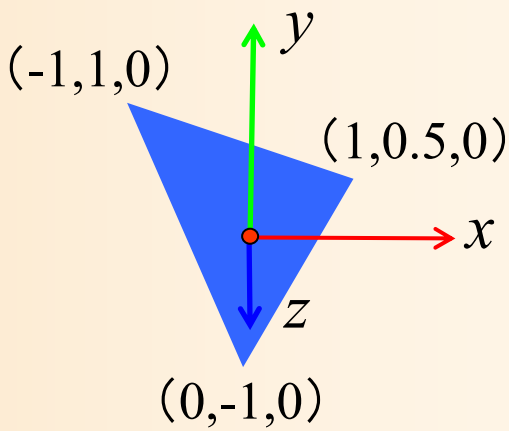




描画関数（3/4）

- 同じく、１枚の三角形を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - ポリゴンを基準とする座標系（モデル座標系）で頂点位置・法線を指定

```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );
        glVertex3f( 0.0,-1.0, 0.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```



参考：複雑なポリゴンモデルの描画

- プログラムに直接頂点座標等を記述するのではなく、以下のように、配列を使ってデータを管理するのが一般的（詳しくは後日説明）

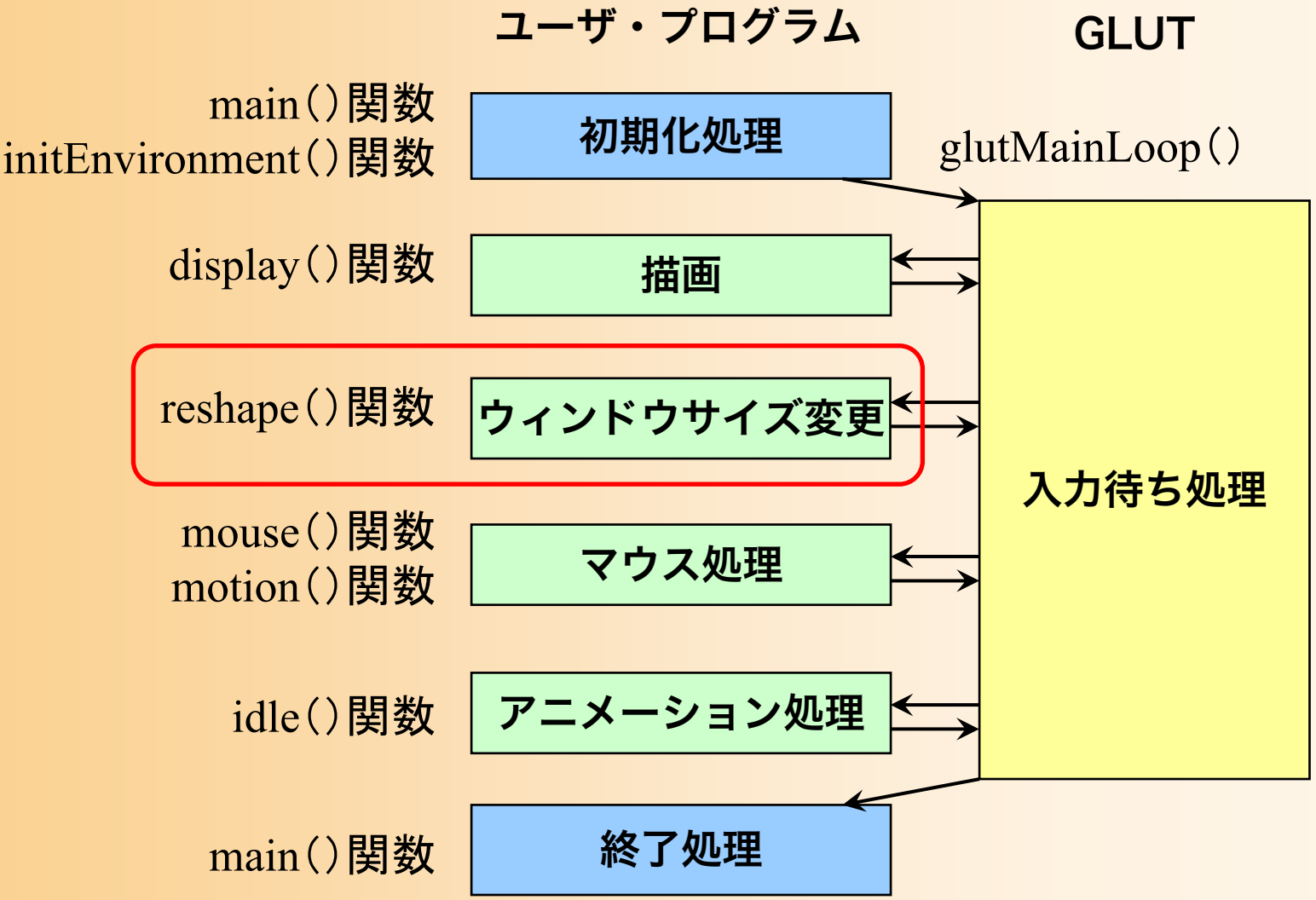
```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 },
          {-1.0,-0.8, 1.0 }, {-1.0,-0.8,-1.0 }
};

// 三角面インデックス（各三角面を構成する頂点の頂点番号）の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};
```

描画関数（4/4）

- 描画完了
  - 描画途中の画面が表示されることを避けるために、描画は裏画面（バックバッファ）に行い、描画が完了したところで、表画面（フロントバッファ）に表示する

```
        ……

        // バックバッファに描画した画面をフロントバッファに表示
        glutSwapBuffers();
}
```

サンプルプログラムの構成



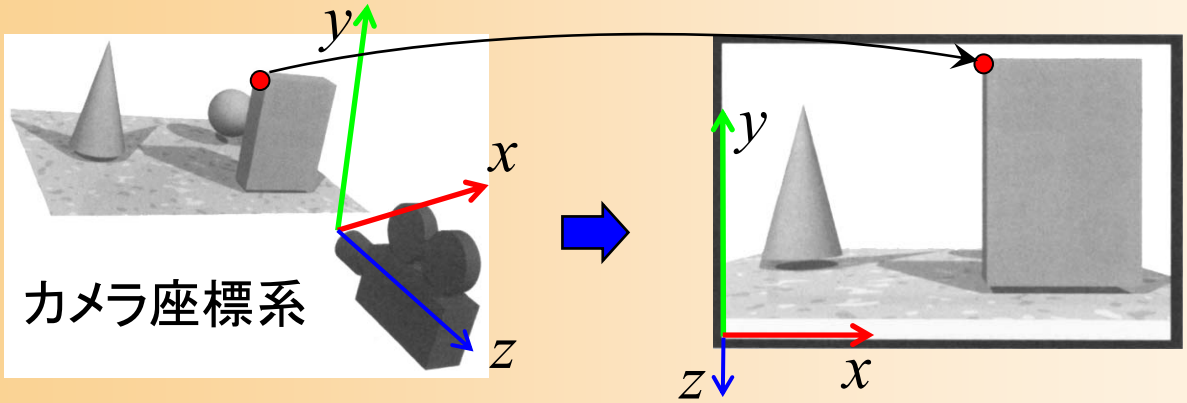
ウィンドウサイズ変更時の処理

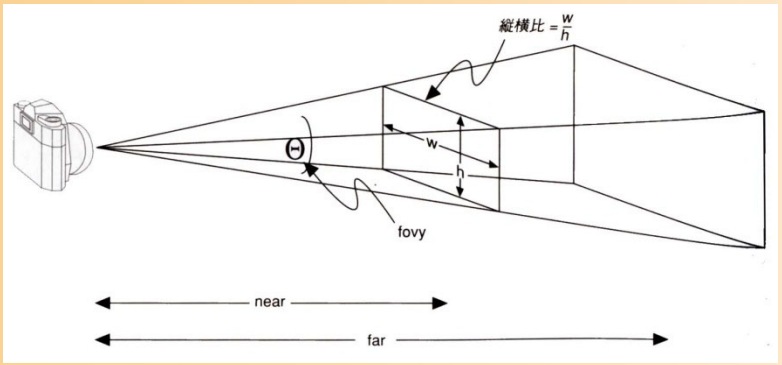
- 画面全体に描画を行うよう設定
- 射影変換行列の設定（視野角を45度とする）
  - 通常は、この設定のままで、変更は必要ない

```
void  reshape( int w, int h )
{
        // ウィンドウ内の描画を行う範囲を設定
        //（ウィンドウ全体に描画するよう設定）
        glViewport(0, 0, w, h);

        // カメラ座標系→スクリーン座標系への変換行列を設定
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();
        gluPerspective( 45, (double)w/h, 1, 500 );
}
```

参考：射影変換の設定

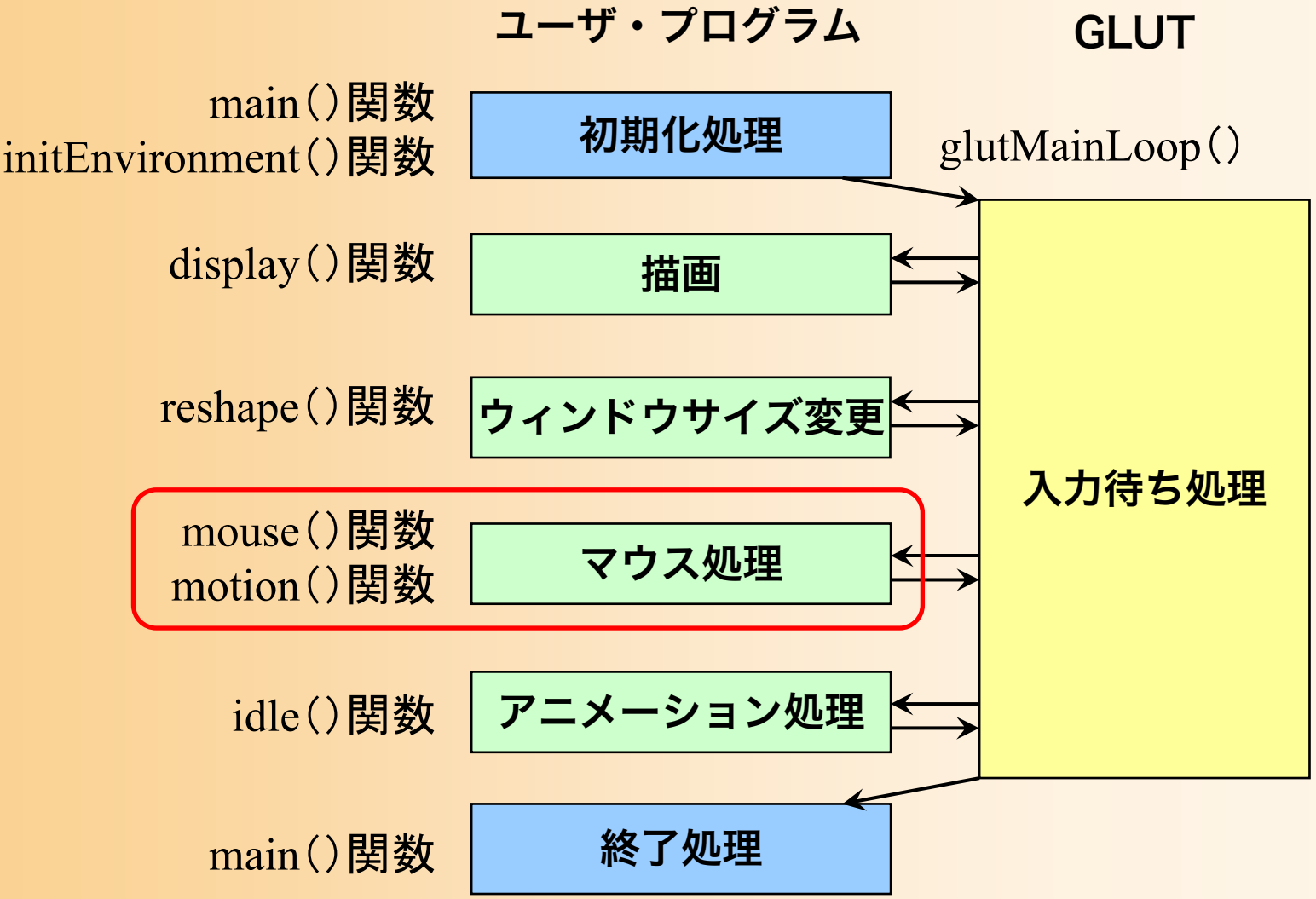
- カメラ座標系からスクリーン座標系への座標変換（射影変換）の設定





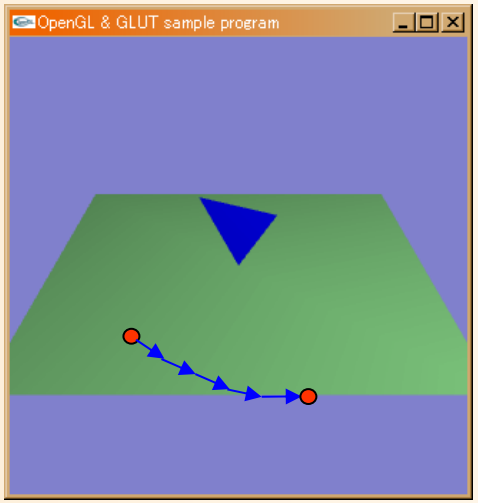
遠くにあるものほど小さく描画されるような変換（透視射影変換）を適用

サンプルプログラムの構成



マウス操作時の処理

- マウス操作のコールバック関数
  - mouse（）関数
    - マウスのボタンが、押されたとき、または、離されたときに呼ばれる
  - motion（）関数
    - マウスのボタンが押された状態で、マウスが動かされたとき（ドラッグ時）に定期的に呼ばれる
    - ボタンが押されない状態で、マウスが動かされたときに呼ばれる関数もある（今回は使用しない）



マウス操作時の処理 (クリック処理関数)

- 右ボタンがクリックされたことを記録
  - 変数 drag_mouse_r に状態を格納

```
// マウスクリック時に呼ばれるコールバック関数
void  mouse( int button, int state, int mx, int my )
{
        // 右ボタンが押されたらドラッグ開始のフラグを設定
        if ( ( button == GLUT_RIGHT_BUTTON ) && ( state == GLUT_DOWN ) )
                drag_mouse_r = 1;
        // 右ボタンが離されたらドラッグ終了のフラグを設定
        else if ( ( button == GLUT_RIGHT_BUTTON ) && ( state == GLUT_UP ) )
                drag_mouse_r = 0;

        // 現在のマウス座標を記録
        last_mouse_x = mx;
        last_mouse_y = my;
}
```

マウス操作時の処理（ドラッグ処理関数1）

- ドラッグされた距離に応じて視点を変更
  - 視点の仰角camera_pitch を変化
    - 前回と今回のマウス座標の差から計算

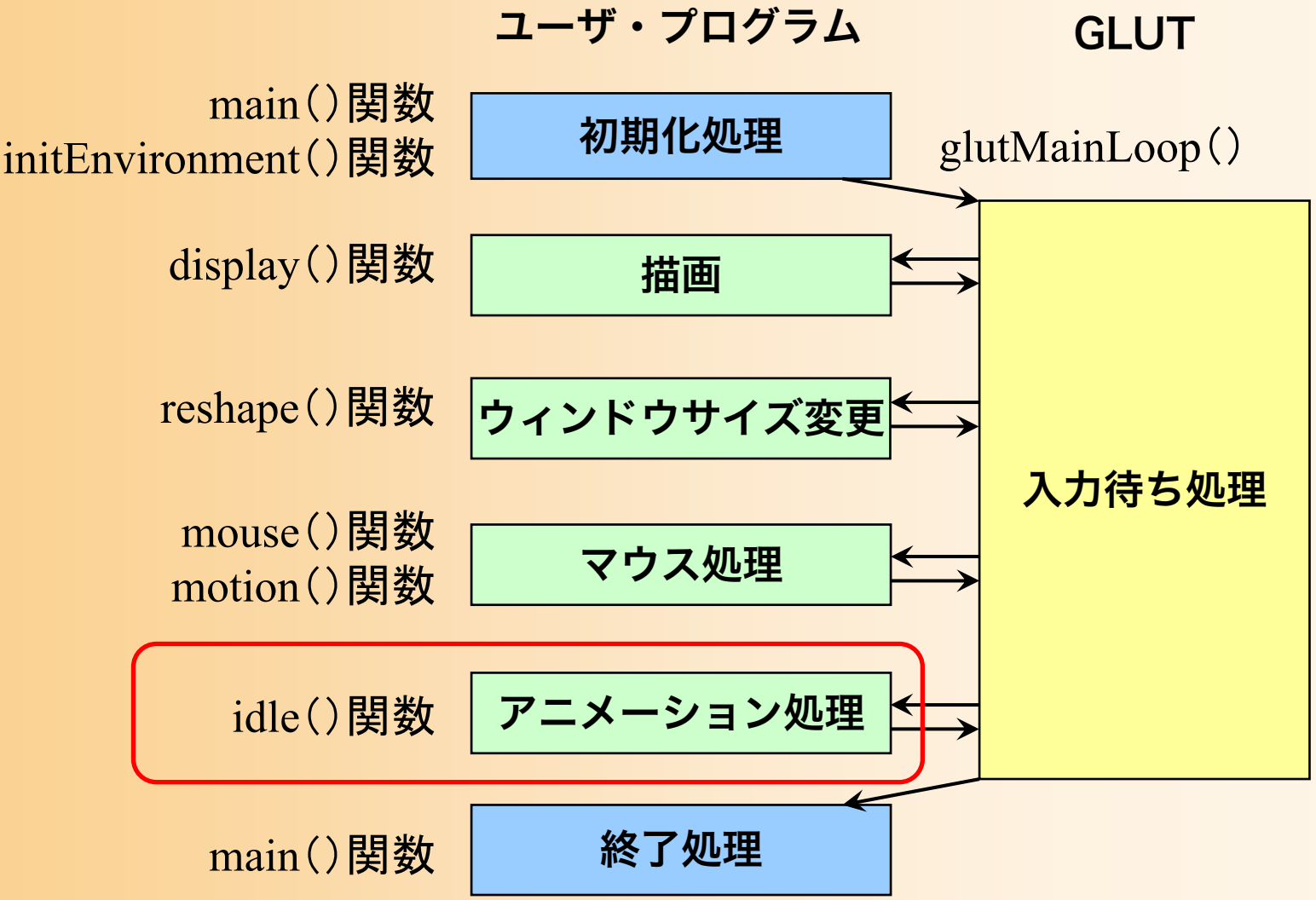
```
void  motion( int mx, int my )
{
        // 右ボタンのドラッグ中であれば、
        // マウスの移動量に応じて視点を回転する
        if ( drag_mouse_r == 1 )
        {
                // マウスの縦移動に応じてX軸を中心に回転
                camera_pitch -= ( my - last_mouse_y ) * 1.0;
                if ( camera_pitch < -90.0 )
                        camera_pitch = -90.0;
                else if ( camera_pitch > 0.0 )
                        camera_pitch = 0.0;
        }
        ......
```

マウス操作時の処理（ドラッグ処理関数2）

- 再描画の指示を行う
  - 視点の方位角 camera_pitch の変化に応じて、画面を再描画するため

```
        // 今回のマウス座標を記録
        last_mouse_x = mx;
        last_mouse_y = my;

        // 再描画の指示を出す
        glutPostRedisplay();
}
```

サンプルプログラムの構成



アイドル時の処理

- 描画やマウス入力を処理する必要がないときに定期的に呼ばれる関数
  - 物体の位置・向きを少しずつ変化させるといった、アニメーションを実現するために利用できる
  - サンプルプログラムでは、現在は何も処理を行っていない（今後処理を追加する）

```
void  idle( void )
{
        // 現在は、何も処理を行なわない
}
```

本日の内容

- ガイダンス
- コンピュータグラフィックスの概要と応用
- 復習：３次元グラフィックスの要素技術
- 復習：３次元グラフィックスのプログラミング
- 復習：OpenGL&GLUT プログラミング
- 復習：OpenGL&GLUT サンプルプログラム

まとめ

- ガイダンス
- コンピュータグラフィックスの概要と応用
- 復習：３次元グラフィックスの要素技術
- 復習：３次元グラフィックスのプログラミング
- 復習：OpenGL&GLUT プログラミング
- 復習：OpenGL&GLUT サンプルプログラム

次回予告

- 視点操作
  - 利用者が視点を操作して、仮想空間や物体を適切な位置・方向から見るための機能
  - 変換行列による、代表的な視点操作の実現方法
- 第1回 レポート課題