佐賀大学理工学部

「コンピュータグラフィックス演習（新カリ）/コンピュータグラフィックス（旧カリ）」
授業資料

担当者: 山下義行

R4 年度後期

目次

第 1 章　はじめに **1**

1.1　授業の概要 1

1.2　成績評価 1

第 2 章　グラフィックス環境の準備 **3**

2.1　プログラミング言語 p5.js 3

2.2　動かしてみる 3

第 II 部 **10**

第 3 章　コンピュータグラフィックスの基礎知識 **11**

3.1　ベクタ・グラフィックス 11

3.2　ラスタ・グラフィックス 15

第 III 部 **18**

3.3　画像情報とフレームメモリ 19

3.3.1　画素 19

3.3.2　画像サイズ 19

3.3.3　画素の順序 20

3.3.4　フレームメモリ 21

3.3.5　X-Y 座標系 22

3.3.6　寄り道：「ラスタ」の語源 23

3.4　画素情報 24

3.4.1　2 値 24

3.4.2　グレースケール 25

3.4.3　フルカラー 27

3.4.4　インデックスカラー 30

3.4.5　色深度 33

3.4.6 寄り道：ラスタ画像の使用メモリ量 33

第 IV 部 **35**

第 4 章 2 次元グラフィックス **36**

4.1 様々な線画 36

4.1.1 三角形 36

4.1.2 正多角形 37

4.1.3 星型正多角形 39

4.1.4 らせんと渦巻き 40

4.1.5 線の色、幅の変更 42

4.2 様々な面画 43

4.2.1 三角形 43

4.2.2 図形の重ね合わせとアルファ値 46

第 V 部 **50**

4.2.3 様々な形状 51

4.3 CG アニメーション 53

4.3.1 様々な CG アニメーション 53

4.3.2 表示のタイミングとダブルバッファ（授業対象外） 57

4.3.3 イベント処理、その他 60

4.3.4 モグラたたき風ゲームの開発 62

第 VI 部 **66**

4.4 ラスタライズ 67

4.4.1 点のラスタライズ 67

4.4.2 線分のラスタライズ（授業対象外） 67

4.4.3 円周のラスタライズ（授業対象外） 78

4.4.4 面のラスタライズ（授業対象外） 84

4.5 滑らかな曲線 88

4.5.1 問題の定義 88

4.5.2 ベジエ曲線 89

4.5.3 様々なスプライン曲線 94

4.6 アンチエイリアシング 96

第 VII 部 **97**

第 5 章　3 次元グラフィックス：線画 **98**

5.1 3D CG とは 98
5.2 3D CG の基本モデル 99
5.3 投影変換 103
5.3.1 透視投影変換 103
5.3.2 p5.js における透視投影変換 106
5.3.3 平行投影変換 113
5.3.4 p5.js における平行投影変換 113

第 VIII 部 **115**

5.4 画角、キャンバスのアスペクト比、クリッピング空間 116
5.4.1 画角 116
5.4.2 アスペクト比 117
5.4.3 クリッピング空間 117
5.4.4 p5.js における透視投影変換のパラメータ変更 119
5.5 視野変換 121
5.5.1 ワールド座標系 122
5.5.2 視点の位置に関する座標の平行移動 124
5.5.3 視点の方向に関する基底変換 125
5.5.4 p5.js における視野変換 126

第 IX 部 **130**

5.6 同次座標系による高速化 131
5.6.1 定義 131
5.6.2 平行移動 132
5.6.3 基底変換 133
5.6.4 透視投影変換 133
5.6.5 キャンバス座標への変換 135
5.6.6 高速化 135
5.7 モデリング変換 137
5.7.1 モデル座標系 137

第 X 部 **140**

5.7.2 さまざまな座標変換 148

5.7.3 ロボットの描画 153

第 XI 部 **157**

第 6 章 3 次元グラフィックス：面画 **158**

6.1 三角形のラスタライズ 159

6.2 隠面消去 160

6.2.1 アイディア 161

6.2.2 Z バッファ法 162

6.2.3 重心座標系による線形補間 164

第 XII 部 **166**

6.3 陰影付け 167

6.3.1 拡散反射 168

6.3.2 環境反射 171

6.3.3 鏡面反射 172

6.3.4 自発光 176

第 XIII 部 **177**

6.3.5 様々な光源 178

6.4 3D CG アニメーション 180

6.5 陰影づけの線形補間 182

6.5.1 フラットシェーディング 182

6.5.2 スムーズシェーディング 182

第 XIV 部 **186**

6.6 ファイルを用いた描画 187

6.6.1 環境設定 187

6.6.2 画像の描画 187

6.6.3 形状定義ファイルの読み込み 193

6.6.4 様々な形状の定義 194

6.6.5　テクスチャマッピング 195

第 XV 部　198

第 7 章　その他の話題　199

7.1　グラフィックスプロセッシングユニット 199

7.2　レイトレーシング 201

第1章　はじめに

1.1　授業の概要

　このテキストは、佐賀大学理工学部の選択科目「コンピュータグラフィックス演習」および知能情報システム学科の選択科目「コンピュータグラフィックス」のための資料です。本学のカリキュラムにマッチするように書かれています。

　この授業では、コンピュータを用いて画像を作成する手法 — いわゆるコンピュータグラフィックス（以下、CG と略記）について講義と演習を行います。内容は非常に初歩的な画像の話から始め、2 次元グラフィックス、3 次元グラフィックスへと進めて行きます。CG について自学している人にとってはすでによく知っている内容が多く、あまり面白い授業にはならないかもしれません。原則、対面授業を行います。数週間に 1 回、演習課題を課す予定です。

　最近ではハリウッド映画にありそうな美しいキラキラした 3D CG を簡単に作ることができるソフトウェア（しかもフリーのもの、たとえば Blender など）が多数あります。しかしこの授業ではそのようなソフトウェアは一切用いず、p5.js という JavaScript の上に作られた最低限の CG 機能のみが提供されるプログラミング環境を用い、CG の基本技術をひとつひとつ学んでいきます。

1.2　成績評価

　まず、コロナ禍の状況によっては以下に述べる内容を変更することもありうるとあらかじめ了解してください。

　講義は原則、対面授業で進めます。講義資料（pdf ファイル、様々な解説動画）は Microsoft Teams にアップロードします。

　成績は以下の 3 項目で評価し、0 点〜100 点の成績を付ける予定です。

毎週の小テスト 以下の二つからなります。

- **確認テスト** その週の授業内容について簡単なオンライン試験を実施します。成績点を 1 点（満点の場合）とします。
- **疑問、質問、感想の入力** 授業内容について疑問、質問、感想を記入してください。成績点を 1 点（ただし 20 字以上入力していること）とします。

授業は全部で 15 回ありますから、小テストの成績点の合計は 30 点です。

演習点 ... 画像作成に関する演習 1 回または 2 回を予定しています。30 点の配点を予定しています。

期末試験点 ... 期末試験に 40 点の配点を予定します。

出席回数は成績評価に用いませんが、対面授業での出席が 60%以下の場合、質問点（満点で 15 点）を減点します（減点率は検討中)。

オンライン試験は以下から入ってください。

第2章　グラフィックス環境の準備

この授業では CG に関する知識を修得することに加え、実際に自分自身で CG 描画のプログラムを作成することも重視します。そのためのプログラム開発環境としてプログラミング言語 p5.js を用います。

2.1　プログラミング言語 p5.js

p5.js はプログラミング言語 Processing をプログラミング言語 JavaScript の上に移植したものです。

その Processing は、MIT（Massachusetts Institute of Technology、マサチューセッツ工科大学）メディアラボが開発したプログラミング言語（およびプログラム開発環境）です。CG を含む電子的ビジュアルアートを簡単にプログラミングできることも目的に開発されており、実際、他のプログラミング言語、環境を用いた場合よりも簡単に CG プログラムを記述できると評価されています。Processing の問題点は、その言語処理系がみなさんのノート PC で動かない場合は多々あるという点です。そのせいで以前の授業で苦労しました。特に 3D CG の描画プログラムはメモリ、CPU を多量に消費するため、性能が十分でない PC ではプログラムが動かない場合があります。JavaScript を用いる p5.s ではこの問題が軽減されていると思われますから、この授業では p5.js を採用することにしました。

JavaScript は、Web ブラウザの上で動作するオブジェクト指向スクリプト言語です。スクリプト言語とは言語構造が比較的単純であって、小さな問題を手軽にプログラミングすることに向いているものを指します。スクリプト言語の対極にあるのはプログラムを堅牢に構築することを目指すタイプのプログラミング言語で、たとえば C++などがそれに相当します。ただし JavaScript について言えば、この言語は必ずしも単純な言語ではなく、スクリプト言語でありながらも高機能を有する言語になっています。

2.2　動かしてみる

p5.js のホームページは、「p5.js」でネット検索すれば上位に現れます。以下が、大元の英語版の URL です。

図 2.1: p5.js ホームページのトップページ

```
https://p5js.org
```

日本語版も用意されており、その URL は以下です。

```
https://p5js.jp
```

このページの様子は図 2.1 の通りです。

p5.js のプログラムを試しに動かしてみるために、**みなさんの PC に p5.js の言語処理系をインストールする必要はありません**。図 2.1 の下の方、赤枠の部分「新しい p5.js Web エディタで今すぐ試してみてください」をクリックしましょう。そうすると、新たに図 2.2 のようなウインドウが開きます。

図 2.2 の左側半分がエディタ、右側半分が実行画面です。2021 年 9 月時点では、エディタにはすでに以下のプログラム：

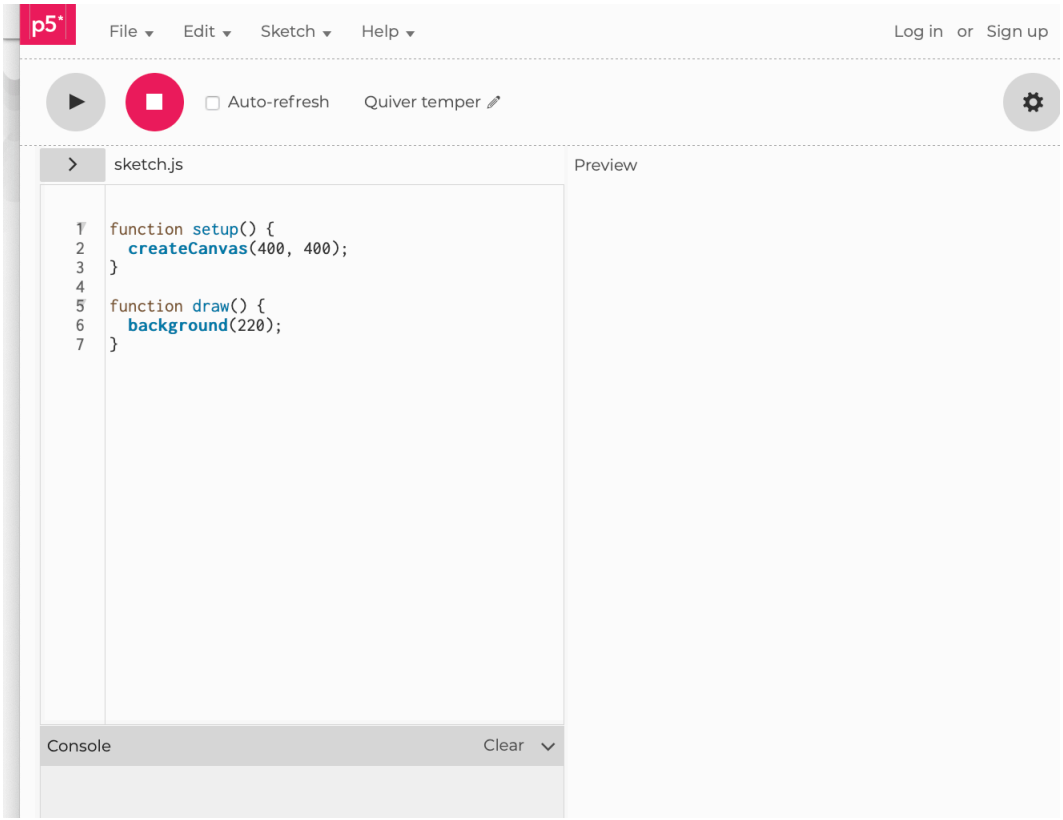


図 2.2: Web エディタ

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
}
```

が載せられています。

　このプログラムは、関数 `setup()` でキャンバス（CG 画像を描くオブジェクト）を初期化し、関数 `draw()` で CG 画像を描画します。`draw()` は 1 秒間に数十回[1] 呼び出されるため、`draw()` で描画する画像を少しずつ変えていくと、CG アニメーションを作成可能です。上のプログラムでは、横 400 画素 × 縦 400 画素の大きさのキャンバスを作り（`createCanvas(400, 400)`）、そのキャンバスの全画素を明るさ 220 の灰色で塗りつぶします（`background(220)`）。同じ色で塗りつぶすだけのため、画面は全く変化しません。

　ともかくさっそく動かしてみましょう。ページ左上の のボタンを押します。そうすると、

[1] 正確な数値は後に解説します。

ページが図 2.3 のように変化するはずです。ページ右側に 400×400 の画素の大きさの灰色の正方形が描画されています。画面は静止したままですが、上の述べたように、実はアニメーションを実行中です。実行を止めるには、ページ左上の のボタンを押します。

次に、アニメーションの実行を実感できるプログラムを実行してみましょう。先ほどのプログラム中の関数 `draw()` を以下に変更します。

```
let i = 0;
function draw() {
  background(i,100,50);
  i = (i+5)%256;
}
```

これは、大域変数 `i` の値を 5 ずつ増やしつつ、256 の剰余を計算し（つまり、`i` の値は、0, 5, 10, ..., 250, 255, 4, 9, ... と変化します）、RGB 値[2] $(\mathtt{i}, 100, 50)$ で定義された色で正方形を塗りつぶすというものです。上のプログラムを実行すると、正方形の色が繰り返し変化していく様子（図 2.4 参照）が見えるはずです。

Web エディタのその他の機能を以下に概説します。まず、メニューについてです。なお、メニューのコマンドリストは状況によって変化することがあります。以下は、初期状態での説明です。

File ... プログラムファイルに関するコマンドメニューです。

- **New** ... エディタで編集中のプログラムを初期化します。
- **Save** ... プログラムを保存します。保存先は PC ではなく、p5.js のサイト・サーバーです。よって、事前に ID を作り、サインインしておく必要があります。
- **Examples** ... 様々なサンプルプログラムをエディタにロードできます。

Edit ... 編集に関するコマンドメニューです。以下のコマンドはショートカットキーで起動すると便利でしょう。

- **Tidy Code** ... プログラムを整形します。
- **Find** ... 文字列検索を行います。
- **Find Next** ... 次の文字列を検索します。
- **Find Previous** ... 直前の文字列を検索します。

Sketch p5.js では、プログラムのことを**スケッチ**（sketch）と呼びます。ここで図 2.2 のエディタウインドウの の直下の をクリックしましょう。そうすると、そこに

[2] RGB 値についても後に授業で解説します。

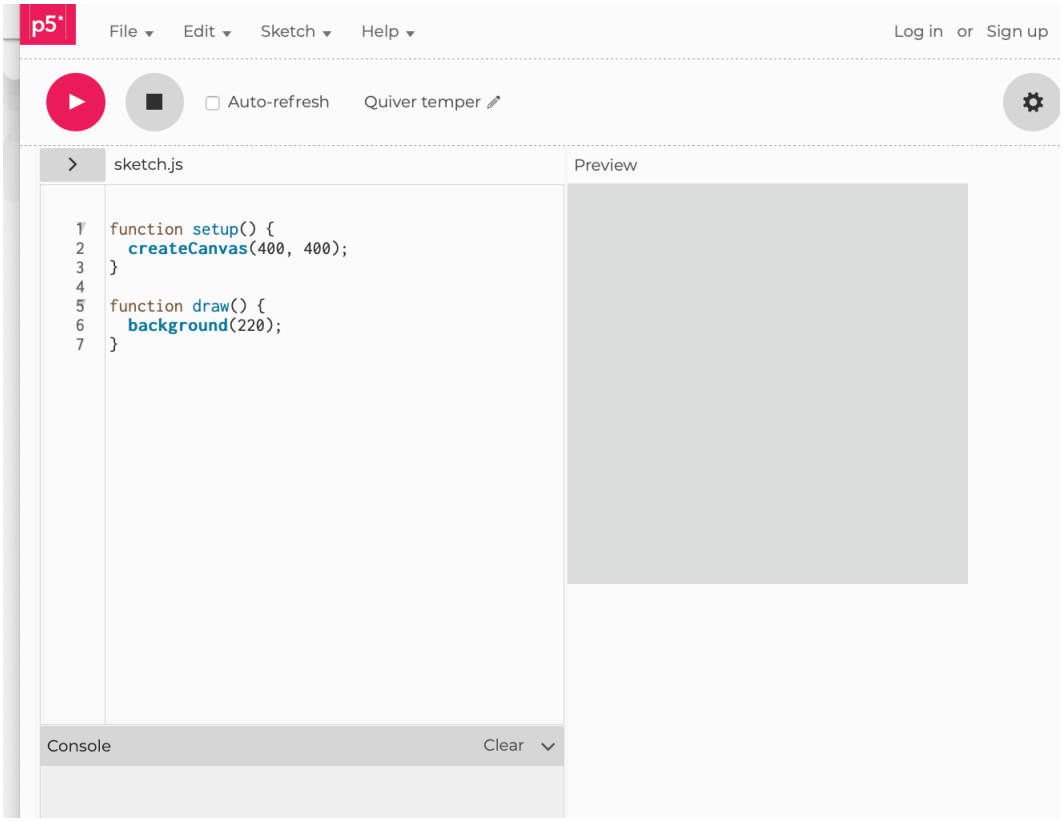

p5*
File
Edit
Sketch
Help
Log in or Sign up
Auto-refresh
Quiver temper
sketch.js
Preview
function setup() {
createCanvas(400, 400);
}
function draw() {
background(220);
}
Console
Clear

図 2.3: 実行中画面

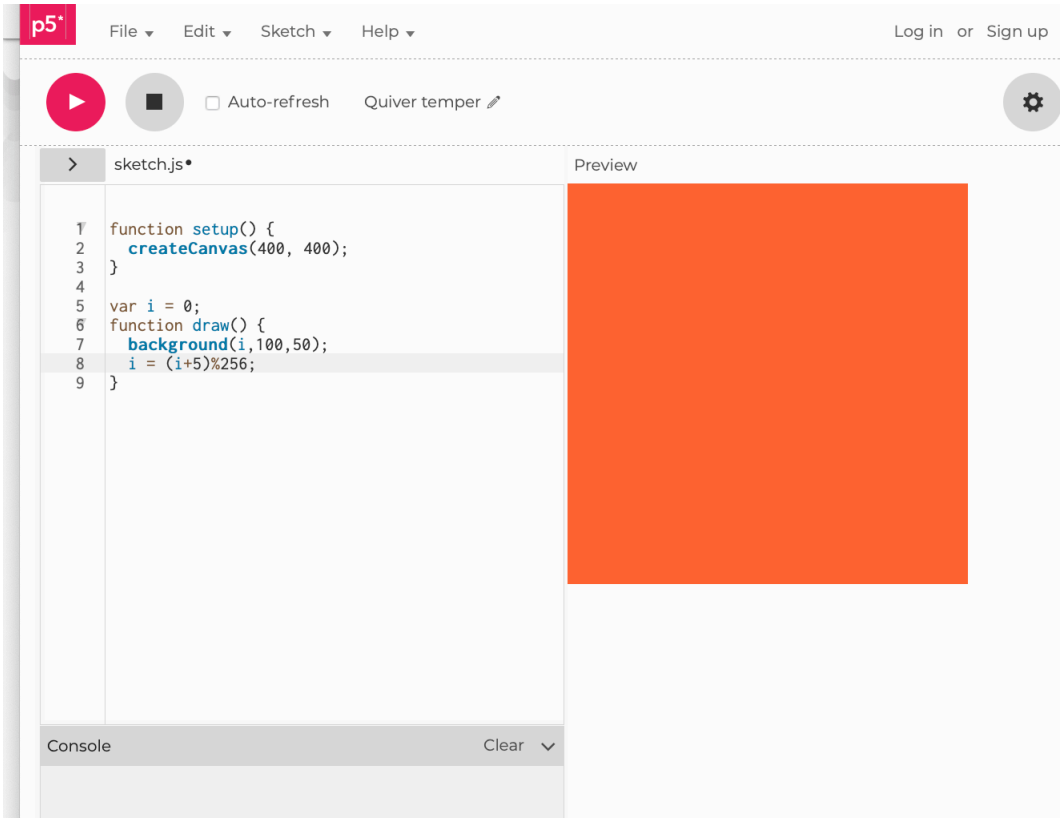
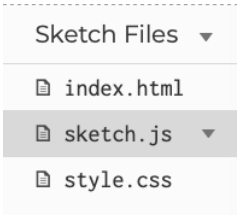

p5*
File
Edit
Sketch
Help
Log in or Sign up
Auto-refresh
Quiver temper
sketch.js•
Preview
function setup() {
createCanvas(400, 400);
}
var i = 0;
function draw() {
background(i,100,50);
i = (i+5)%256;
}
Console
Clear

図 2.4: 実行中画面



というファイル名の並びがで表示されます。実は p5.js のプログラム（スケッチ）はこの三つのファイルから成っています。sketch.js がプログラムの本体です。サフィックス「.js」が示すように、これは実は JavaScript のプログラムです。index.html はプログラムが動く環境を設定している html ファイルです。Web ブラウザはこのファイルを読み込んで、このファイルに記載されている “sketch.js” を実行します。style.css はページの見た目を指定するスタイルシートです。

Sketch メニューの以下のコマンドは、プログラムの構築や実行に関するものです。

Add File プログラム実行環境（三つのファイル群）に新たにファイルを追加します。

Add Folder 同上、フォルダーを追加します。

Run プログラムを起動します。 と同じ機能です。

Stop プログラムの実行を終了します。 と同じ機能です。

Help 各種のサポートコマンドです。

Keyboard Shortcuts このエディタで使用できるキーボードショットカットの表示と編集が可能です。

Reference p5.js のホームページを新しいウインドウで表示します。

About p5.js の関連ページのリストを表示します。

これらのコマンドの使い方、その他のコマンドについてはテキスト中で適宜、取り扱うこととします。

第II部

第3章　コンピュータグラフィックスの基礎知識

この章では CG 描画を行うための基礎知識について解説します。CG 画像を作るために当然知っておくべき知識と考えてください。

コンピュータを用いて画像を定義・加工・表示する方法には**ベクタ・グラフィックス**（vector graphics）と**ラスタ・グラフィックス**（raster graphics）の二つの方式が知られています。これらの方法は以下の 2 種類の技術においてそれぞれの個性を持っています。

1. そのデータ形式に従って画像を定義し、加工する技術
2. そのデータ形式の画像をその専用装置を用いて表示する技術

この章ではこれらについて解説していきます。

3.1　ベクタ・グラフィックス

「ベクタ」とは**ベクトル**（vector）のことです。ただし、ここでいうベクトルは高校で学ぶベクトル — 大きさと方向を持った量 — ではなく、2 次元座標系上に始点と終点が与えられた**線分**（line element）と考えてください。ベクタ・グラフィックスでは多数のベクトル（線分）を用いて画像を定義します。

たとえば図 3.1 は線分の組み合わせで立方体を描画したものです[1]（5 章参照）。また図 3.2 はドローイングツール[2]を用い、円、直線、直方体を組み合わせて時計のイラストを描いたものです。狭義にはベクタ・グラフィックスは線分のみを扱いますが、線分が始点と終点で幾何学的に厳密に定義できるように円は中心点と半径で幾何学的に定義できるという点では類似しているため、広義には円、円弧などの曲線もベクタ・グラフィックスに含めます。

図 3.1 はいわゆる**線画**（line drawing）です。図 3.2 は、線分（と曲線）で定義された円、直方体の内側が特定の色で塗りつぶされているため、特に**面画**（surface drawing）と呼ばれます[3]。面画も幾何学的に厳密に定義できるという意味ではベクタ・グラフィックスに含めて考えます。

[1] この図が立方体に見えないのは、本来物体の裏側に位置し、見えないはずの稜線が描かれているためです。線画において見えない線を消去する方法（陰線消去）が知られていますが、かなり面倒な処理になります。

[2] この図は実際に Adobe Illustrator で描きました。

[3] 面画という呼称は一般的ではありませんが、点画（あるいは点描）、線画、面画という系列上に定義された用語として CG で用いられます。

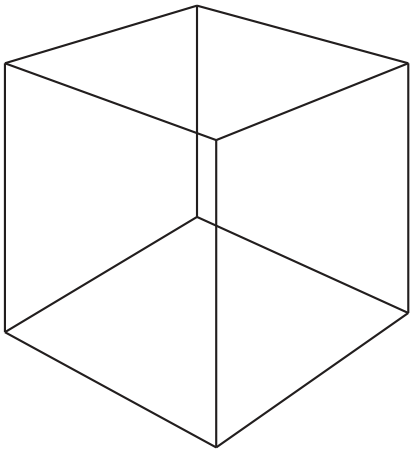


図 3.1: ベクタ・グラフィックス画像の例（その 1）

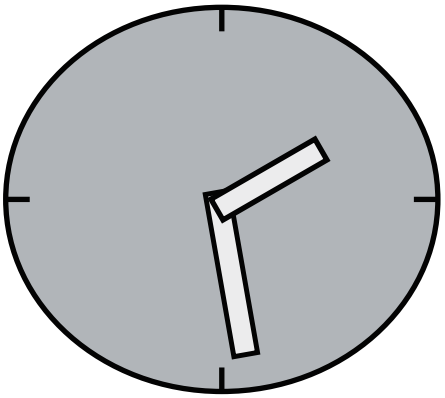


図 3.2: ベクタ・グラフィックス画像の例（その 2）

この章の冒頭に挙げた「そのデータ形式に従って画像を定義し、加工する技術」としてのベクタ・グラフィックスは重要です。今後ともその重要性が減ることはないでしょう。代表的例は以下の通りです。

フォントの形状： コンピュータ・ディスプレイに様々なデザインの美しい文字を表示するために活字の書体データ（＝デジタルフォント・データ）が必要です。そのデータとしてベクタ・グラフィックスが用いられています。ベクタ情報はフォントの拡大縮小に容易に対応できる点[4]で優れています。たとえば「グ」という文字を順にサイズを変えて表示すると、以下のように、フォントサイズに依らず同じ品質の美しい文字を表示できます。

゙グ゙グ゙グ゙グ゙グ゙グ゙グ゙グ゙グ

これは「グ」の形状が表示サイズに依存しない幾何学的な情報として管理されているためです。

ドローイングソフト： マウスなどのポインティングデバイス（pointing device）を用いてコンピュータで図形を描画するソフトウェア（たとえば Adobe Illustrator）の内部描画データはベクタ情報として生成・加工されます。たとえば図 3.1、図 3.2 はともに Adobe Illustrator を用いて作成しました。その図 3.1 を svg（Scalable Vector Graphics）と呼ばれるベクタ画像形式で保存した場合、そのファイルの内容は図 3.3 の通りです。この内容の、たとえば上から 10 行目の

[4] ベクトルデータを拡大縮小するには、単に座標値を座標変換すればよいだけです。

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 14.0.0, SVG Export Plug-In . SVG Version: 6.00
    Build 43363)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/
    DTD/svg11.dtd">
<svg version="1.1" id="レイヤー_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink
    ="http://www.w3.org/1999/xlink" x="0px" y="0px" width="595.28px" height=
    "841.89px" viewBox="0 0 595.28 841.89" enable-background="new 0 0 595.28 841.89"
    xml:space="preserve">
<line fill="none" stroke="#221E1F" x1="288" y1="193.889" x2="288" y2="337.889"/>
<line fill="none" stroke="#221E1F" x1="288" y1="337.889" x2="378" y2="274.889"/>
<line fill="none" stroke="#221E1F" x1="288" y1="193.889" x2="378" y2="157.889"/>
<line fill="none" stroke="#221E1F" x1="378" y1="157.89" x2="378" y2="274.889"/>
<line fill="none" stroke="#221E1F" x1="288" y1="193.889" x2="189" y2="157.889"/>
<line fill="none" stroke="#221E1F" x1="189" y1="157.889" x2="279" y2="130.889"/>
<line fill="none" stroke="#221E1F" x1="279" y1="130.889" x2="378" y2="157.889"/>
<line fill="none" stroke="#221E1F" x1="189" y1="157.889" x2="189" y2="283.889"/>
<line fill="none" stroke="#221E1F" x1="288" y1="337.89" x2="189" y2="283.889"/>
<line fill="none" stroke="#221E1F" x1="279" y1="130.889" x2="279" y2="229.889"/>
<line fill="none" stroke="#221E1F" x1="279" y1="229.889" x2="189" y2="283.889"/>
<line fill="none" stroke="#221E1F" x1="279" y1="229.889" x2="378" y2="274.889"/>
</svg>
```

図 3.3: 図 3.1 の svg ファイル

```
x1="288" y1="193.889" x2="288" y2="337.889"
```

はひとつの線分の始点と終点を定義している箇所です。ここの四つの数値を以下のように 1/5 倍：

```
x1="57.6" y1="38.7778" x2="57.6" y2="67.5778"
```

した場合、線分の長さは 1/5 倍になります。次に数値を 5 倍すれば、図形の形状は元に戻ります。このように、ベクタデータでは幾何学情報をそのまま保持するため、図形の拡大縮小、移動、回転などの変換操作を行なっても[5]、画像の劣化は生じません。

これらの優れた性質を、後に紹介するラスタ・グラフィックスで代替することは困難です。

次に「そのデータ形式の画像をその専用装置を用いて表示する技術」について述べると、ベクタ・グラフィックスの専用表示装置は現在、ほとんど利用されていません。ベクタ（線分）情報をそのまま表示できる表示装置として過去には以下のような表示装置がよく利用されましたが、現在はほとんど廃れています。

蓄積管（storage tube）： 通常のブラウン管[6]では、電子が蛍光面に照射されたときに照射された点がごく短時間だけ発光し、その点の光はすぐに消えます。これに対して蓄積管では、照射

[5]ただし、演算に伴う浮動小数点数の数値誤差は無視します。
[6] 液晶ディスプレイに置き換えられる以前のテレビを思い出しましょう。

図 3.4: 蓄電管ディスプレイ端末の例（ウィキペディア 「Direct-view bistable storage tube」（英語版）の図を転載）

図 3.5: X-Y プロッタ装置の例（ウィキペディア 「Plotter」（イタリア語版）の図を転載）

された点が永続して発光し続けるように蛍光面が加工されています。これを利用すれば、蓄積管上に多数の線（直線、曲線）を描画していくことが可能です。描画終了後に蛍光面のハードコピーを取る専用プリンタも開発されました。図 3.4 がその製品例です。蓄積管は、1980 年代前半まで多くの計算機センター、企業で利用されていました。画像を半導体メモリに記憶するのではなく、ブラウン管の管面状態として記憶するため、半導体メモリが高価であった時代に安価な装置として利用されました。

X-Y ペンプロッタ： 水平台に張り付けた紙面の上にペン先を接地させ、コンピュータ制御のモーターでペンを水平 2 方向（X-Y 方向）へ移動させて線画を描く装置です。蓄積管と同様に 1980 年代まで広く利用されました。低価格であったため、アマチュア愛好家や大学研究室などで広く利用されました。図 3.5 がその製品例です[7]。図の台の上のバーが左右に動き、さらにバー上のペンが前後に動くことで、台上の前後左右の全ての座標点へペンを移動できる仕組みです。ペンを換装することで多色の描画も可能です。

これらの表示装置は線分しか表現することができず、表現力は限られています。たとえば図 3.2 では楕円部分が灰色、超短針が薄い灰色で塗られていますが、蓄電管も X-Y プロッタも「面を塗る」という描画機能を持ちませんから、この図を描くことができません。 せいぜい領域全体にハッチ

[7] ちなみに、この図の装置は現在もイタリアの会社で販売されています。主用途は、線画の描画というよりも、ペンの代わりにカッターを装着することで革、布などを裁断する装置です。1980 年代に研究室等に普及した装置は、机に載るような小型のもので、価格は 5 万円〜20 万円程度でした。

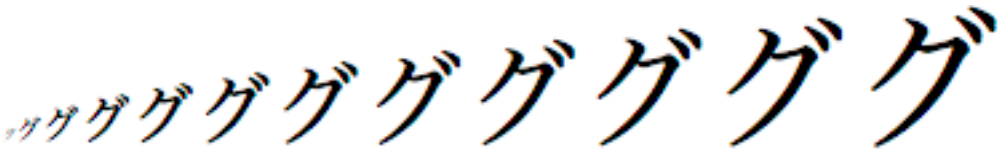
ング・パターン[8]を描くことくらいです。上記のようなベクタ・グラフィックス型の表示装置の画像表現能力は後に述べるラスタ・グラフィックス型の表示装置に比べ相当に劣るため、今やほとんど利用されていません。例外的にベクタ情報を表示する装置の現代的な方法として以下の例が知られていますが、この種の需要は限定的です。

レザリアム（Laserium[9]）： コンピュータ制御で高速に角度を変える鏡面を使ってレーザー光線の軌道を変化させ、プラネタリウムなどの大規模スクリーンに複雑な線画のアニメーションを映写する装置です。独特の表現ができるため、アート、エンターテインメントで利用されています。

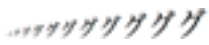
3.2　ラスタ・グラフィックス

ラスタ・グラフィックスでは、画像を多数の点の集合として取り扱います。テレビ、コンピュータディスプレイがまさにこの方式で映像を表示しています。テレビとコンピュータディスプレイの違いは、前者はアンテナから入ってきた映像信号を単に時系列で表示するのに対して、後者はコンピュータのメモリに格納されている内容を映像として表示する点です。そのために、コンピュータディスプレイには画像情報を保持する大容量のメモリが必要であり、それがラスタ・グラフィックス型表示装置の普及の大きな障害となってきました。

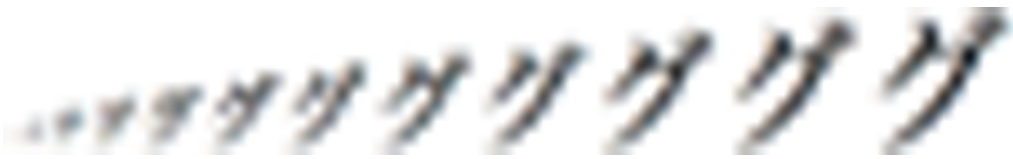
「そのデータ形式に従って画像を定義し、加工する技術」としてのラスタ・グラフィックスは単純です。画像点の情報が順に並んだだけのきわめて単純・平坦な形式です。その単純さと引き換えに、ラスタ・グラフィックスは画像の加工に適しません。たとえば 3.1 節のフォントの例「ググ...グ」を下図の画像（ラスタ・グラフィックスのデータ）に変換したと仮定します。



上の図は 3.1 節の元図と大差ありません。次にこの画像を縦横 1/5 の大きさへ縮小変換してみます（下図）。



さらに縦横 5 倍の大きさへ拡大変換し、元の大きさへ戻すと以下の通りです。



[8] 斜線や交差線などの一様な模様を繰り返し描いて、領域に濃淡や淡い色を表現する手法。

表 3.1: ベクタ・グラフィックスとラスタ・グラフィックスの比較

	取り扱う対象	定義・加工の容易さ	装置の表示能力（画像の品質）
ベクタ・グラフィックス	図形	○	×
ラスタ・グラフィックス	画像	△	○

元の画像に戻らないことが分かります。画像サイズを 1/5 に縮小した段階で画像情報が大きく欠損したため、拡大操作でサイズを元に戻しても欠損した情報を取り戻すことはできません。画像情報を、高次構造を持たない単純な点の集合で保持する方式はデータの加工に不適なのです。

次に、「そのデータ形式の画像をその専用装置を用いて表示する技術」としてのラスタ・グラフィックスでは、点の情報を順に表示すればよく、表示装置の基本構造は単純です。後に正確に述べますが、表示装置上の画像点の密度のことを**解像度**（resolution）と呼びます。表示装置が十分な解像度を持つならば非常に高品位な画像を表示できます。問題は十分な解像度を達成できるか否かです。ブラウン管、液晶パネル、そして最近では有機 EL パネルの解像度の向上、画像情報を保持するメモリの低価格化が進み、1980 年代半ば〜90 年代初頭になってようやくラスタ・グラフィックスの解像度がベクタ・グラフィックスのそれを超え、それを境にラスタ・グラフィックス装置が急速に普及していきました（3.4.6 節参照）。

ここでベクタ・グラフィックスとラスタ・グラフィックスの違いをまとめておきます。

表 3.1 に、(1) 画像を定義・加工する作業の容易さと、(2) 表示装置が高品位な画像を表示する能力について適/不適をまとめました。

ベクタ・グラフィックスが取り扱う対象は、簡単に言えば、線や面で構成された**図形**（shape、figure）です。その内部データ形式は線や面などの幾何情報の構造を持ちます。既に述べたように、デジタル・フォントやドローイングツールに活用されており、現在のコンピュータに不可欠になっています。幾何情報を操作することで画像（図形）の加工は容易です。しかし、線のみで画像（図形）を表示する装置では領域の塗りつぶしすらできず、フォトリアリスティック（photorealistic）な表現はほとんど不可能です。そのような装置は、特殊用途を除けば、ほとんど廃（すた）れています。

ラスタ・グラフィックスが取り扱う対象は、図形や写真画像などの広い範囲の、いわゆる**画像**（image）です。その内部データ形式は点の集合体という単純なものであって、構造は有しません。単純であるため取り扱い易いものの、高度な加工・変形には適しません。逆に、表示装置は多数の点を表示できればよいため、点の数さえ十分に多ければ、きわめて高品位な表示が可能です。

これらの特徴から、現在の CG では

```
function setup() {
  createCanvas(400, 400);
  background(220);
  circle(200,200,300);
  rect(100,100,200,200);
}
```

図 3.6: 円と正方形のラスタライズ：関数 `draw()` は削除してください。

1. 画像（および図形）のデータ形式としてはベクタ・グラフィックス、ラスタ・グラフィックスの両方式がそれぞれの特徴に応じて使い分けられ、

2. 表示においてはラスタ・グラフィックスで行う

ことが一般的です。表示の際にはベクタ・グラフィックスのデータはラスタ・グラフィックスの画像データへ変換 – この操作を一般に**ラスタライズ**と呼びます– され、ラスタ・グラフィックスの表示装置へ出力されます。

プログラミング言語 p5.js では、線分を描く、円を塗りつぶすなどのベクタ形式の描画関数を利用できますし、ラスタ形式の描画関数も用意されています。キャンバスはラスタ形式ですから、関数を実行した後の画像形式はラスタ形式のみが保持されます。たとえば線分を描画する関数が実行されると、関数内で線分がラスタライズされ、キャンバス内の適切な画素を塗ります。

考察　図 3.6 のプログラムを実行し、画像を確認しなさい。これは、円と正方形をラスタライズしてキャンバスに描画します。プログラムの意味の詳細は後章で解説します。

ベクタ、ラスタいずれの方式で CG 画像を生成するにせよ、最終的にラスタ・グラフィックスの出力を行うため、CG を学ぶにはラスタ・グラフィックスの理解が必須になっています。そこで次にラスタ・グラフィックスについてやや詳しく解説します。

第III部

表 3.2: 代表的な画像サイズの規格

規格名	水平方向の画素数 M	垂直方向の画素数 N	総画素数	備考
VGA	640	480	307,200	1990 年代の PC
NTSC	720	480	345,600	アナログ世代のテレビ（日本）
HDTV	1,920	1,080	2,073,600	ハイビジョン
WUXGA	1,920	1,200	2,304,000	最近の PC
4K UHDTV	3,840	2,160	8,294,400	4K テレビ
8K UHDTV	7,680	4,320	33,177,600	8K テレビ

3.3 画像情報とフレームメモリ

3.3.1 画素

既に述べたように、ラスタ・グラフィックスでは画像情報を点の集合として表わします。各点の情報を一般に**画素**、**ピクセル**（pixel）と呼びます。画素は一般には色情報を保持しますが、その他の情報を補助的に付加される場合もあります。画素の詳細は 3.4 節で述べます。

3.3.2 画像サイズ

画像は多数の画素を矩形に整列させたもので、それをそのままコンピュータ・ディスプレイに矩形に表示するのが一般的です。その矩形の大きさ（画像サイズ、画面サイズ）を画素の数で表わすとき、画素が水平方向（横）に M 個、垂直方向（縦）に N 個並んでいるならば、その画像の総画素数は $M \times N$ 個です。各種出力装置との接続のために画像サイズには一定の規格が作られています。代表的なものについては表 3.2 の通りです。

画像サイズと混同しやすいのが**画面解像度**（display resolution）、あるいは単に解像度です。画面解像度は表示装置の画素の密度のことです。これは表示装置の特徴量であって、本来、画像サイズとは独立した量です。しかし大小様々な画像を表示装置（たとえばスマーフォン）上で閲覧する場合、見かけ上の画素の密度はしばしば画像サイズに比例する（サイズの小さな画像は拡大して見る、サイズの大きな画像は縮小して見る）ため、解像度と画像サイズは同一視されがちです。しかも同一視しても大抵の場合に大きな混乱を生じません。

必要十分な画面解像度　表示装置にはどの程度の解像度が必要なのでしょうか。

画像は最終的に人の目で見られるものであるため、目の解像力がそれを決める基準となります。視力 1.0 の人が約 30cm 離れた印刷物を見るとき、1 インチ当たり 300 本〜400 本の密度で印刷された直線を識別できると言われています。この「1 インチ当たりの本数」は一般に dpi（dots per

inch) と表され、印刷解像度の単位となっています。ドット密度と呼ぶこともできます。つまり、300〜400dpi のドット密度があれば、人の目は表示された画像の荒さがそれほど気にならないと考えられます[10]。

ところで最近では、dpi に似た単位として ppi（pixels per inch）も用いられます。これは印刷物の単位ではなく、コンピュータディスプレイの表示解像度の単位です。画素密度と呼ぶこともできます。目の解像度との関係で言えば、dpi と ppi は同等に扱うことができると考えられています[11]。

今、仮に 350dpi=350ppi を人の目の限界解像度の目安としましょう。この目安を cm に換算すると

$$350 \text{ dpi} = 350 \text{ dots per inch} = 350 \text{ dots per } 2.54\text{cm} \approx 138 \text{ dots per cm}$$

となります。よって、ディスプレイの横幅が 30cm のノート PC を仮定するとき、そのディスプレイは水平方向に $138 \times 30 \approx 4,140$ 画素の解像度を持っていれば、人の目にとって十分な満足な解像度と考えてよいでしょう。現在市販されているノート PC でこれを超えるものはありませんが、最近、4K の解像度（水平画素 3,840）を持つものが現れてきています。現在、iPhone のディスプレイがこの解像度に達しています[12]。

なお、デジタルカメラが普及する以前のフィルムカメラに撮影されていた銀塩写真の解像度は 2000dpi 以上と考えられており、上の 300dpi よりも一桁高い解像度になります。その意味で、未だ銀塩写真で撮り続けているカメラマンがいることは納得できるところであり、全ての画像がデジタルに置き換わるにはもう少し時間が掛かるでしょう。

3.3.3 画素の順序

画像情報はコンピュータのメモリへ格納され、一定時間毎に（たとえば毎秒 30 回）映像信号へ変換され、信号はディスプレイへ表示されます。

画像情報は 2 次元構造、メモリは 1 次元構造であることに注意してください。2 次元のデータを 1 次元構造のメモリへ格納するために、画像に含まれる全画素には図 3.7 のような順序が決められており、メモリ中にこの順番に画素の情報が並んでいます。なお、この図は VGA（640×480）を想定しています。つまり、画素の並び順は

1. 画像の左上の画素を先頭とし、
2. 同じ行の画素を右へ向かって数え上げ、

[10] 感覚には個人差や個々人のこだわりが強く関係するため、このドット密度で十分という意味ではありません。

[11] 詳細に見れば、コンピュータディスプレイの画素の表現力は印刷物のドットの表現力よりも高いと思われるため、同等に扱うのではなく、N dpi の表現力は $N/2$ ppi に相当すると考える研究者もいるようです。

[12] iPhone X（2018）の画素密度は 458ppi です。現在、Apple 社は同社の高解像度ディスプレイの商標として「retina」を用いていますが、retina は日本語では網膜を意味しており、すなわち、Apple 社の Retina でディスプレイは網膜の認知力を超える画素密度を有することを主張している訳です。

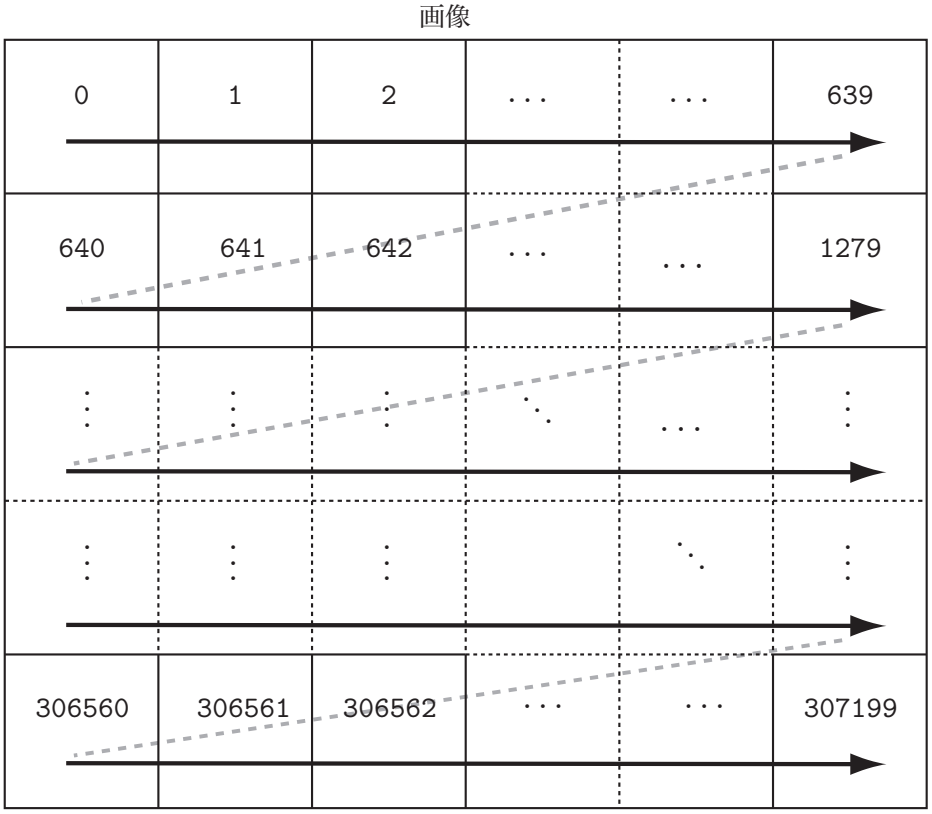


図 3.7: 画像内の画素の順序：VGA（640 × 480）の場合

3. その行が右端まで到達したならば、ひとつ下の行の左端から同様に数え上げ、

4. 画像の右下の画素を末尾とする

ような順序です。よって、右上の先頭画素を 0 番目とするならば、その行の末尾の画素は 639 番目となり、次の行の先頭画素は 640 番目となります。右下の末尾の画素は 30711（$= 640 \times 480 - 1$）番目という順番になります。

なお、この画素の順序付けは、CG 画像を生成するときにプログラマが念頭に置くべきデータ構造を示したものです。jpeg 画像のようなデータ圧縮された画像での画素の並びはこれとは全く異なります。圧縮画像のデータ形式はこの授業の対象外ですので解説を省略します。

3.3.4 フレームメモリ

メモリから映像信号を取り出す**ビデオ・ディスプレイ・コントローラ**（Video Display Controller、以下、VDC）は、図 3.7 の順番に画素情報にアクセスし、それを映像信号へ変換し、コンピュータ・ディスプレイに送り出します。VDC は 1 秒間に数十回から 100 回程度これを繰り返していま

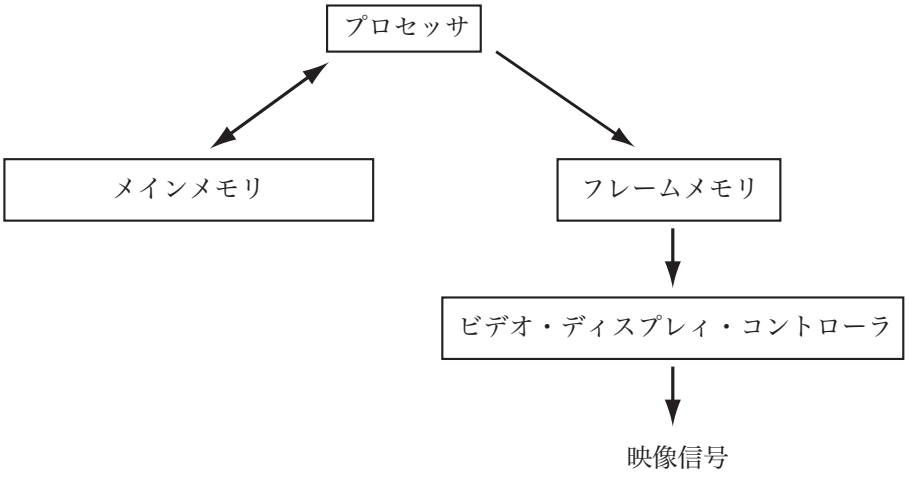


図 3.8: フレームメモリとコンピュータアーキテクチャ

す[13]。

通常、画素情報を保持するメモリはプロセッサ（CPU）に結合されたメインメモリ（main memory）ではありません。その理由として、メインメモリはプロセッサが演算に使用するものであり、それを VDC と共有することはアクセス競合の観点から非効率だからです[14]。画素情報を保持するメモリは通常、**フレームメモリ**、**フレームバッファ**、**ビデオメモリ**、**VRAM** などと呼ばれ、コンピュータ・アーキテクチャ的には図 3.8 に示すような位置づけになっています。図からも分かるように、フレームメモリはメインメモリとは分離されています[15]。またプロセッサからフレームメモリへデータを入力するポート（port、接続端子）とフレームメモリから VDC へデータを出力するポートを個別に設けることでプロセッサと VDC のアクセス競合を避ける仕組み[16]が実現されています。最近の多くのコンピュータでは、フレームメモリ、VDC をまとめて**グラフィックスプロセッシングユニット**（GPU）として実装し、高速化・高機能化しています。

3.3.5 X-Y 座標系

CG 画像を作るには、適当な数値計算プログラムを用いて各画素情報を求め、それをフレームメモリ内の特定の位置へ書き込みます。

[13] この繰り返しの頻度をリフレッシュレートと呼びます。
[14] メインメモリにはプロセッサと VDC の両方から同時にアクセスすることはできません。もしアクセスが同時になった場合には一方が待たされることになります。VDC が待たされるとディスプレイ上の映像が乱れる可能性があるため、同時アクセスの場合には VDC のアクセスが優先されます。そうなると、プロセッサによる処理が遅延する問題を生じます。
[15] フレームメモリのアドレスをメインメモリの未使用アドレスに組み込み、メモリ空間を統合する場合もあります。最近、プロセッサとグラフィックスプロセッシングユニットの協調による高速演算においてこの手法が見直されている。ヘテロジニアス・ユニファイドメモリーアクセス （heterogeneous Unified Memory Access、hUMA）などと呼ばれている。
[16] これをデュアルポートと呼びます。

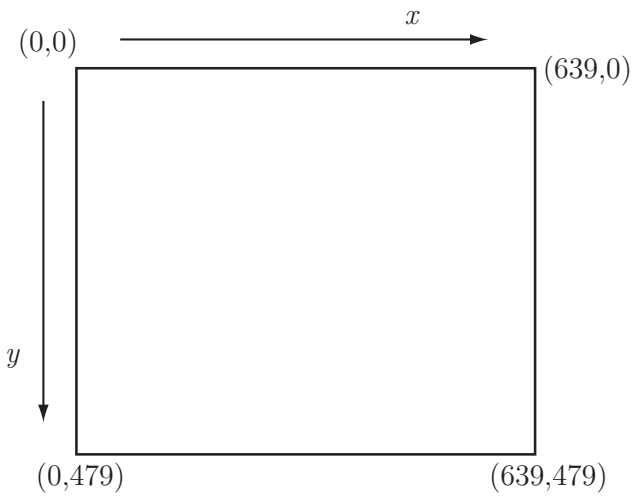


図 3.9: 画像の X-Y 座標系：VGA（640×480）の場合

画素の位置は図 3.7 の順序から計算すればよいのですが、やはり 2 次元座標系 (X, Y) を用いる方が便利です。通常の数学の座標系同様に、CG でも水平方向を X 座標とし、垂直方向を Y 座標とする 2 次元座標系を用いますが、しばしば

- CG では、Y の値は上から下へ向かって増える

点が通常の X-Y 座標系と異なっています（図 3.9 参照)。これは奇妙に思えますが、フレームメモリ内の画素の順番（図 3.7）が左上の点を起点にしているためです。その結果、CG の X-Y 座標系（図 3.9）では左上の点が座標原点 $(0, 0)$ になっています。コンピュータ・アーキテクチャと画像の座標系が整合していることは、それに関連するコンピュータの演算処理が効率的にできる点で重要です。つまり、画像サイズを $M \times N$ とするとき、座標点 (X, Y) の画素のフレームメモリ上の位置（順番）は積和演算式 $X + M \cdot Y$ で計算できるのです。ほとんど全てのプロセッサは積和演算を 1 命令で実行できるため、これは高速に演算可能です。これはコンピュータの演算速度が遅かった時代に特に重要でした。

CG の X-Y 座標系は通常の X-Y 座標系と上下が反転した関係にありますから、通常の感覚で描画すると、実際の画像は上下逆さまになってしまうことに注意してください。ただし、現在では通常の X-Y 座標系を用いたグラフィックスツール（たとえば Open GL など）も多数存在します。新たにグラフィックスツールを使用する場合には必ず事前にチェックしましょう。

3.3.6 寄り道：「ラスタ」の語源

ベクタ・グラフィックスの「ベクタ」は、既に述べたように、「ベクトル」のことです。私たちにとって馴染みがある言葉であり、ベクタ・グラフィックスが線画を意味することは十分納得でき

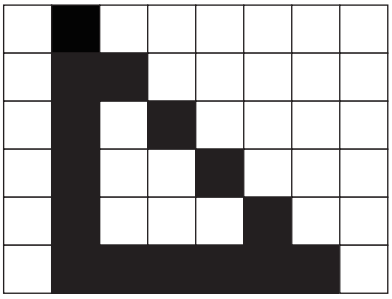


図 3.10: 2 値画像（画像サイズ 8×6）の例

ます。

これに対してラスタ・グラフィックスの「ラスタ」は馴染みのない言葉です。もともと「ラスタ」という単語は 19 世紀末から 20 世紀に掛けてのハーフトーン印刷手法（新聞写真を点で表現する手法）に関するドイツ人研究者の論文で使われ始め、それが後にテレビ技術の「ラスタ・スキャン」に延用されたと思われます[17]。そのドイツ語の raster は英語の screen に対応しており、ラスタ・グラフィックスとはスクリーン・グラフィックスと言い換えることができます。さらにドイツ語の raster の語源はラテン語の rastum（日本語訳＝熊手）を語源とするようです[18]。図 3.7 を見ると、左から右へ伸びる矢印がちょうど熊手で地面を掃いた跡に似ています。ドイツ人研究者が raster という術語を論文に用いた理由もその辺にあるのでしょう。

いずれにせよ、一旦「ラスタ」という語が現代的に定着してしまうと、その意味は語源から離れてしまいます。CG で「ラスタ」といえば、図 3.7 のような画素の並びのこと、対象物を点で表わすこと、あるいは点で表わされた対象物を意味します。

3.4 画素情報

前節では画素の並びについて種々解説しましたが、ここではその画素の中身（画素情報）について代表的なデータ形式を解説します。

3.4.1 2 値

最も単純な画素は黒または白の 2 色の情報を持ちます。画素当たりの情報量は 1bit で、通常、値 0 が黒、値 1 が白を表わします。この値のことを**画素値**（pixel value）と呼びます。このような画素を用いて構成される画像を**2 値画像**（binary image）、**モノクローム画像**（monochrome image）、あるいは略して**モノクロ画像**と呼びます。

[17] Wikipedia「ラスタースキャン」の項を参照。その他のネット情報でも同様の内容あり。
[18] 英語では rake、イタリア語では rastrello が熊手を意味します。また、五線譜の 5 本の線を同時に引く筆記用具（熊手に似ています）も、raster と呼ばれています。

図 3.11: 7 号館の 2 値画像（画像サイズ 600×460）の例

たとえば、図 3.10 は非常に小さな 2 値画像です（画像サイズ 8×6）。画素は、図 3.7 で述べたように左上から右下に向かって並んでいます。よって、この画像のデータを 2 進数で全て書き出すならば以下の通りです[19]。

```
101111111001111110101111101101111011101110000001
```

しかし上の表記法は読みづらいため、画素値と画素値の間を 1 マス空け、各行毎に改行して画素値を以下のように表わすこととします。

```
1 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
1 0 1 0 1 1 1 1
1 0 1 1 0 1 1 1
1 0 1 1 1 0 1 1
1 0 0 0 0 0 0 1
```

もうひとつの例として、図 3.11 に理工学部 7 号館の写真を 2 値化した画像を示します。

2 値を用いた画像表現のことをしばしば**ビットマップ**（bitmap）と呼び、2 値画像のことを**ビットマップ画像**（bitmap image）とも呼びます。

3.4.2 グレースケール

白と黒に加え、様々な明るさの灰色を持つ画素の場合を**グレースケール**（gray scale）と呼びます。一般にグレースケールには画素当たり 8bit のデータを割り当てます。つまり 256 段階の明暗を表わすことができます。明暗の区別に関する人間の認知能力は 256 段階よりもやや優れている

[19] メモリはバイト単位ですから、バイトの並びとして 16 進数表記すると `BF 9F AF B7 BB 81` となります。

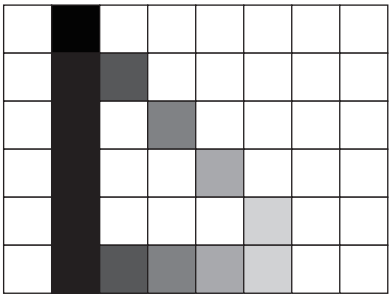


図 3.12: グレースケール画像（画像サイズ 8×6）の例

図 3.13: 7 号館のグレースケール画像（画像サイズ 600×460）の例

と言われています[20]が、8bit=1byte であり、現在のコンピュータで扱い易いため、グレースケールでは 8bit が標準になっています。画素値と輝度値の対応は、0 が黒を表わし、255 が白を表わし、1〜254 の値は値が大きいほど明るい色を表わします。

　たとえば、図 3.12 は、図 3.10 と類似のグレースケール画像です。各画素値を 2 桁の 16 進数で表わした場合の画像のデータが以下の通りです。

```
FF 00 FF FF FF FF FF FF
FF 00 33 FF FF FF FF FF
FF 00 FF 66 FF FF FF FF
FF 00 FF FF 99 FF FF FF
FF 00 FF FF FF CC FF FF
FF 00 33 66 99 CC FF FF
```

もうひとつの例として、図 3.13 に理工学部 7 号館の写真をグレースケール画像で示します。

[20] 出典：Korean Society of PACS Technology：医用画像 PACS、日本放射線技師会出版会 (2006).

図 3.14: 3 原色による加色混合

3.4.3 フルカラー

ある画素が、人間が認識できるほとんど全ての色を表現できるとき、それはしばしば**フルカラー**（full color）と呼ばれます。人間が様々な色を認識している仕組みに立ち入って議論すると、網膜、視神経の構造から始めねばならず、説明が長くなりますから、ここではその辺を全て省略しますが、CG では、赤色（Red）、緑色（Green）、青色（Blue）の **3 原色**（three primary colors）を適当な輝度で加色混合[21]して様々な色を作っています。加色混合はちょうど図 3.14[22]のような、赤、緑、青の光による色の混ぜ合わせに相当し、このようにして作られる色は 3 原色の頭文字を取って**RGB カラー**（RGB color）と呼ばれています。また以下では赤、緑、青の各色をしばしば簡単に **R**、**G**、**B** と略記します。

RGB カラーでは、一般に 3 原色のそれぞれの輝度に 8bit、計 24bit のデータを割り当てるため、**24bit カラー**とも呼びます。この場合、3 原色の各色は 256 段階の明暗を表わすことができ、計 1670 万（$= 256^3$）通りの色を表現可能です。過去にメモリが高価だったころには bit 数を抑えたデータ表現（たとえば各色 4bit = 16 段階の輝度を表す場合など）も使用されましたが、現在はフルカラーと言えば 24bit カラーのことを指すのが一般的です。1670 万色は人間が区別できる色数を越えているとされ、1 画素にこれ以上の bit を割り当てることはまれです[23]。データ表現では R、G、B の順に 3 原色の輝度値を並べるのが一般的です[24]。グレースケールの場合と同様、0 が最も低い輝度、255 が最も高い輝度を表わします。

たとえば、図 3.15 は、図 3.10、図 3.12 と類似の形状のフルカラー画像です。各画素値を 6 桁の 16 進数（R、G、B の各色を 2 桁の 16 進数）で表わした場合の画像のデータが以下のようになります。

[21] これに対比する言葉は減色混合です。絵の具の色の混ぜ合わせがそれに相当します。
[22] この図は Wikipedia「RGB」の項の図をそのまま引用した。
[23] より多くの bit 数を割り当てるべき例外として、画素値間の演算を行なう場合には数値誤差が生じることがありますから、中間データとしてより多くの bit を割り当てることがあります。また、グラフィックスデザイナーなどの色を専門的に扱うプロフェッショナルがより多くの色数を求める場合もあります。
[24] エンディアンの関係で、見かけ上、B、G、R の並びになっている場合もあります。

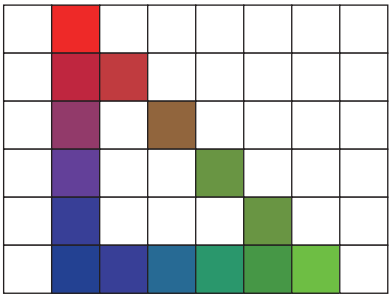


図 3.15: フルカラー画像（画像サイズ 8×6）の例

図 3.16: 7 号館のフルカラー画像（画像サイズ 600×460）の例

```
FFFFFF FF0000 FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF
FFFFFF CC0033 CC3300 FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF
FFFFFF 990066 FFFFFF 996600 FFFFFF FFFFFF FFFFFF FFFFFF
FFFFFF 660099 FFFFFF FFFFFF 669900 FFFFFF FFFFFF FFFFFF
FFFFFF 3300CC FFFFFF FFFFFF FFFFFF 33CC00 FFFFFF FFFFFF
FFFFFF 0000FF 0033CC 006699 009966 00CC33 00FF00 FFFFFF
```

ここに FFFFFF、FF0000、00FF00、0000FF はそれぞれ白色、赤色、緑色、青色を表わしています。また、たとえば上から 2 行目左から 2 列目の CC0033 は、少し暗い赤（CC）に暗い青（33）が混じった色（すなわち、少し暗い赤紫）を表わします。もうひとつの例として、図 3.16 に理工学部 7 号館の写真をフルカラー画像で示します。

3.4.1 節で、2 値画像表現のことをビットマップと呼ぶと述べました。これと対比して、フルカラー画像表現のことを**ピクセルマップ**（pixel map[25]）と呼びますが、最近ではフルカラー画像を含めたラスタ画像表現のことを総じてビットマップと呼ぶのが一般的です（フルカラー画像は本来

[25] または pixmap。

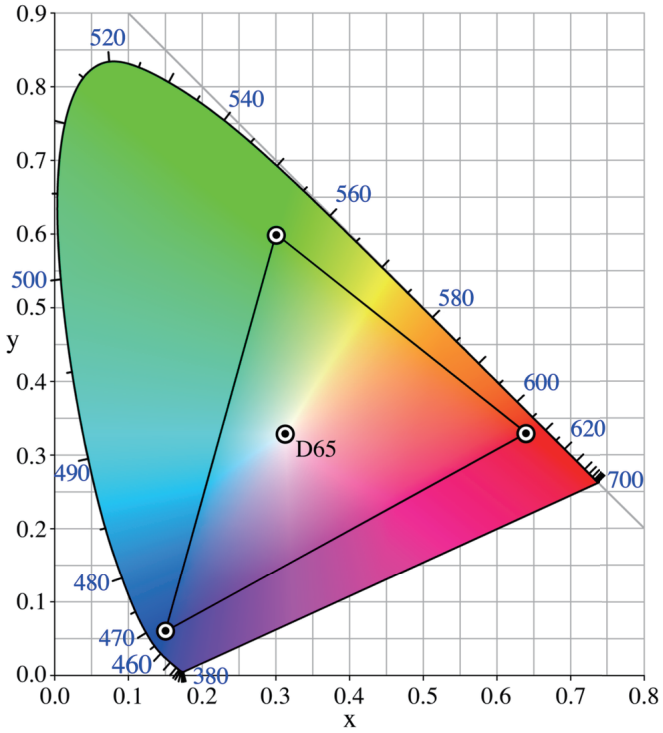


図 3.17: 色度図内の RGB カラー

ビットのマップではないのですが)。

RGB カラーの表現力　RGB カラーで作り出せる色が人間が目で認識できる全ての色ではありません[26]。図 3.17 に表示されている（馬蹄形の領域の）色は人間が認識できる全ての色（ただし明るい色のみ）を平面上に便宜的に表したものです[27]。そして、この図の 3 角形で囲まれた領域の色が RGB カラーで表現できる色です。逆に 3 角形の外側の色は RGB カラーでは表現できません[28]。しかし、3 角形の外側の色は、それと最も近い 3 角形の辺上の色で近似しても実用上、困らないと考えられています[29]。

　なお、図 3.17 には便宜上、明るい色のみを示しています。この図には載っていませんが、黒、灰色、薄暗い青、暗い赤紫などの色は、この図の色の輝度値を下げることで表現できることに注意してください。

[26] 先に「画素が人間が認識できるほとんど全ての色を表現できるとき」にフルカラーと呼ばれると述べましたが、「ほとんど」であって「全て」ではないことに注意してください。
[27] この図は Wikipedia「RGB」の項の図をそのまま引用しました。
[28] たとえばレーザー光などの単一波長の光は RGB カラーの基となっている視覚モデルでは取り扱うことが難しく、その色を正しく表現できません。詳細は各自で調べてください。
[29] 多くの人にとってはこの代替で問題のないレベルですが、芸術家、グラフィックスデザイナーなどの色彩を専門的に扱う人の中には不満を持つ場合もあるようです。

```
function setup() {
  createCanvas(400, 400);
  background(0);
  noStroke();
  blendMode(ADD);
  fill(255, 0, 0);          // ... 赤
  circle(150, 250, 150);
  fill(0, 255, 0);          // ... 緑
  circle(250, 250, 150);
  fill(0, 0, 255);          // ... 青
  circle(200, 164, 150);
}
```

図 3.18: 3 原色による加色混合（図 3.14 参照）をプログラムで描画するプログラム：関数 `draw()` は削除してください。

考察　図 3.14 は p5.js のプログラムでも作ることができます。図 3.18 を実行してみてください。関数 `fill()` の三つの引数は、赤、緑、青の輝度値を意味します。0 が最小、255 が最大輝度です。その他の関数意味の詳細は後章で解説します。

3.4.4　インデックスカラー

図 3.15 のフルカラー画像データは以下の通りでした。

```
FFFFFF FF0000 FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF
FFFFFF CC0033 CC3300 FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF
FFFFFF 990066 FFFFFF 996600 FFFFFF FFFFFF FFFFFF FFFFFF
FFFFFF 660099 FFFFFF FFFFFF 669900 FFFFFF FFFFFF FFFFFF
FFFFFF 3300CC FFFFFF FFFFFF FFFFFF 33CC00 FFFFFF FFFFFF
FFFFFF 0000FF 0033CC 006699 009966 00CC33 00FF00 FFFFFF
```

これをよく見ると、このデータ表現はきわめて冗長であることに気づきます。何故ならば、上のデータに含まれる画素値の種類は 16 種類だけであり、16 種類のデータならば 4bit で一意に指定できるからです。わざわざフルカラーの 24bit を用いるのは冗長です。

そこで今、4bit データから 24bit データへの関数（写像） f を以下のように定義しましょう。ここに引数値、関数値は全て 16 進数で表しました。

$$
\begin{aligned}
&f(\texttt{0}) = \texttt{FFFFFF}, \quad f(\texttt{1}) = \texttt{FF0000}, \quad f(\texttt{2}) = \texttt{CC0033}, \quad f(\texttt{3}) = \texttt{CC3300},\\
&f(\texttt{4}) = \texttt{990066}, \quad f(\texttt{5}) = \texttt{996600}, \quad f(\texttt{6}) = \texttt{660099}, \quad f(\texttt{7}) = \texttt{669900},\\
&f(\texttt{8}) = \texttt{3300CC}, \quad f(\texttt{9}) = \texttt{33CC00}, \quad f(\texttt{A}) = \texttt{0000FF}, \quad f(\texttt{B}) = \texttt{0033CC},\\
&f(\texttt{C}) = \texttt{006699}, \quad f(\texttt{D}) = \texttt{009966}, \quad f(\texttt{E}) = \texttt{00CC33}, \quad f(\texttt{F}) = \texttt{00FF00}.
\end{aligned}
$$

この写像を用いるならば、画像データとして各画素値は 4bit（1 桁の 16 進数）で以下のように表わすことができます。

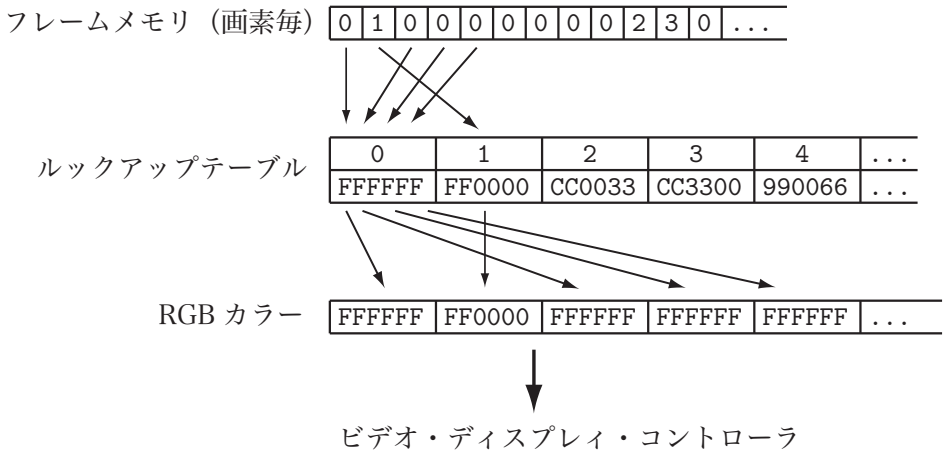


図 3.19: インデックスカラーから RGB カラーへの変換

```
0 1 0 0 0 0 0 0
0 2 3 0 0 0 0 0
0 4 0 5 0 0 0 0
0 6 0 0 7 0 0 0
0 8 0 0 0 9 0 0
0 A B C D E F 0
```

この方法で必要メモリ量は 1/6 に削減できます。元のフルカラーデータを求めるときには、各 4bit データを写像 f で 復元します。

上に述べた方法は単なるアイディアではなく、図 3.19 のような構造で多くのコンピュータにハードウェア的に実装されています。以下はその説明です。

1. まず、フレームメモリ上には f によって圧縮された画素値を格納しておきます。このフレームメモリ内のデータを**インデックスカラー**（indexed color[30]）と呼んでいます。

2. その画素値をフルカラーの映像信号として取り出すために f に相当する配列を別途用意しておきます。これを一般に**ルックアップテーブル**（Lookup Table、検索表、参照表）[31]と呼びます。

3. 画素値を添字としてルックアップテーブルから配列要素値を取り出せば、それが求める映像信号です。

このインデックスカラー方式の欠点は、ルックアップテーブルに存在しない色を表示したい場合には、その色をテーブルに存在する色で近似せねばならないことです。そのため、インデックスカ

[30] 直訳すると「インデックスされたカラー」となるべきですが、読みにくいせいか、専門書の日本語訳では「された」は省略されています。

[31] ルックアップテーブルは CG に限らず、コンピュータによる計算省力化、メモリ使用量削減の手法として様々な処理に広く利用されています。LUT とも略します。

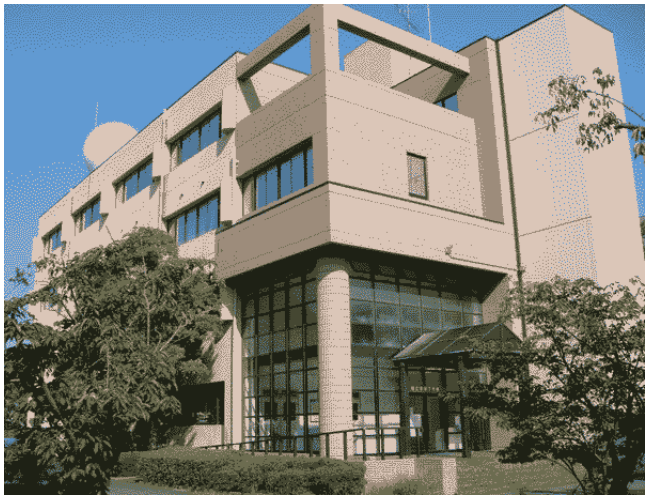


図 3.20: 7 号館のインデックスカラー画像（画像サイズ 600×460）の例

ラーでは正しい画素色が表示できない場合があり、画質が劣化します。メモリ効率と画質は相反するトレードオフの関係にある訳です。かつてメモリが高価であった頃にはインデックスカラーは非常に有効な表現方法でした。しかし、メモリが安価となった現在では、タブレット端末においてさえフルカラーを実装しており、インデックスカラーの利用頻度は減少しています。

ところで、2 値やグレースケールはインデックスカラーの特殊な場合であることに注意してください。2 値の場合のルックアップテーブルは

$$f(0) = 000000, \quad f(1) = \mathtt{FFFFFF}$$

であり、グレースケールの場合は

$$f(x) = xxx \qquad (00 \leq x \leq \mathtt{FF})$$

と定義できるからです。

インデックスカラーの使用例として、図 3.20 に理工学部 7 号館の写真のインデックスカラー画像を載せました。この画像は画素当たり 8bit の情報を保持しています。同じ写真のフルカラー画像（図 3.16）と比較してください。随所に画像の劣化を見ることができます。

3.4.5 色深度

画像内の個々の画素情報に割り当てられたビット数のことを**色深度**（color depth）と呼び、その画像の色の表現力を知る単純な指標として利用されています。この節で学んだ 2 値、グレースケール、フルカラーについて表 3.3 にまとめました。インデックスカラー（3.4.4 節）の色深度は実装により様々であるため、この表には載せませんでした。一例をあげれば、図 3.20 は画素当たり 8bit を使用しているため、色深度は 8 です。

表 3.3: 代表的な色深度

画素名	色深度（bit 数）
2 値	1
グレースケール	8
フルカラー	24 または 32

表 3.4: NEC PC-9801 シリーズのメインメモリ容量の変遷（ニコニコ大百科「PC-9801 のデスクトップ機一覧」の項目を参照）

年. 月	製品名	メモリ（KB）
1982.10	PC-9801	128
1985.07	PC-9801VM21	384
1986.06	PC-9801UV21	384
1986.11	PC-9801VX	640
1988.07	PC-9801RA	1600
1989.04	PC-9801ES	1600
1990.11	PC-9801DA	1600
1992.01	PC-9801FA	1600
1993.01	PC-9801BA/U6	3600
1993.11	PC-9801BA2	3600
1995.01	PC-9801BA3/U2W	7600

なお、表中の 32bit のフルカラーとは、32bit 中の 24bit でフルカラーの情報に使い、残りの 8bit（= 32bit−24bit）を画像の重ね合わせのときの不透明度のパラーメータとして使用する方式です。この 8bit のパラメータは α 値、α（アルファ）チャネルなどと呼ばれており、次章で述べる予定です。

3.4.6 寄り道：ラスタ画像の使用メモリ量

既に述べたように、ラスタ・グラフィックスがベクタ・グラフィックスを席巻した背景にはメモリの急速な低価格化が大きな要因です。これを簡単に検証してみます。

1980 年代から 90 年代の日本の代表的な家庭用 PC として、NEC PC-9801 シリーズが有名です。この PC のメモリ容量の変遷をまとめたものが表 3.4 です。これを見ると、メモリ容量はおおむねムーアの法則[32]に従いながら増加しています。

[32] 集積回路のコストは 1.5 年で約 1/2 になるという経験則です。単純化してメモリについて言えば、1.5 年で容量が 2 倍になる、つまり 3 年で 4 倍、6 年で 16 倍、9 年で 64 倍、10 年で約 100 倍になる計算です。その意味では、表 3.4 のメモリの伸びはムーアの法則よりもやや鈍いようです。

ところで、VGA 規格（表 3.2）の 24bit フルカラー画像[33]に必要なメモリは約 920KB（$= 640 \times 480 \times 3$）です。これを表 3.4 と比較すると、1988 年に PC-9801RA のメモリ容量が 1MB を越えており、この時点で 1 枚の VGA フルカラー画像データをメインメモリに載せるだけの容量に到達したことになります。もちろんメインメモリはフレームメモリ（図 3.8）とは別ものですが、この 1990 年前後から PC でフルカラーのラスタ画像を扱うことのできる時代に入っていきます。同時にこの時代から X-Y プロッタなどの線画の表示装置は急速に廃れていきます。

[33] 厳密に言えば、VGA は画像サイズ（$= 640 \times 480$）だけではなく、各画素データは 4bit（=16 色）のインデックスカラーという規定があるのですが、ここでは画像サイズのみを VGA 規格と見なします。

第IV部

第4章　2次元グラフィックス

CG といえば 3 次元 CG を思い浮かべる人が多いことでしょう。しかし、2 次元的な図形を描画する手法を理解すること無くして、3 次元 CG を深く理解することはできません。この章では 2 次元 CG の様々な基礎技術を順に学んでいきます。

なお、この授業は p5.js をマスターすることを目指すものではありません。p5.js の必要最低限の基本機能のみを紹介し、それを使った CG プログラムの例、関連アルゴリズムを解説することが主目的です。より詳しく学びたい人、興味を持った人は p5.js の機能のまとめページ `https://p5js.org/reference/` などを利用して自ら調べてください。

4.1　様々な線画

ここでは様々な線画、特に多角形の描画とそのバリエーションを紹介し、2 次元 CG に慣れ親しみます。線画ですので、3.1 節のベクタ・グラフィックスをベースとした内容です。

4.1.1　三角形

まず、簡単な例題として三角形を描くプログラムを紹介します。図 4.1 がそのプログラム、図 4.2 がその実行結果です。以下は、このプログラムで使用されている関数 `createCanvas()`、`line()` の解説です。なお、このテキストでは実行結果を表示するウインドウ内の描画領域を**仮想的なキャンバス**（virtual canvas）、あるいは単にキャンバスと呼ぶこととします。

```
function setup() {
  createCanvas(640, 480);

  line(200, 100, 100, 300);
  line(100, 300, 500, 300);
  line(500, 300, 200, 100);

}
```

図 4.1: 三角形の描画プログラム

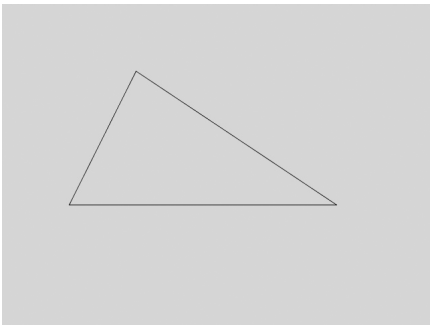


図 4.2: 左のプログラムの実行結果

`setup()` ... 初期化関数です。プログラムの実行において初めに 1 回だけ実行されます。`draw()` が定義されていない場合には、この関数での描画結果がプログラムの実行結果としてキャンバスに保持されます。

`createCanvas(width, height)` [1] ... キャンバスの幅、高さを定数値で指定します。この関数を明示的に呼び出さない場合、`width`=`height`=100 がデフォルト値です。何かを描画した後にこの関数を呼び出した場合、それまでの描画がクリアされますから、注意してください。

`line(x1, y1, x2, y2)` [2] ... 2 次元 X-Y 座標上の点 (`x1`,`y1`) から点 (`x2`,`y2`) へ直線を描きます。3.3.5 節で述べたように、p5.js の 2 次元座標系はキャンバスの左上が原点であって、Y 座標値は下に向かって増えていきます。

4.1.2　正多角形

次に正 6 角形に挑戦しましょう。たとえば座標点 $(300, 250)$ に中心を持つ半径 200 の円を想定し、その円に内接する正 6 角形を描画します。そのような正 6 角形の 6 個の頂点座標は以下の通りに表わすことができます。ここに $i = 0, 1, ..., 5$ です。

$$(200\cos(60i^\circ),\ 200\sin(60i^\circ)) + (300, 250) \tag{4.1}$$

上の数式では角度を度数で表わしていますが、実際の数値計算ではラジアンで計算せねばなりません。一般に度数 d をラジアン r に変更するには式 $r = d \cdot \pi/180$ で計算すればよいのですが、p5.js では以下の変換関数 `radians()` が用意されています。

`radians(degrees)` ... 度数で表された角度 `degrees` に対応するラジアン値を返します。逆の方向の計算（ラジアンを度数へ変換）は関数 `degrees(radians)` で可能です。

以上の考察をプログラム化したものが図 4.3、その実行結果が図 4.4 です。

注意（その 1）　図 4.3 のプログラム中のキーワード `const` は、そのキーワードの後に続く変数の値を変更できないことを示す JavaScript のキーワードです。`const` が付いた変数は定数値と同じ扱いになります。一回だけ変数へ値を代入できることを **単一代入**（single assignment）と呼びます。ひとつの変数に何度も値を代入する処理は、一般にプログラムの読みやすさを低下させる[3]ため、できるだけ避けるべきです。メモリが潤沢に利用できる現在のコンピュータでは単一代入が推奨されます。

---

[1] 同名の関数 `createCanvas(width, height, renderer)` については後章で解説します。
[2] 同名の関数 `line(x1, y1, z1, x2, y2, z2)` については後章で解説します。
[3] 単一代入でない場合、ある時点でその変数に代入されている値を知るためにはプログラムを丁寧に読まないと把握できないため、プログラム理解の妨げとなります。

```
function setup() {
  createCanvas(600, 500);

  const xoffset = 300.0, yoffset = 250.0;
  const r = 200.0;

  for (let i = 0; i < 6; i++) {
    const t1 = radians(60.0 * float(i));
    const x1 = r * cos(t1) + xoffset;
    const y1 = r * sin(t1) + yoffset;

    const t2 = radians(60.0 * float(i + 1));
    const x2 = r * cos(t2) + xoffset;
    const y2 = r * sin(t2) + yoffset;

    line(x1, y1, x2, y2);
  }
}
```

図 4.3: 6 角形の描画プログラム

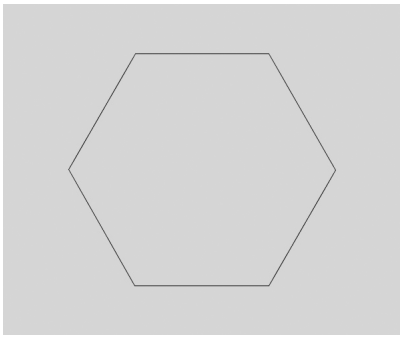


図 4.4: 図 4.3 のプログラムの実行結果

これに対して、図 4.3 のプログラム中の `for` 文のヘッダー部分に現れるキーワード `let` はその直後に宣言された変数 `i` の値を変更できることを意味します。もちろんこの場合、変数 `i` を `const` で宣言することはできません。

JavaScript にはこれ以外にキーワード `var` も使用可能です。詳細は述べませんが、プログラムの可読性からは `var` は推奨されません[4]。

注意（その 2）　p5.js のベース言語である JavaScript では、変数は特定のデータ型を持ちません。このような言語を**動的型付け**（dynamic typing）の言語と呼びます。変数に代入される値のデータ型は、その時点で変数に代入されている値の型です。図 4.3 の変数 `xoffset` は代入される値が数値 `300.0` であるため、結果として数値を保持しています。変数のデータに関するこの性質は、C/C++/Java とは異なり、Python と同じです。

さて、正 6 角形の議論は簡単に正 n 角形へ拡張できます。つまり式 (4.1) を

$$\left(200\cos\left(\left(\frac{360}{n}\right)i^\circ\right),\ 200\sin\left(\left(\frac{360}{n}\right)i^\circ\right)\right)+(300,250)$$

に変えればいいだけです。そのプログラムが図 4.5、実行結果は図 4.6 の通りです。

注意　JavaScript の数値は暗黙では全て浮動小数点型です。ですから、図 4.5 のプログラム中の角度の増分値の計算 `360/n` は計算結果が浮動小数点型になることに注意してください。しかし

[4] プログラムの実行速度の観点からは、`var` を使った方が速いという報告もあるようです。

```
function setup() {
  createCanvas(600, 500);

  const xoffset = 300.0, yoffset = 250.0;
  const r = 200.0;

  const n = 11;              // 角数の設定
  const incAngle = 360 / n;
                     // 角度の増分値の計算

  for (let i = 0; i < n; i++) {
    const t1 = radians(incAngle *
                       float(i));
    const x1 = r * cos(t1) + xoffset;
    const y1 = r * sin(t1) + yoffset;

    const t2 = radians(incAngle *
                       float(i + 1));
    const x2 = r * cos(t2) + xoffset;
    const y2 = r * sin(t2) + yoffset;

    line(x1, y1, x2, y2);
  }
}
```

図 4.5: 正 n 角形の描画プログラム：$n = 11$ の場合

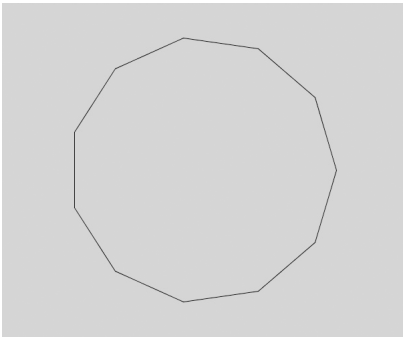


図 4.6: 図 4.5 のプログラムの実行結果

C/C++/Java では、`360/n` の結果は、 `n` が整数ならば除算結果も（小数点以下を切り捨てた）整数です。浮動小数点型と整数では結果が大きく異なりますので、注意してください。

考察　図 4.5 のプログラムにおいて n を十分大きくすれば近似的に円を描くことができるはずです。人の目で見て円と区別できない n を自分の目で調べなさい。

4.1.3　星型正多角形

さらにプログラムを改造します。多角形の n 個の頂点について、隣接する頂点間を描画するのではなく、頂点を飛ばして描画すると星型の図形を描画できます。五芒星（図 4.7 参照）が著名な例です。

一般に正 n 角形の頂点を、飛び数 m で結んで（頂点を $m-1$ 個飛びで結んで）描かれる図形を**正 n/m 角形**、総じて**星形正多角形**（regular star polygon）と呼びます。ただし、ここでは n と m が互いに素とします[5]。五芒星は正 5/2 角形に相当します。図 4.8 は正 23/7 角形です。これを描

[5] n と m が互いに素でない場合には議論が少し複雑になり、複合型という形状を考えねばなりません。たとえば六芒星がこの場合に当たります。

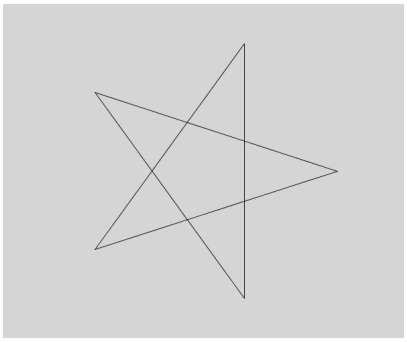


図 4.7: 五芒星（＝正 5/2 角形）

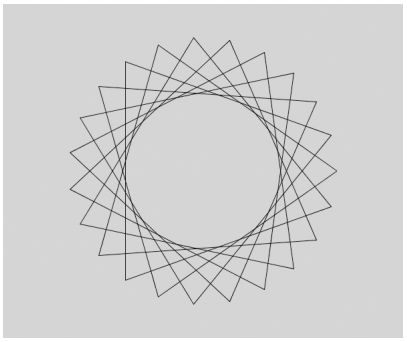


図 4.8: 図 4.9 のプログラムで作成した正 23/7 角形

画するプログラムは図 4.9 の通りです。

4.1.4　らせんと渦巻き

ここまで作ってきた多角形のプログラムでは内接円の中心は $(300, 250)$ に固定していました。この中心を角度と共に平行移動することで**らせん**（螺旋、helix）を作ることができます。たとえば図 4.10 のようなものです。

ここまで作ってきた多角形のプログラムでは半径 $r = 200$ の円の内接点を用いました。この半径 r を角度と共に単調増加させることで**渦巻き**（spiral）を描画できます。たとえば図 4.11 のようなものです。

考察　図 4.10 や図 4.11 のような画像を作成するプログラムを作成しなさい。

```
function setup() {
  createCanvas(600, 500);

  const xoffset = 300.0, yoffset = 250.0;
  const r = 200.0;
  const n = 23;
  const m = 7;                          // m の宣言を追加
  const incAngle = 360.0 / float(n);

  for (let i = 0; i < m * n; i += m)  // i を m ずつ増やす
  {
    const t1 = radians(incAngle * float(i));
    const x1 = r * cos(t1) + xoffset;
    const y1 = r * sin(t1) + yoffset;

    const t2 = radians(incAngle * float(i + m)); // 終点を m 個先へ変える
    const x2 = r * cos(t2) + xoffset;
    const y2 = r * sin(t2) + yoffset;

    line(x1, y1, x2, y2);
  }
}
```

図 4.9: 正 n/m 角形の描画プログラム：$n = 23$、$m = 7$ の場合

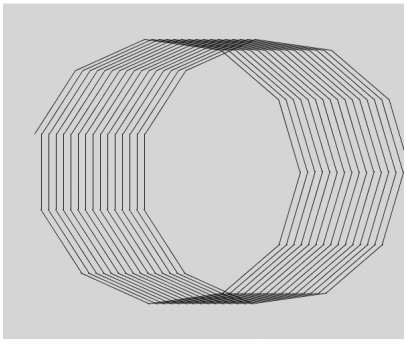


図 4.10: らせんの例

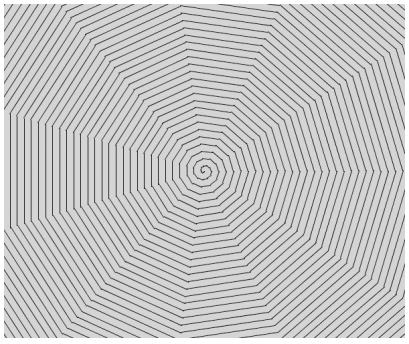


図 4.11: 渦巻きの例

```
function setup() {
  createCanvas(500, 500);

  background(128);

  for (let i = 100; i <= 400; i += 100) {
    stroke(i / 2);
    strokeWeight(i / 10);
    line(100, i, 400, i);
  }
}
```

図 4.12: 背景色、線の色をグレースケールで指定し、さらに線幅を変えるプログラム

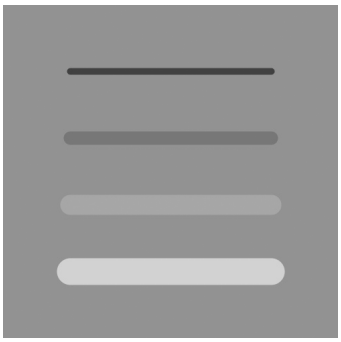


図 4.13: 左のプログラムの実行結果

```
function setup() {
  createCanvas(500, 500);

  background(0, 255, 0);

  for (let i = 100; i <= 400; i += 100) {
    stroke(i / 2, 0, 0);
    strokeWeight(i / 10);
    line(100, i, 400, i);
  }
}
```

図 4.14: 背景色、線の色を RGB カラーで指定し、さらに線幅を変えるプログラム

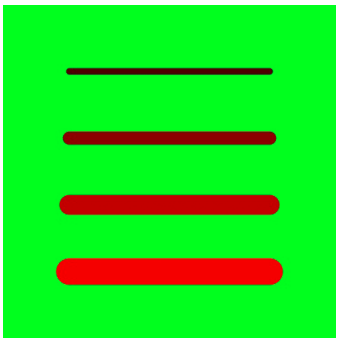


図 4.15: 左のプログラムの実行結果

4.1.5　線の色、幅の変更

ここまでの描画では線の色、幅はデフォルト値（色は黒、幅は 1 画素）を用いてきましたが、これを変更することができます。

図 4.12 は、背景色、線の色、線の幅を様々な明るさの灰色に変更するプログラムの例です。図 4.13 がその実行結果です。図 4.14 は、背景色、線の色をフルカラー RGB 値で指定するプログラムです。その実行結果が図 4.15 です。

これらのプログラムでは以下の関数を用いてます。

`background(gray)` ... キャンバスの背景色をグレイスケール値で指定します。0 が黒色、255 が白色、1〜254 は暗い灰色から明るい灰色です。プログラム中でこの関数を呼び出さない場合

```
function setup() {
  createCanvas(640, 480);

  stroke(0, 255, 0);
  strokeWeight(10);

  fill(255, 0, 0);

  triangle(200, 100, 100, 300, 500, 300);
}
```

図 4.16: 内部を赤色で塗り、境界を緑色で塗る三角形の描画プログラム

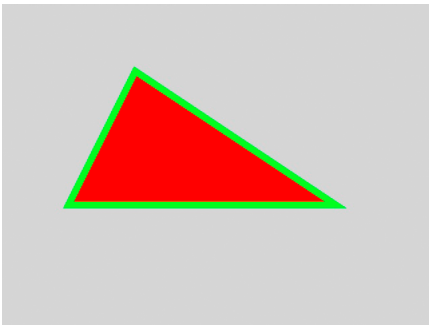


図 4.17: 左のプログラムの実行結果

は明るい灰色がデフォルト値[6]です。

`background(red, green, blue)` ... キャンバスの背景色を RGB 値 (`red`, `green`, `blue`) で指定します。

`stroke(gray)` ... 線の色をグレイスケールで指定します。プログラム中でこの関数を呼び出さない場合は黒色がデフォルト値です。

`stroke(red, green, blue)` ... 線の色を RGB 値 (`red`, `green`, `blue`) で指定します。

`strokeWeight(weight)` ... 線の幅を画素単位で指定します。デフォルト値は 1 です。

注意　`background(gray)` と `background(red, green, blue)` のように同じ関数名で異なる引数の関数は別の関数として認識されます。このように同じ名前で異なる引数の個数を持つ関数は互いに**オーバーロード**（overload、多重定義）されていると言います。

考察　p5.js の関数 `strokeCap()` の処理内容を調べなさい。

4.2　様々な面画

複数の線分で囲まれた領域を特定の色で**塗りつぶす**ことで面画を作ることができます。

4.2.1　三角形

まず、簡単な例題として三角形の内側を赤色で塗るプログラムを図 4.16 に示します。実行結果は図 4.17 です。

[6] 明るい灰色の具体的な数値は不明です。

```
function setup() {
  createCanvas(640, 480);

  noStroke(); // この行を変更
  strokeWeight(10); // 削除してもよい

  fill(255, 0, 0);

  triangle(200, 100, 100, 300, 500, 300);
}
```

図 4.18: 境界線を描かない三角形の描画プログラム

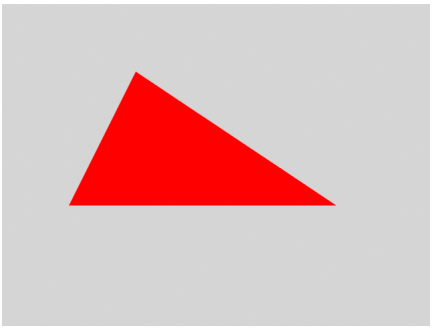


図 4.19: 左のプログラムの実行結果

このプログラムでは、図 4.2 と同じ形状の三角形を、境界線を緑色で太さが 10 画素の線分で描き、内部を赤色で塗っています。これには以下の関数 `fill()`、`triangle()` を用います。

`fill(gray)` ... 描画形状の内部の色をグレイスケールで指定します。

`fill(red, green, blue)` ... 描画形状の内部の色を RGB 値 (`red`, `green`, `blue`) で指定します。

`triangle(x1, y1, x2, y2, x3, y3)` ... 2 次元 X-Y 座標上の 3 点 (`x1`, `y1`)、(`x2`, `y2`) 、(`x3`, `y3`) でひとつの三角形を指定します。

プログラム中で `fill()` が呼ばれないときの内部色のデフォルト値は白です。

図 4.17 の緑の境界線を描画したくないときには図 4.18 のようにプログラムを改造します。実行結果は図 4.19 です。図 4.16 のプログラムの `stroke(0, 255, 0)` を線分の描画を行わないことを設定する関数 `noStroke()` へ変更するだけです。

逆に図 4.17 の内部を塗りたくないときには図 4.20 のようにプログラムを改造します。実行結果は図 4.21 です。図 4.16 のプログラムの `fill(255, 0, 0)` を図形の内部を塗らないことを設定する関数 `noFill()` へ変更するだけです。

関数 `noStroke()` および `noFill()` の両方を同時に呼び出した場合にはキャンバスには何も描画されません。

考察　p5.js の関数 `strokeJoin()` の処理内容を調べなさい。

```
function setup() {
  createCanvas(640, 480);

  stroke(0, 255, 0);
  strokeWeight(10);

  noFill(); // この行を変更

  triangle(200, 100, 100, 300, 500, 300);
}
```

図 4.20: 内部を塗らない三角形の描画プログラム

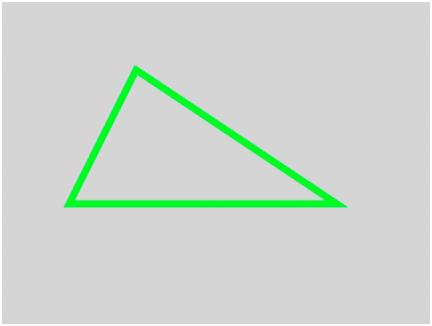


図 4.21: 左のプログラムの実行結果

```
function setup() {
  createCanvas(640, 480);

  noStroke();

  background(0, 0, 255);

  fill(255, 0, 0);
  triangle(200, 100, 100, 300, 500, 300);

  fill(0, 255, 0);
  triangle(300, 100, 200, 300, 600, 300);
}
```

図 4.22: 二つの三角形の描画プログラム（その 1）

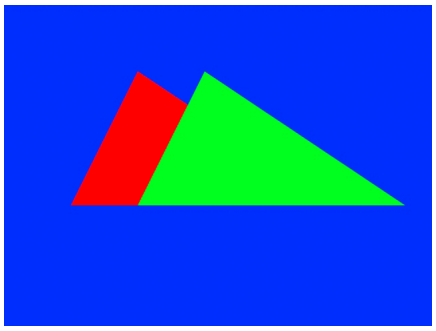


図 4.23: 左のプログラムの実行結果

4.2.2　図形の重ね合わせとアルファ値

複数の図形を重ねて描画する場合、デフォルトの設定では先に描画された図形を後に描画される図形が単純に上書きします。図 4.22 は二つの三角形を重ねて描画するプログラムです。図 4.23 はその実行結果です。

しかし関数 `blendMode()` とアルファ値を用いれば、様々な色の重ね合わせが可能です。

3.4.4 節において色深度が 32bit であるときに、そのうちの 8bit を不透明度を表すアルファ値に用いると述べました。ここで述べるのはそのことです。

図 4.24 のプログラムと図 4.25 の実行結果が色を重ね合わせる簡単な例です。このプログラムには関数呼び出し `blendMode(ADD)` を追加しています。この関数呼び出しは、色を重ねるときに加算することを指示するものです。その結果、図 4.23 では単純な上書きだったものが、図 4.25 では色の混ぜ合わせ[7]：赤+緑=黄、青+緑=水色になっています（図 3.14 参照）。

さらに、図 4.26 のプログラムでは関数 `fill()` に引数を追加しています。その結果、図 4.27 の実行結果のように、重ね合わせた色が背景色に寄った色になります。

この重ね合わせの仕組みを解説します。実はこの重ね合わせの仕組みは、p5.js に限らず、多くのグラフィックスライブラリの共通の機能です。

まず、p5.js のキャンバスは 32bit フルカラーの色深度を有しており、24bit の RGB 情報に加え、8bit のアルファ値を持っています（図 4.28 参照）。アルファ値はその色をキャンバスに塗る際にその色の透明の度合いを指定できます。図 4.26 のプログラムで使用した関数 `fill()` はアルファ値も指定しており、その内容は以下の通りです。

[7] この色の重ね合わせは加色混合に従います。

```
function setup() {
  createCanvas(640, 480);

  noStroke();

  background(0, 0, 255);

  fill(255, 0, 0);
  triangle(200, 100, 100, 300, 500, 300);

  blendMode(ADD);  // 関数呼び出しを追加

  fill(0, 255, 0);
  triangle(300, 100, 200, 300, 600, 300);
}
```

図 4.24: 二つの三角形の描画プログラム（その 2）

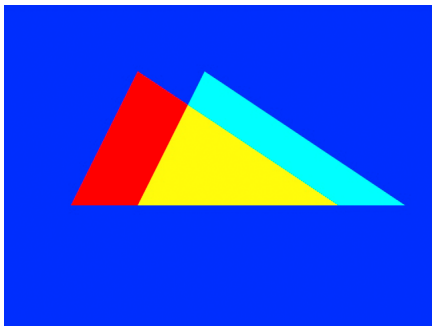


図 4.25: 左のプログラムの実行結果

4 `fill(red, green, blue, alpha)` ... 描画形状の内部の色を RGB 値 (`red`, `green`, `blue`) で指定
1 し、アルファ値を `alpha` で指定します。

今、図 4.29 のように、キャンバス上のある画素位置に RGB カラーを $A = (A_r, A_g, A_b)$ を新たに塗ることを考えます。このとき、キャンバス上のその画素位置には既に RGB カラーを $B = (B_r, B_g, B_b)$ が塗られていると仮定します。また、A に付随するアルファ値を α_A とします。ここに各輝度値、アルファ値 は $[0, 255]$ の範囲です。図 4.24 のプログラムのように `blendMode(ADD)` を設定したならば、実際にキャンバスに塗られる色 $C = (C_r, C_g, C_b)$ は以下の式で計算されます。

$$C_i = \left(\frac{\alpha_A}{255}\right) \cdot A_i + B_i \qquad i = r, g, b$$

ただし、右辺が 255 を超えた場合には左辺には 255 が代入されます[8]。アルファ値を指定しない図 4.24 のプログラムの場合、暗黙には $\alpha_A = 255$ とみなされますから、

$$C_i = A_i + B_i$$

と計算されます。アルファ値 128 を指定した図 4.26 のプログラムでは

$$C_i = 0.5 \cdot A_i + B_i$$

2 と計算されることになります。

3 計算式の形状は関数 `blendMode()` で指定しますが、その詳細は以下の通りです。

[8] よって、正確には

$$C_i = \min\left(\left(\frac{\alpha_A}{255}\right) \cdot A_i + B_i,\ 255\right) \qquad i = r, g, b$$

と書くべきですが、煩雑なため、$\min(\ \ , 255)$ を省略しました。

```
function setup() {
  createCanvas(640,480);
  noStroke();

  background(0,0,255);

  fill(255, 0, 0);
  triangle(200,100, 100,300, 500, 300);

  blendMode(ADD);

  fill(0, 255, 0, 128); // 引数を追加
  triangle(300,100, 200,300, 600, 300);
}
```

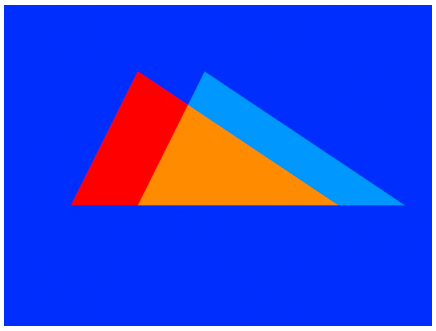


図 4.27: 左のプログラムの実行結果

図 4.26: 二つの三角形の描画プログラム（その 3）

8bit	8bit	8bit	8bit
赤の輝度値	緑の輝度値	青の輝度値	α 値

図 4.28: 32bit フルカラー画素情報の内訳

`blendMode(mode)` ... `mode` に以下のいずれかのマクロ値を与え、重ね合わせの式を指定します。

`BLEND` ... A と B の色を α_A の値で線形補間します。すなわち

$$C_i = \left(\frac{\alpha_A}{255}\right) \cdot A_i + \left(1 - \frac{\alpha_A}{255}\right) \cdot B_i$$

これは `blendMode()` を呼び出さない場合の重ね合わせのデフォルト式でもあります[9]。

また、関数 `fill()` の第 4 引数を与えない場合のアルファ値のデフォルトは 255 です。

よって、重ね合わせを全く指定しない場合には

$$C_i = A_i$$

が実行されます。すなわち単純な上書きです。

`ADD` ... 既に解説した通りです。A の色に α_A 分の重みを付けて B の色と加算します。ただし、各輝度値の最大は 255 です。

`SUBTRACT` ... B の色から A の色に α_A 分の重みを付けた色を引きます。

$$C_i = B_i - \left(\frac{\alpha_A}{255}\right) \cdot A_i$$

[9] p5.js のホームページの `blendMode()` の当該説明箇所には

$$C_i = \left(\frac{\alpha_A}{255}\right) \cdot A_i + B_i$$

に相当する式が載っていますが、これは間違いと思われます。

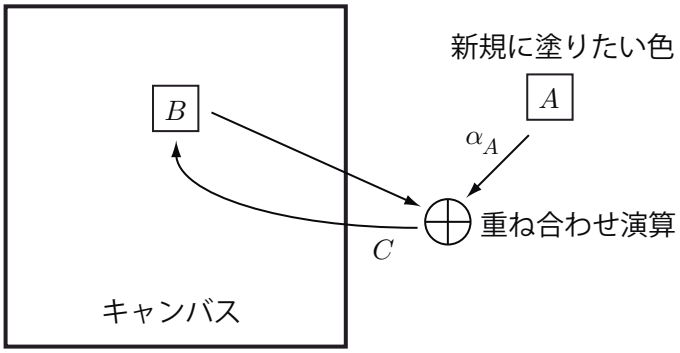


図 4.29: 32bit フルカラー画素情報の内訳

ただし、右辺が負値の場合には左辺は 0 です。

`DARKEST` ... A の色に α_A 分の重みを付けた色と B の色の小さい方の輝度値を用います。すなわち

$$C_i = \min\left(\left(\frac{\alpha_A}{255}\right) \cdot A_i,\ B_i\right)$$

その他 ... その他に、`LIGHTEST`、`DIFFERENCE`、`EXCLUSION` 、`MULTIPLY`、`SCREEN`、`REPLACE` が使用可能です。内容は自身で調べてみてください。

どのような重ね合わせを用いればどのような効果が得られるのかは、実際にプログラムを実行してみないと分かりにくいものです。図 4.26 のプログラムを参考に、自分自身で種々、試してみてください。

第Ⅴ部

```
function setup() {
  createCanvas(100, 150);

  arc(50, 75, 80, 120, 0, HALF_PI);
}
```

図 4.30: 縦長の 1/4 の円弧の描画プログラム

図 4.31: 左のプログラムの実行結果

```
function setup() {
  createCanvas(200, 100);

  ellipse(100, 50, 150, 60);
}
```

図 4.32: 横長の楕円の描画プログラム

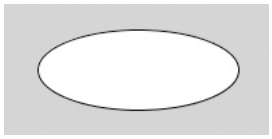


図 4.33: 左のプログラムの実行結果

4.2.3 様々な形状

円弧

円弧型の図形を描画するには以下の関数を用います。

`arc(x, y, w, h, startAngle, endAngle)` ... (x, y) に円の中心座標を指定し、`w` に x 方向（水平方向）の幅、`h` に y 方向（垂直方向）の幅を指定します。引数 `startAngle` には円弧の開始角度、`endAngle` に円弧の終了角度を指定します。

使用例が図 4.30 のプログラム、実行結果は図 4.31 です。この例題では、やや縦長の円弧で、角度が 0 度から 90 度（ラジアンで言えば $\pi/2$=`HALF_PI`）までの全体の 1/4 の大きさの円弧を描画します。

考察　上記関数にさらに引数 `mode` を加えた関数 `arc(x, y, w, h, startAngle, endAngle, mode)` について調べなさい。

楕円

楕円を描画するには以下の関数を用います。

`ellipse(x, y, w, h)` ... (x, y) に円の中心座標を指定し、`w` に x 方向（水平方向）の幅、`h` に y 方向（垂直方向）の幅を指定します。幅は楕円の長径、短径に相当し、長半径（長径の 1/2）、短半径（短径の 1/2）ではないことに注意してください。

```
function setup() {
  createCanvas(200, 100);

  rect(10, 10, 150, 60);
}
```

図 4.34: 横長の長方形の描画プログラム

図 4.35: 左のプログラムの実行結果

使用例が図 4.32 のプログラム、実行結果は図 4.33 です。

長方形

長方形を描画するには以下の関数を用います。

`rect(x, y, w, h)` ... (x, y) に長方形の左上の点の座標（中心の座標ではありません）を指定し、`w` に x 方向（水平方向）の幅、`h` に y 方向（垂直方向）の幅を指定します。

使用例が図 4.34 のプログラム、実行結果は図 4.35 です。

考察　関数 `quad()` を用いれば、任意の四辺形を描画できます。詳細を調べてください。

4.3 CG アニメーション

描画プログラムにパラメータを導入し、そのパラメータを時間と共に連続的に更新させながら描画を繰り返せば、GC アニメーションが可能です。p5.js では CG アニメーションを非常に簡単に実装できます。

この節ではまず p5.js における CG アニメーションの基本技術を解説します。次に、動画のチラつきを抑えるための技法であるダブルバッファについて概説します。最後に、マウス、キーボードを用いたイベント処理について紹介します。

4.3.1 様々な CG アニメーション

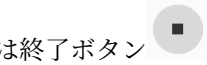
アニメーションプログラムの基本形

実は p5.js のプログラムには二つの形態があります。

ひとつは、ここまで紹介してきた CG アニメーションを行わない p5.js のプログラムです。この形のプログラムを p5.js では **static sketch**（静的なスケッチ）と呼んでいます。

もうひとつは、CG アニメーションを行うための p5.js のプログラムで、以下の二つの関数を定義せねばなりません。この形のプログラムを **active sketch**（動的なスケッチ）と呼んでいます。static sketch は、`setup()` だけからなる active sketch と見なすこともできます。

`setup()` ... 4.1.1 節でも紹介しましたが、この関数に初期化処理を記述します。プログラムの開始直後にシステムから 1 回だけ呼び出されます。

`draw()` ... ここに毎回の描画処理を記述します。`setup()` の終了後、デフォルトの設定では `draw()` は 1 秒間に 60 回ずつ繰り返し呼び出されますが、個々の PC に依存します。プログラムを終了するには終了ボタン  を押してください。

左から右へ黒い領域が広がる

図 4.36 は非常に簡単な CG アニメーションのプログラム例です。図 4.37 はその実行途中の様子です。このプログラムでは、以下の処理を行います。

1. 関数 `setup()` では、キャンバスのサイズを 300×200 に設定します。
2. グローバル変数 `x` はプログラム実行直後に 0 に初期化されます。
3. 関数 `draw()` では、$(\mathtt{x}, 0)$ から $(\mathtt{x}, 100)$ まで縦に 100 画素の長さの線分を描画し、`x` の値を 1 増やします。

```
function setup() {
  createCanvas(300, 200);

  background(200);
}

let x = 0;

function draw() {
  line(x, 0, x, 100);
  x++;
}
```

図 4.36: 黒い領域が右側へ広がっていく CG アニメーションのプログラム

図 4.37: 左のプログラムの実行途中の様子

結果、キャンバス内に $(0,0)$ から $(0,100)$ の線分、$(1,0)$ から $(1,100)$ の線分、$(2,0)$ から $(2,100)$ の線分... が順に引かれるため、図 4.37 のように、左から右へ黒い領域がだんだんと広がっていきます。x の値が 300 を超えても描画処理は続きますが、もはや線分はキャンバス外のため、キャンバス上の変化は無くなります。

左から右へ線分が移動する

図 4.36 のプログラムでは線分の描画に伴って黒い領域が広がっていきました。というのも、キャンバスに過去に描画した内容が消されないまま残っているからです。

もし過去の（直前の）描画をクリアし、常に新しい描画像のみを表示したいならば、`draw()` の冒頭で関数 `background()` を呼び出します。

図 4.38 は図 4.36 に関数 `background()` を追加したプログラムです。実行途中の様子が図 4.39 です。実行では縦の線分が左から右へ移動していく様子が見えるはずです。

さて、上に述べた二つの事項：(1) `setup()` と `draw()` を記述すること、(2) 必要に応じて `background()` を呼び出すことが p5.js における CG アニメーションの基本です。

星が回転する

CG アニメーションの少し複雑な例として、星が回転するプログラムを図 4.40 に示します。これは図 4.9 を改造したものです。星の回転角を変数 `alpha` に保持し、`draw()` が呼ばれる度に 1 度ずつ増やしていきます。よって 360 回呼ばれると星は 1 回転します。p5.js のデフォルトでは、関数 `draw()` は 1 秒間に 60 回呼ばれるように設定されています。つまり 1 秒間に 60 枚の画像を更新

```
function setup() {
  createCanvas(300, 200);

  background(200);
          // この 1 行は削除してもよい
}

let x = 0;

function draw() {
  background(200);  // この 1 行を追加
  line(x,0, x,100);
  x++;
}
```

図 4.38: 線分が右へ移動していく CG アニメーションのプログラム

図 4.39: 左のプログラムの実行途中の様子

しているのですが、このような画像の更新頻度 — これをフレームレート（frame rate）とも呼びます — には一般に **FPS**（Frames Per Second、フレーム/秒、fps とも書く）という単位を用います。上の場合には 60FPS の描画速度/フレームレートです。

星の回転速度を変える

p5.js では描画速度の設定を以下の関数で変更できます。

`frameRate(fps)` ... FPS 値を引数 `fps` で指定します。複雑な描画を行う場合などには、仮に大きな FPS 値を指定してもその描画速度を達成できないことがあります。デフォルト値は 60 です。この関数は `setup()` 内から呼ぶことが想定されており、`draw()` から呼んだ場合の効果（実行途中で描画速度を変えること）は保証されません。

図 4.41 は、描画速度を 10FPS で実行するプログラムです。星がゆっくりと回転することを確認してください。

もちろん星の回転速度は変数 `alpha` の増分値（図 4.40 のプログラムでは 1）を変えることでも可能です。星の回転速度を変えたいときに、フレームレートを変えるのか、描画パラメータの増分値を変えるのかはケースバイケースで判断すべきです。一般にはフレームレートが高いとコンピュータの計算負荷が高まりますが、アニメーションは滑らかです。逆の場合には計算負荷は減りますが、アニメーションがぎこちない動きになってしまいます。どの程度のフレームレートでアニメーションが滑らかに見えるかには個人差があります。

```
function setup() {
  createCanvas(600, 500);
}

let alpha = 0;                                                    // 回転角を表す変数

function draw() {
  background(200);

  const xoffset = 300.0, yoffset = 250.0;
  const r = 200.0;
  const n = 23;
  const m = 7; // m の宣言を追加
  const incAngle = 360.0 / float(n);

  for (let i = 0; i < m * n; i += m)
  {
    const t1 = radians(incAngle * float(i)+alpha);       // 回転角を加算
    const x1 = r * cos(t1) + xoffset;
    const y1 = r * sin(t1) + yoffset;

    const t2 = radians(incAngle * float(i + m)+alpha);   // 回転角を加算
    const x2 = r * cos(t2) + xoffset;
    const y2 = r * sin(t2) + yoffset;

    line(x1, y1, x2, y2);
  }
  alpha += 1; // 回転角の更新
}
```

図 4.40: 星が回転する CG アニメーションのプログラム

```
function setup() {
  createCanvas(600, 500);
  frameRate(10);                                          // この 1 行を追加
}
以下、同様
```

図 4.41: 星がゆっくり回転する CG アニメーションのプログラム

4.3.2　表示のタイミングとダブルバッファ（授業対象外）

CG アニメーションを、active sketch を用いるのではなく、static sketch の中にループを作って無理やりに実現できないものでしょうか。答えは否です。というのも、p5.js のプログラムの実行とキャンバスの表示のタイミングは以下のように定められているからです。

static sketch の場合、まず、プログラムの開始直後に `createCanvas()` で指定した大きさのウインドウが開きます。しかし実行途中（描画途中）のキャンバスの様子は表示せず、全ての実行（描画処理）が終わった後にキャンバスをウインドウに表示します。よってプログラマは、キャンバス上に様々な図形が描かれていく過程を見ることはできません[10]。static sketch は 1 枚の CG 画像を作成、表示するための仕組みであって、描画途中の画像は見せないのです。

active sketch の場合は、まず、プログラムの開始直後に `createCanvas()` で指定したウインドウが開きます（これは static sketch と同じです）。そして関数 `setup()` の実行が終わった後にその時点でのキャンバスの内容を表示します。その後は関数 `draw()` の実行が終わった時点毎にキャンバスの内容をウインドウに表示します。

つまり、p5.js では如何なる場合も描画途中のキャンバスの様子は表示しないのです。

さて、上の表示方法を実現するにはキャンバスが 2 枚必要です。何故ならば、`draw()` が呼び出され、キャンバスに CG 画像が描画されているとき、ひとつ前の `draw()` によって描画されたキャンバスの内容はウインドウに表示されているはずです。作成中のキャンバスと表示中のキャンバスをひとつのキャンバス — 言い方を換えると、ひとつのフレームメモリ — で賄（まかな）うことはできませんから、結果、2 枚のキャンバスが必要になるのです。

この 2 枚をキャンバス A、キャンバス B と呼びましょう。active sketch ではこの 2 枚を図 4.42 のように使い分けます。

(1) 今、キャンバス A の内容がウインドウ上に表示されていると仮定します。図 4.42 では、表示されているキャンバス — これを **オンスクリーンキャンバス**（onscreen canvas）と呼びます — を実線が囲みました。これに対して、表示されていないキャンバス — これを **オフスクリーンキャンバス**（offscreen canvas）と呼びます — を点線が囲みました。今、`draw()` が呼ばれ、キャンバス B への描画が始まったと仮定します。

(2) 既に述べたように描画途中のキャンバス B は表示されません。キャンバス A の内容が表示されています。

(3) キャンバス B への描画が終わったならば、すなわち `draw()` の実行が終わったならば、キャンバス B をオンスクリーンに、キャンバス A をオフスクリーンに入れ替えます。たいていの

[10] これはあくまでも p5.js に関する性質であって、他のグラフィックスライブラリの中にはループを使ってアニメーションを実装できるものも多数あります。

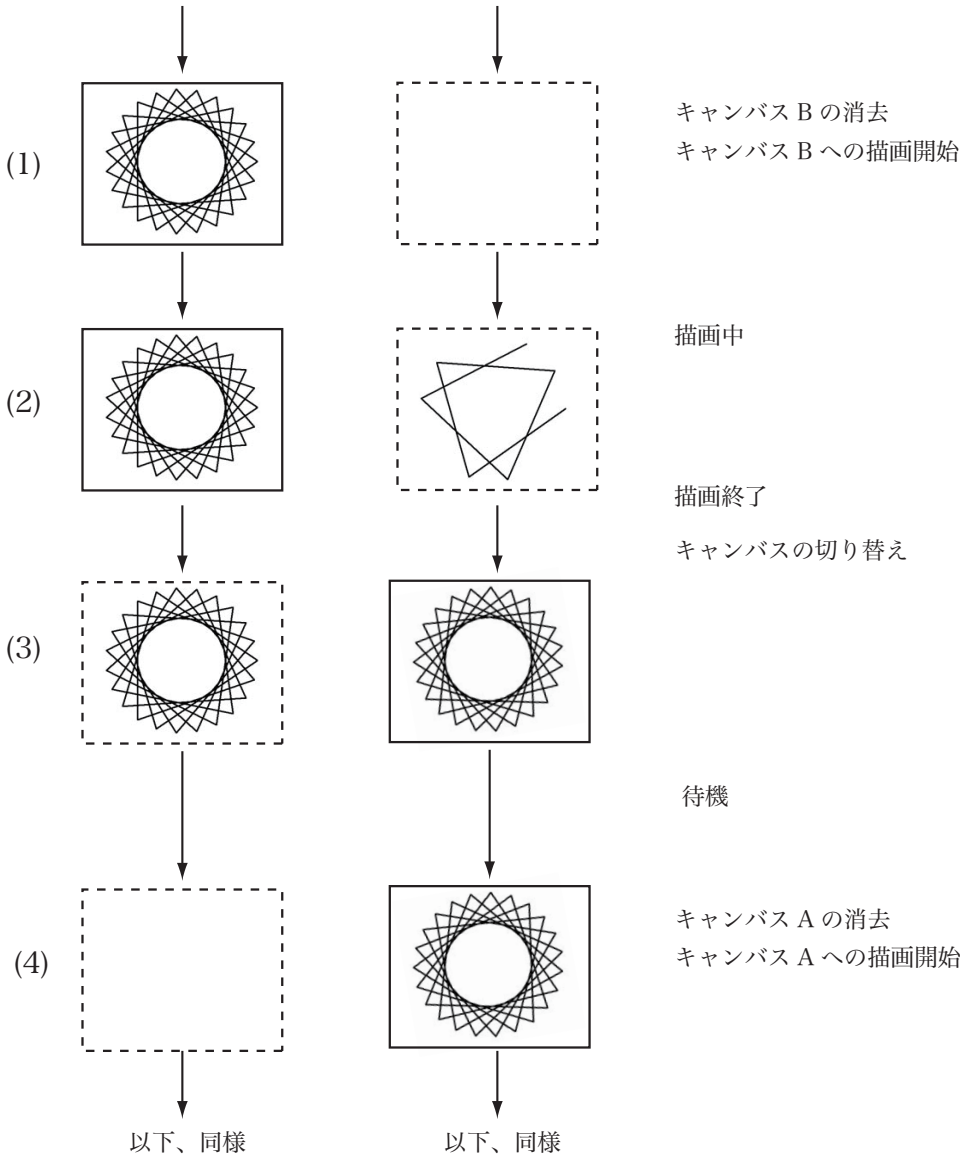


図 4.42: 2 枚のキャンバスによる実行の様子

場合、キャンバスの入れ替えは、キャンバスの内容を保持するフレームメモリの先頭アドレスを入れ替えるだけで済みますから一瞬の操作です。入れ替えた後は、次の `draw()` を呼び出すタイミングまでそのままキャンバス A/B ともに待機します。次の `draw()` は、あらかじめ設定されたフレームレートに従って呼び出されます。

(4) 上の (1)〜(3) を、キャンバスを入れ替えながら繰り返していきます。

この方法の長所は、描画途中の画像を表示せず、完成した画像を瞬時に切り替えながら表示していくため、アニメーションにチラつきが発生しない点です[11]。2 枚のキャンバスを用いるこの方法をしばしば**ダブルバッファ法**（double buffering method）と呼びます。リアルタイム CG アニメーションをサポートするほとんど全てのグラフィックスライブラリはダブルバッファ法を採用しています。

実はダブルバッファ法は、並列計算手法のひとつを応用したものです。N 個の処理が直列に並んだパイプライン処理を N 個のバッファメモリを介して効率的に実行する手法のことを**マルチバッファ法**（multiple buffering method）などと呼びます。CG のダブルバッファ法は、それをアニメーションの (1) 描画する、(2) 表示する、の二つの処理に応用したものになっています。

[11] キャンバスを一枚しか用いない場合、描画途中の画像が表示されるため、アニメーション画像のチラつきが避けられません。

```
function setup(){
  createCanvas(500,500);
}

function draw(){
  print(mouseX,mouseY);
}
```

図 4.43: マウスの位置をコンソールに表示するプログラム

```
function setup(){
  createCanvas(500,500);
}

function draw(){
  rect(mouseX,mouseY,10,10);
}
```

図 4.44: マウスの位置に 10×10 の正方形を描画するプログラム

4.3.3 イベント処理、その他

p5.js ではマウスやキーボードからの入力を受け付けることができます。これを利用すれば、対話的（interactive）なアプリケーション、たとえばビデオゲームを簡単に開発できます。以下ではマウスイベントの取得方法などを紹介し、簡単なビデオゲームを作ってみます。

マウスの位置

キャンバス上のマウスの x 座標値、y 座標値はシステム定義のグローバル変数 `mouseX`、`mouseY` の値として常にプログラム中のどの位置からでも読み取り可能です。ただし、マウスが実行ウインドウから出たときには変数の値は更新されません。

図 4.43 は `draw()` が呼び出されたときの位置をコンソールに表示するプログラムです。コンソールはエディタのプログラム・エリアの下（ウインドウの左下）にあります。コンソール出力関数 `print()` は引数の値をコンソールに出力する関数です。`draw()` は繰り返し実行されますから、図 4.43 のプログラム実行中はコンソールにマウスの位置が出力され続けます。

注意　p5.js のプログラムでは、引数を与えない `print()` は JavaScript の `window.print()` と見なされます[12]。これはウインドウを印刷する関数ですから、この関数が呼ばれるとブラウザの印刷ダイアログが開きます。もしコンソール出力で改行だけをしたいときには空白文字をひとつ出力する関数呼び出し `print(' ')` や改行文字を出力する呼び出し `print('\n')` を実行してください。

これを応用し、`print()` を `rect()` に変更した図 4.44 のプログラムではマウスの位置に 10×10 の大きさの正方形を描画します。実際にプログラムを動かしてみましょう。

考察　`mouseX`、`mouseY` に類似したシステム定義の変数として `pmouseX`、`pmouseY` も用意されています。`mouseX`、`mouseY` との違いについて調べなさい。

[12]引数の個数によって関数の機能を区別することを関数の多重定義と呼びます（4.1.5 節参照）。ここでもそれが働いたことになります。

```
function setup() {
  createCanvas(500, 500);
}

function draw() {}

function mousePressed() {
  print(mouseX, mouseY);
}
```

図 4.45: マウスが押されたときの位置をコンソールに表示するプログラム

```
function setup() {
  createCanvas(500, 500);
}

function draw() {}

function mousePressed() {
  rect(mouseX, mouseY, 10, 10);
}
```

図 4.46: マウスが押されたときの位置に 10×10 の正方形を描画するプログラム

マウスのクリック

p5.js ではマウスに関する以下の関数群で**イベント駆動処理**（event driven processing）[13]が可能です。いずれもマウスが実行ウインドウ内にあるときのみ有効です。関数の内容の違いに分かりにくいところがありますから注意してください。

`mousePressed()` ... マウスボタンが押された時に自動的に呼び出される関数です。押されたボタンの種類はシステム定義のグローバル変数 `mouseButton` で知ることができます。また、システム定義の `boolean` 型グローバル変数 `mousePressed` が `true` にセットされます。詳細は各自で調べてください。

`mouseReleased()` ... マウスボタンが離された時に自動的に呼び出される関数です。

`mouseClicked()` ... マウスボタンがクリックされた時に自動的に呼び出される関数です。`mousePressed()`、`mouseReleased()` との違いは、マウスが押され、マウスの位置が移動し、マウスが離された場合はクリックと見なされません。位置の移動がない場合に限り、クリックと判断されます。

`mouseMoved()` ... マウスの位置が移動した時に自動的に呼び出される関数です。

`mouseDragged()` ... マウスボタンが押されたまま、位置が移動した時に自動的に呼び出される関数です。

`mouseWheel(MouseEvent event)` ... マウスホイールが回転した時に自動的に呼び出されます。メンバー関数呼び出し `event.getCount()` で回転量を正負の数値として知ることができます。

[13] イベント駆動処理は、アプリケーションフレームワークを用いる最近のプログラム開発の典型的な処理方法です。プログラムは通常、イベント待ち状態にあって、イベントが発生したときに特定の関数が自動的に呼び出されます。

図 4.45 は、マウスが押された時の位置を print() でコンソールに出力するプログラムです。図 4.46 は、マウスが押された時の位置に正方形を描画するプログラムです。この二つのプログラムには本体が空の draw() が定義されていることに注意してください。この関数定義を削除することはできません。削除すると、このプログラムは active sketch ではなく、static sketch と見なされ、アニメーションが発動しないからです。

考察　マウスイベントと同様に、p5.js ではキーボードイベントも使用できます。システム定義のグローバル変数 key、keyCode、keyPressed、関数 keyPressed()、keyReleased()、keyTyped() が用意されています。それらの内容を調べなさい。

文字列の描画

p5.js には、キャンバス（コンソールではありません）に文字列を描画する機能が用意されています。主な関数を以下にまとめます。

text(String str, x, y) ... 文字列 str を座標位置 (x, y) に描画します。文字の色は別途 fill() で指定します。

textSize(size) ... 描画する文字の大きさを画素の単位で指定します。

textAlign(alignX) ... 引数に、文字列の左右の位置付けを左詰（LEFT）、中央並び（CENTER）、右詰（RIGHT）で指定します。デフォルトは左詰です。

textAlign(alignX, alignY) ... 第 1 引数は上と同じです。第 2 引数に、文字列の縦の位置付けを TOP、BOTTOM、CENTER、BASELINE で指定します。これらの違いは自ら調べてください。デフォルトは BASELINE です。

textLeading(leading) ... 文字列が改行コード '\n' を含み、描画が複数行に渡る場合の行間を画素の単位で指定します。デフォルトでは textSize() で指定した値（すなわち行間は 1 行）です。

より凝った描画も可能です。詳細は自ら調べてみてください。描画例は次のゲームの例で紹介します。

4.3.4　モグラたたき風ゲームの開発

ここまで述べた内容を基にして講義担当者が図 4.47 のような簡単なゲームを作ってみました。

```
function setup() {
  createCanvas(500, 500);

  background(255);                    // ゲーム開始直後にキャンバスをクリア
  noStroke();                         // 円の輪郭を表示しない。
  textAlign(CENTER, CENTER);          // 文字列を縦横、指定位置の中央に描画する。

}

const r = 200.0;                      // 円の直径
const interval = 45;                  // 円の表示間隔をフレーム数で指定
const number = 15;                    // 円の総数

let x, y;                             // 円の中心位置
let point = 0;                        // 得点

function draw() {
  if (frameCount == interval * (number + 1)) {
                                        // もし全ての円の表示を超えたならば
    textSize(100);                    // 文字サイズを 100pt に
    fill(255, 0, 0);                  // 文字を赤色で表示
    text(point + "!", 250, 250);      // 得点を表示
    noLoop();                         // 操作待ち状態に入る

  } else if (frameCount % interval == 0) { // 次の円を描画するタイミングならば
    fill(100 + random(155),          // R、G、B の輝度値をそれぞれ乱数で求めるが
      100 + random(155),              // パステル調の明るい色になるように調整
      100 + random(155),
      200);                           // 少しだけ透明感を加える。
    x = random(500);                  // x 座標値を 0~500 の乱数で求める。
    y = random(500);                  // y 座標値を 0~500 の乱数で求める。
    ellipse(x, y, r, r);              // 円を描画する。

    textSize(r / 3);                  // 文字サイズを r/3 に
    fill(255);                        // 文字を白色で
    text(point, x, y);                // 経過得点を表示
  }
}

function mousePressed() {
  let d = dist(mouseX, mouseY, x, y); // マウスの位置と円の中心の距離
  if (d < r/2) point += 1;
}
```

図 4.47: モグラ叩き風プログラム

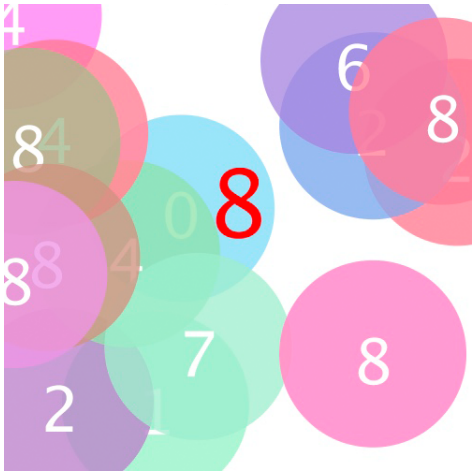


図 4.48: 図 4.47 のプログラムの実行終了時のキャンバス

このゲームでは一定時間毎にキャンバス上のランダムな位置に円が現れます。その円の中にマウスを移動させてクリックすると 1 点ゲットできます。円は全部で 15 個現れ、ゲーム終了後にキャンバスの中央に点数が赤字で表示されます。

このプログラムではまだ解説していない以下の機能を用いています。

`frameCount` ... プログラム開始直後からの描画フレーム数です。開始直後の値が 0 です。関数 `draw()` がそれまでに呼ばれた（実行された）回数と一致します。

`noLoop()` ... 関数 `draw()` を繰り返し呼び出す動作を停止させ、プログラムはマウス、キーボード、メニューの操作待ち状態に入ります。

`loop()` ... 上記の操作待ち状態を解消し、再び `draw()` をフレームレートで繰り返し呼び出す動作が始まります。図 4.47 では用いていません。

上のふたつの関数は、ゲームの実行を止めたり、再開するときに必要です。

以下は、計算を簡単にするユーテリティ関数です。

`random(high)` ... 0 以上、`high` よりも小さい範囲の一様乱数値を求めます。

`random(low, high)` ... `low` 以上、`high` よりも小さい範囲の一様乱数値を求めます。

`dist(x1, y1, x2, y2)` ... 二つの座標値 (`x1`, `y1`) と (`x2`, `y2`) の距離を計算します。同等の計算は

```
sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))
```

でも可能ですが、一般に組み込み関数を用いた方が高速な計算が期待できます。

他にも、三角関数、対数／指数関数、天井関数、床関数などが利用可能です。

考察　プログラムに以下のキーボードイベント処理関数を追加したときに、プログラムの実行がどのように変わるか調べなさい。

```
function keyPressed(){
  loop();
}
```

第VI部

4.4　ラスタライズ

3.2 節において、ベクタ・グラフィックスのデータをラスタ・グラフィックスの画像データへ変換することを**ラスタライズ**と呼びました。p5.js の関数 `line()`、`triangle()`、`rect()`、`ellipse()` などはまさにラスタライズを行なう関数です。これらの関数は図形の幾何情報を引数として受け取り、キャンバス上の特定画素群に色付けを行ない、目的の図形を描画しています。

では、どの画素にどのように色付けを行えば、期待される図形が正しく描画できるのでしょうか。この節ではそのラスタライズのアルゴリズムを概説します。

ところで、「ラスタライズ」とほぼ同じ意味で「キャン・コンバージョン」[14]（scan conversion、走査変換、スキャン変換）という語もよく用いられます。両者の違いは、後者はベクタ・データがラスタ・データへ変換されることに加え、そのラスタ・データが図 3.7 の意味での 1 次元データ（スキャンデータ）に変換されることもそのニュアンスに含んでいる点です。ここではデータを 1 次元化することは考慮しません[15]から、専ら「ラスタライズ」を用います[16]。

4.4.1　点のラスタライズ

2 次元座標上の 点（point）を描画することは最も単純なラスタライズです。数学的な点は大きさを持たない極小の図形ですが、キャンバス上では 1 画素が 1 点に対応するとみなすのが自然です。

p5.js には特定画素に色を付ける以下の関数が用意されています。

`point(x, y)` ... 2 次元座標位置 $(\mathtt{x}, \mathtt{y})$ の画素に `stroke()` で設定した色を塗ります。画素の色は関数 `stroke()` で事前に設定しておきます。

たとえば図 4.49 は、座標点 $(\mathtt{x}, \mathtt{y})$（$0 \leq \mathtt{x}, \mathtt{y} \leq 255$）の位置の各画素を RGB 値 $(\mathtt{x}, \mathtt{y}, 255)$ で色付けするプログラムです。実行結果は図 4.50 です。色のグラデーションを表示されています。

様々な図形のラスタライズは、この `point()` を繰り返し呼び出して実現します。

4.4.2　線分のラスタライズ（授業対象外）

ここでは始点と終点が与えられた線分のラスタライズについて解説します。つまり関数 `line()` の内部で上記の `point()` が実行されているか、考察します。

---

[14] 動詞としてはもちろん “scan convert” ですが、ここでは品詞についてはあまり拘らないことにしています。

[15] スキャンデータに変換する作業はハードウェアに任せればよく、この節ではベクタ・データからラスタ・データを作ることに集中します。

[16] ラスタライズは動詞ですから、動名詞はラスタライジング（rasterizing）、名詞はラスタライゼーション（rasterization）です。しかしテキストでは、これらの語を本文の日本語の中で使用するときに品詞の使い分けには神経質にならず、不自然にならない範囲で「ラスタライズ」を使用します。

```
function setup() {
  createCanvas(256,256);

  for (x = 0; x <= 255; x++)
  {
    for (y = 0; y <= 255; y++)
    {
       stroke(x, y, 255);
       point(x, y);
    }
  }
}
```

図 4.49: `point()` を使ったプログラムの例

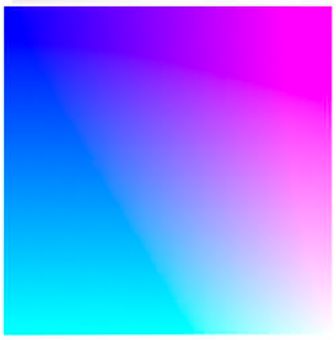


図 4.50: 図 4.49 の実行例

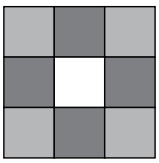


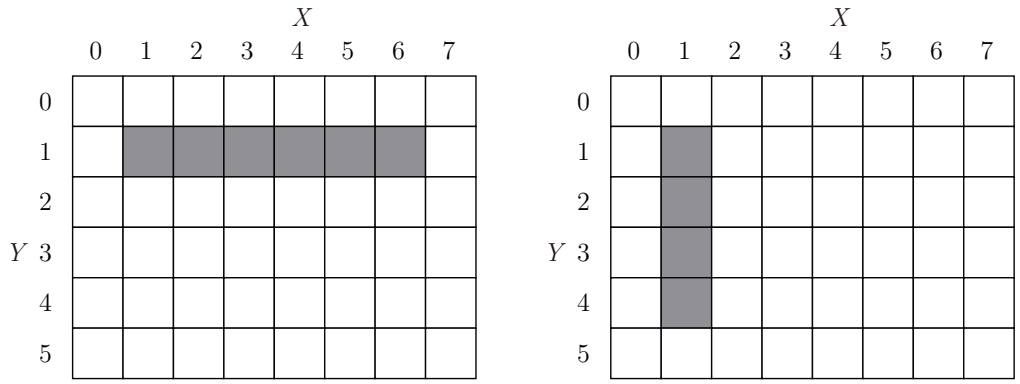
図 4.51: 8 近傍（中心画素に隣接する 8 画素）

なお、ラスタライズの議論では数値のデータ型（浮動小数点数型か整数型か）をしっかり認識することが必須です。しかし p5.js と（とそのベース言語の JavaScript）では暗黙の数値は全て浮動小数点数型であるため、ラスタライズの議論に馴染みません。そこで以下のラスタライズの議論では、データ型を明確に立て分ける Processing（とそのベース言語の Java）を用いて行うこととします。

線分の定義

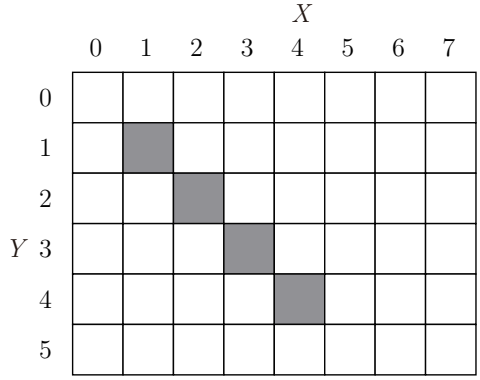
線分は以下の定義を満たすものとします。

1. 線分の始点と終点の座標値はそれぞれ整数値の組 (X_s, Y_s)、(X_e, Y_e) で与えられるものとし、その始点、終点に対応する画素は必ず色付けすること。
2. 個々の色付けされた画素の 8 近傍（図 4.51 参照）に色付けされた画素が一つ以上隣接すること。
3. 8 近傍で隣接する画素の列によって始点から終点まで連結されていること。
4. 色付けされた画素の数は最小であること。

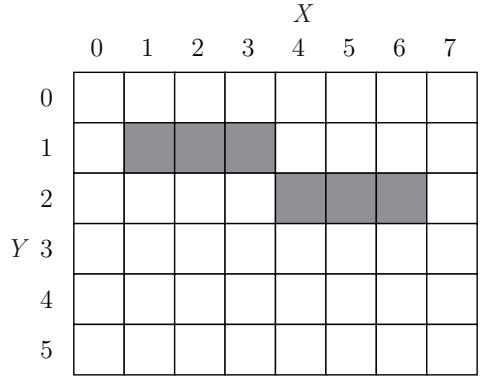


(a) $(1,1) \rightarrow (6,1)$

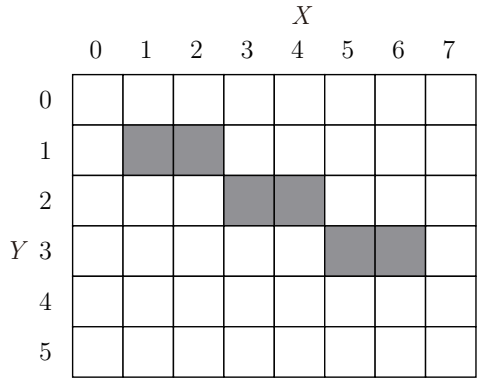
(b) $(1,1) \rightarrow (1,4)$



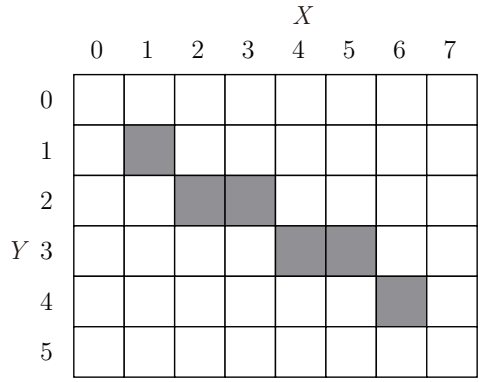
(c) $(1,1) \rightarrow (4,4)$



(d) $(1,1) \rightarrow (6,2)$



(e) $(1,1) \rightarrow (6,3)$



(f) $(1,1) \rightarrow (6,4)$

図 4.52: 線分のラスタライズ例

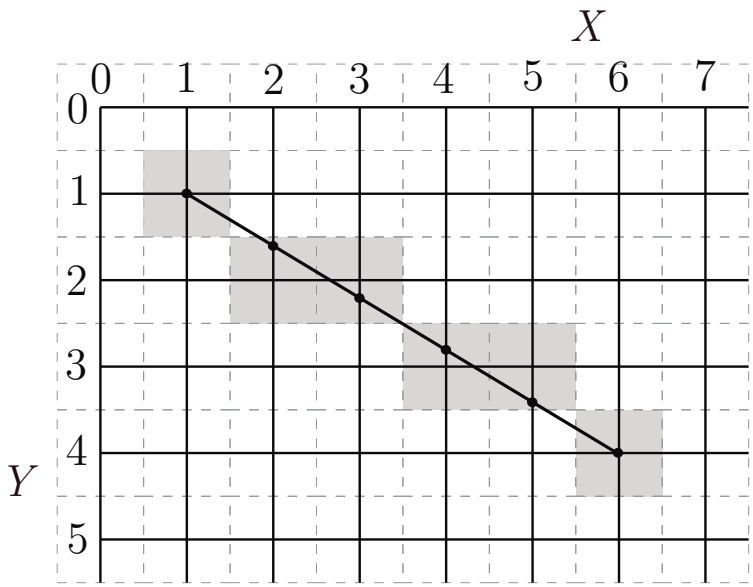


図 4.53: 図 4.52 (f) の線分（黒い格子が画素点を表す。黒点は線分と $X = 1, 2, ..., 6$ の交差点を表す。黒点に最も近い画素点に色付けする）

上の定義は一見、複雑ですが、図 4.52 の例題を見れば一目瞭然です。

たとえば座標点 $(1, 1)$ から座標点 $(6, 1)$ への線分を引いた例が図 4.52 (a) です。$(1, 1)$ から $(1, 4)$ への線分を引いた例が同図 (b) です。$(1, 1)$ から $(4, 4)$ への線分を引いた例が同図 (c) です。これらの例の線分の引き方（色付けする画素の選び方）は自明であり、異論はないでしょう[17]。次に、$(1, 1)$ から $(6, 2)$ 、 $(6, 3)$ 、 $(6, 4)$ へ線分を引いた例がそれぞれ図 4.52 (d)、(e)、(f) です。これらの色付けされた画素のパターンの規則性はどうなっているのでしょうか。

基本アルゴリズム

2 次元 X-Y 平面上に始点 (X_s, Y_s) と終点 (X_e, Y_e) が与えられたとき、図 4.52 のような線分を自動的に描く（線分をラスタライズする）アルゴリズムを考えてみましょう。

図 4.52 の 6 例を眺めると、(b) の場合を除いて、$X = 1, 2, 3, ...$ のそれぞれの X 座標値について縦方向にはひとつの画素のみが色付けされています。よって、X について `for` ループを回し、各 X について色付けすべき画素の Y 座標値を求めるプログラムを作ればよいはずです。

Y 座標値の求め方を詰めるために、図 4.52 (f) の場合を例にしてさらに検討を進めます。

まず、この例の線分の始点 $(X_s, Y_s) = (1, 1)$ と終点 $(X_e, Y_e) = (6, 4)$ を通る直線の方程式は以

[17] 線分の幅（線分の方向と垂直方向の画素の厚み）に注目すると、 (a)、(b) の場合、1 画素です。(c) の場合、細い場所では 0 画素、太い場所では $\sqrt{2}$ 画素、よって平均で $\sqrt{2}/2$ 画素（$= 0.7071...$ 画素）となり、1 画素よりも細いのですが、この程度の差は受け入れることにしましょう。平均の線幅を厳密に 1 画素にするラスタライズは不可能です。

```
final int Xs = 1;
final int Ys = 1;
final int Xe = 6;
final int Ye = 4;

for (int X = Xs; X <= Xe; X++) {
  int Y = round((float(Ye-Ys)/float(Xe-Xs))*(X-Xs)+Ys);
  point(X, Y);
}
```

図 4.54: 線分を point() でラスタライズするプログラム

下の通りです（図 4.53 参照）。

$$Y = \frac{Y_e - Y_s}{X_e - X_s} \cdot (X - X_s) + Y_s = \frac{4-1}{6-1} \cdot (X-1) + 1 = 0.6 \cdot X + 0.4$$

$X = 1$ のときには $Y = 1$ ですから画素点 $(1, 1)$ に色付けします。次に $X = 2$ のときには $Y = 1.6$ です。そこで、座標点 $(2, 1.6)$ に最も近い画素点を色付けすればよいでしょう。そのような画素点は $Y = 1.6$ を四捨五入した $Y' = 2$ ですから、次に $(2, 2)$ を色付けしましょう。同様に $X = 1, ..., 6$ について Y とその四捨五入 Y' 求めれば以下の通りです。

X	Y	Y'
1	1.0	1
2	1.6	2
3	2.2	2
4	2.8	3
5	3.4	3
6	4.0	4

結局、線分をラスタライズするには、

1. 直線の方程式を求め、
2. 各 X に対する Y を求め、さらに
3. それを四捨五入した Y' を求めればよい

となります。このアイディアを実装したプログラムが図 4.54 です。各 X について対応する Y を直線の方程式に基づいて float 型で計算し、四捨五入しています。四捨五入には Processing の関数 round() を用いています[18]。

[18] 式 (float(Ye-Ys)/float(Xe-Xs)) をループ中で何度も計算することを非効率と考えるかもしれません。しかし、最近のコンパイラはループ中で値が変わらない式（ループ不変式）の計算をループの前に移動して 1 回の計算で済ませる最適化を自動的に行ないますから、問題ありません（たぶん）。

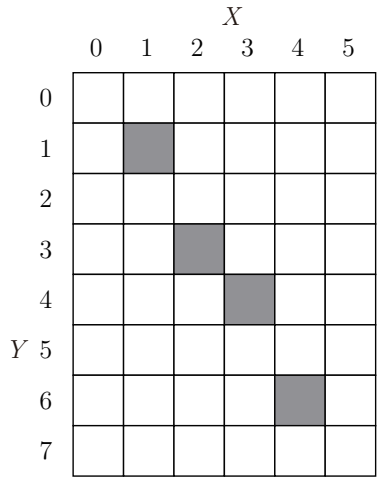


図 4.55: 図 4.54 のプログラムで $(1,1) \to (4,6)$ を描画した場合

さて次に、図 4.54 のプログラムで $(1,1)$ から $(4,6)$（$\neq (6,4)$）へ線分を引いた場合の実行結果を図 4.55 に示します。線分が途切れており、不都合であることに気づきます。途切れる理由は、上に述べた方法で (X, Y, Y') を計算すると、以下の通りです。

X	Y	Y'
1	1.0	1
2	2.66...	3
3	4.33...	4
4	6.0	6

つまり、線分の傾きが 1 よりも大きい場合（今の例では

$$(Y_e - Y_s)/(X_e - X_s) = (6-1)/(4-1) = 5/3 > 1$$

）には、X が 1 増えるとき、四捨五入後の Y' の値が直前の Y' に比べて 2 以上増えることがあり、線分が途切れるのです。これを解決する方法は単純です。線分の傾きが 1 を超える場合には X と Y の役割を入れ替えればよいのです。つまり、以下の直線の方程式：

$$X = \frac{X_e - X_s}{Y_e - Y_s} \cdot (Y - Y_s) + X_s = \frac{4-1}{6-1} \cdot (Y-1) + 1 = 0.6 \cdot Y + 0.4$$

を用い、Y についてループを回し、各 X を四捨五入した値 X' を求めればよいのです。以下はその方法での計算結果です。

```
final int Xs = 1;
final int Ys = 1;
final int Xe = 4;
final int Ye = 6;

if (Xe-Xs >= Ye-Ys){
  for (int X = Xs; X <= Xe; X++) {
    int Y = round((float(Ye-Ys)/float(Xe-Xs))*(X-Xs)+Ys);
    point(X, Y);
  }
} else {
  for (int Y = Ys; Y <= Ye; Y++) {
    int X = round((float(Xe-Xs)/float(Ye-Ys))*(Y-Ys)+Xs);
    point(X, Y);
  }
}
```

図 4.56: 線分の傾きを考慮してラスタライズするプログラム

Y	X	X'
1	1.0	1
2	1.6	2
3	2.2	2
4	2.8	3
5	3.4	3
6	4.0	4

線分の傾きを考慮したプログラムは図 4.56 の通りです。図 4.52 (b) の垂直な線分もこのプログラムでラスタライズできます。なお、`if` 文の条件式 `Xe-Xs >= Ye-Ys` は $(Y_e - Y_s)/(X_e - X_s) \leq 1$ の言い換えです。除算を省くために式を変形しました[19,20]。

もうひとつ解決すべき問題が残っています。ここまでは始点 (X_s, Y_s) と終点 (X_e, Y_e) について、$X_s \leq X_e$ かつ $Y_s \leq Y_e$ となる線分（図的に言えば、右下がりの線分）のみを検討してきました。そうでない場合にも一般化するならば、以下の 3 つの条件を調べる必要があります。

1. $X_s \leq X_e$ が成り立つか否か
2. $Y_s \leq Y_e$ が成り立つか否か
3. $|X_e - X_s| \geq |Y_e - Y_s|$ が成り立つか（線分の傾きの絶対値は 1 以下か否か）

結局、線分のラスタライズは始点と終点の相対的な位置関係（座標値の大小関係）、線分の傾きについて、図 4.57 のように 8 通りに場合分けして処理せねばなりません。

[19] 除算はきわめて計算コストの高い演算ですから、これを避ける努力は重要です。
[20] `>=` から `=` を除き、`Xe-Xs > Ye-Ys` としても大差ありません。両辺が等しいとき、`then` 部と `else` 部のどちらを実行しても同じ描画になるからです。

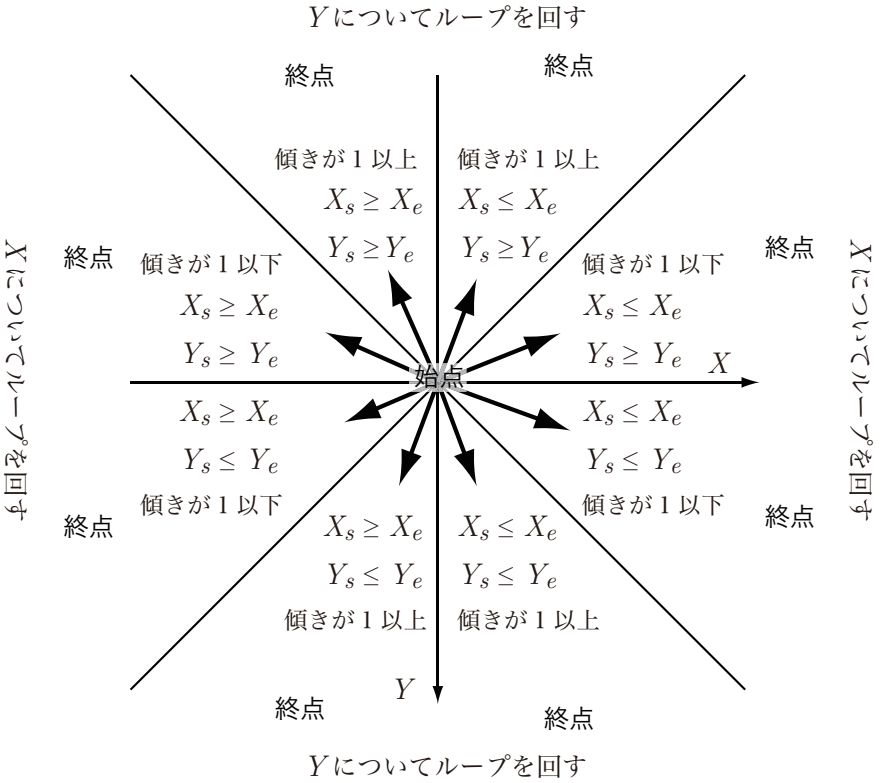


図 4.57: 線分のラスタライズの場合分け（図中の「傾き」は「線分の傾きの絶対値」のこと。境界上についてはどちらに場合分けしてもよい）

考察　任意の始点、終点が与えられた場合に対応する、線分のラスタライズのプログラムを作りなさい。

ブレゼンハムのアルゴリズム

図 4.54、図 4.56 のプログラムでは `float` 型データを使用しています。しかし、`float` 型データの演算は `int` 型データの演算よりもコストが高い[21]ため、`float` 型データを用いない方がプログラムの高速実行に有利です。特に線分の描画のようなプリミティブな処理は可能な限り高速であるべきです。ここで紹介する高速化アルゴリズムは、1965 年にアメリカ人研究者ブレゼンハム（Bresenham）が考案したもので、先述の線分をラスタライズするプログラムを `int` 型演算だけで実現しています。浮動小数点演算が整数演算で代替できることを不思議に思うかもしれません。それができる理由は、線分のラスタライズに用いる浮動小数点数はせいぜい有理数程度の複雑さしか持たないためです。有理数は二つの整数を用いて、m/n と表現できます。よって簡単に言えば、プログラムに現れる全ての式を n 倍すれば全ての計算は整数演算で可能となるのです。

[21] 演算時間の比較で言えば、現在のほとんどの商用プロセッサにおいて `float` 型の演算は `int` 型の演算の 4 倍以上の演算時間を要します。

このテキストではブレゼンハムのアルゴリズムを 3 段階のプログラムの変形によって実現します。ただし簡単のため、ここでは最初に用いた図 4.54 のプログラムを対象とします。図 4.57 のように 8 通りの場合全てを考慮するプログラムについては各自で考察してください。

第 1 段階（ループ構造の変形）　図 4.58 のループでは、ループ本体で毎回、直線の方程式

$$Y = ((Y_e - Y_S)/(X_e - X_s)) \cdot (X - X_s) + Y_s$$

を計算し、それを四捨五入して変数 `Y` の値を求めています。ここで、Y の値は線分の傾き $(Y_e - Y_S)/(X_e - X_s)$ に比例して増えることに注目してください。また、四捨五入は `if` 文で代替実行できます。この性質を利用すれば、図 4.58 のループは図 4.59 のように置換できます。ここに `X` は主ループ変数、`Y` は補助ループ変数です。変数 `e`（error（誤差）の意味）は四捨五入で `Y` の値が 1 増えるか否かを知るために使用し、`e` の値が 0.5 を超えた場合には `Y` の値を 1 増やし（“`Y++`”）、その代わりに `e` の値を 1 減らします（“`e -= 1.0`”）。

第 2 段階（演算の整数化）　第 1 段階の変形後のループでは `float` 型変数 `e` を用いていますが、先に述べたように、これを `int` 型変数へ変形できれば高速化が期待できます。

図 4.59 のプログラムにおいて `e` を `int` 型変数へ変形するためには、`e` に関する代入文 “`e += double(Ye-Ys)/double(Xe-Xs)`” の右辺の分母 `double(Xe-Xs)` を除去せねばなりません。また、`e` に関する条件式 “`e > 0.5`” の右辺 `0.5` を整数化せねばなりません。そこで、プログラムの中で変数 `e` が関連する箇所を全て `2*(Xe-Xs)` 倍します。これによって `e` を `int` 型として宣言可能です。変更後のプログラムは図 4.60 の通りです。

第 3 段階（条件式の簡単化）　最後の変形はアルゴリズムに関する議論ではなく、特定の種類のプロセッサ[22]の機械語命令に関するプログラム最適化です[23]。

機械語命令の中には、比較命令ではないにも関わらず、その命令の実行終了後、演算結果が「0 に等しいか否か」、「負数か否か」をプロセッサ内のフラグレジスタにセットするように設計されているものがあります。これを利用すると、比較命令の実行を省略できるため、プログラムの実行時間が短縮できると期待できます。そこで、図 4.60 のプログラム中の条件式 “`e > (Xe-Xs)`” を “`e > 0`” に変形しましょう。プログラムの意味を変えずこれを実現するためには、プログラム中の `e`

[22] インテル x86 系プロセッサがこれに相当します。x86 系とはインテルが 1978 年代に開発した 16bit プロセッサ 8086 とその後継プロセッサの総称、より正確には、8086 の基本命令セットを実行可能なプロセッサの総称です。現在の PC に搭載されているプロセッサはほとんど全て x86 系です。30 年以上も前の技術が現代に生き残っている理由は、プロセッサ技術が 30 年間まったく進歩しなかったためではなく、過去に開発したソフトウェア遺産を使い続けるためにはハードウェアは変化しない方が都合がよいという事情があるためです。現在の x86 系プロセッサは最新鋭のプロセッサ・アーキテクチャに x86 の衣を被った形をしており、プロセッサの実行速度はもちろん 30 年前から比べると格段に高速化しています。

[23] アルゴリズムとは、特定のプログラミング言語、プロセッサを想定しない抽象度の高い手順のことです。その意味で言えば、この変形はアルゴリズムに関する工夫ではなく、プログラム最適化（コード最適化）と呼ぶ方が適切でしょう。

```
for (int X = Xs; X <= Xe; X++) {
  int Y = round((float(Ye-Ys)/float(Xe-Xs))*(X-Xs)+Ys);
  point(X, Y);
}
```

図 4.58: 基本アルゴリズムのプログラム（図 4.54 のループを再掲）

```
float e = 0.0;
for (int X = Xs; X <= Xe; X++) {
  point(X, Y);

  e += float(Ye-Ys)/float(Xe-Xs);
  if (e > 0.5) {
      Y ++;
      e -= 1.0;
  }
}
```

図 4.59: ブレゼンハムのアルゴリズムに基づく第 1 段階の変形

```
int e = 0;                    // = 0 * 2*(Xe-Xs)   結果として変化しません
for (int X = Xs, Y = Ys; X <= Xe; X++){
  point(X, Y);

  e += 2*(Ye-Ys);             // = double(Ye-Ys)/double(Xe-Xs) * 2*(Xe-Xs)
  if (e > Xe-Xs) {            // = 0.5 * 2*(Xe-Xs)
    Y ++;
    e -= 2*(Xe-Xs);           // = 1.0 * 2*(Xe-Xs)
  }
}
```

図 4.60: ブレゼンハムのアルゴリズムに基づく第 2 段階の変形

```
int e = -(Xe-Xs);                       // 右辺 = 0-(Xe-Xs) 。
for (int X = Xs, Y = Ys; X <= Xe; X++) {
  point(X, Y);

  e += 2*(Ye-Ys);                       // 値のスライドでは不変
  if (e > 0) {                          // 右辺 = (Xe-Xs)-(Xe-Xs) = 0
    Y ++;
    e -= 2*(Xe-Xs);                     // 値のスライドでは不変
  }
}
```

図 4.61: ブレゼンハムのアルゴリズムに基づく第 3 段階の変形

表 4.1: 線分をラスタライズするプログラムの実行時間

プログラムの種類	実行時間（1 画素当たり）	比率
基本アルゴリズム	1.18	1.00
第 1 段階の変形後（ループ構造の変形）	1.62	1.37
第 2 段階の変形後（演算の整数化）	0.65	0.55
第 3 段階の変形後（条件式の簡単化）	0.67	0.57

実行時間の単位はナノ秒 $= 10^{-9}$ 秒

に関連する箇所で（`Xe-Xs`）分だけ値を減らす方向へスライドさせます。結果、図 4.61 のプログラムを得ます。これが最終形です。

しかし、これは図 4.57 の 8 種類の場合分けのひとつをプログラム化したものに過ぎません。実際には、他の 7 種類についても同様のプログラムを用意する必要があり、単純には 8 倍のプログラムサイズとなります。プログラムサイズを小さく抑えたいならば、プログラム構造を巧妙に作る必要があります。

寄り道：ブレゼンハムのアルゴリズムは本当に高速なのか？

ブレゼンハムのアルゴリズムは本当に高速なのでしょうか。

詳細な実験方法は省略します[24]が、図 4.58〜図 4.61 の四つのプログラムの実行時間を実際に測定してみました。その結果を表 4.1 に載せました。

基本アルゴリズムのプログラムの実行時間は、各画素の座標値を計算する毎に 1.18 ナノ秒（$= 1.18 \times 10^{-9}$ 秒）掛かりました。そのプログラムのループ構造を変形した第 1 段階のプログラムでは 1.62 ナノ秒と返って遅くなっています。この理由は、基本アルゴリズムのプログラムのループ本体には `if` 文が含まれていませんが、第 1 段階のプログラムには `if` 文が含まれています。最近のプロセッサは、条件分岐の処理が低速であるため、この結果になっていると思われます。

次に第 2 段階のプログラムは第 1 段階のそれよりも 2 倍以上高速、基本アルゴリズムよりも約 2 倍高速です。これは `double` 型演算を除去した成果であり、私たちの目論み通りでした。

第 3 段階のプログラムは第 2 段階のそれよりもやや遅くなっています。以前のプロセッサでは第 3 段階の変形（すなわち、条件式の簡略化）は有効だったのかもしれませんが、現在の最新鋭のプロセッサではむしろ逆効果になっているようです。プロセッサの進歩もさることながら、コンパイラの進歩もめざましく、プログラマが小技を駆使しても無駄なのかもしれません。

[24] アップル Mac Mini（Intel Core i7、2.6GHz）の UNIX 環境上において、`point()` の呼び出しを配列への値の書き込みで代用し、`c++ -O3` でコンパイルし、$|X_e - X_s| = 10000$ の線分を引いたときのマシンサイクル数をカウントしました。実際のフレームメモリへの画素値の書き込みは変数（メインメモリ）への書き込みよりも遅いため、表 4.1 の実行時間は実際の描画よりも小さい値が出ていると予想されます。

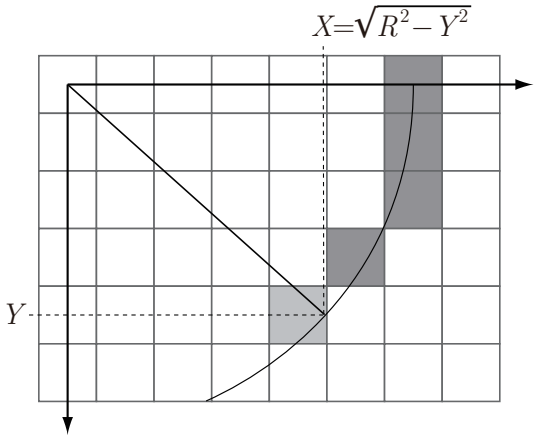


図 4.62: 基本アルゴリズムによる円周のラスタライズ（各 Y に対して X を求め、最近傍の画素点を決定する場合）

4.4.3 円周のラスタライズ（授業対象外）

線分をラスタライズした手法は曲線のラスタライズにも応用できます。

基本アルゴリズム

基本アルゴリズムでは、線分の場合と同様に、X または Y でループを回しながら 2 次元平面における円の方程式の計算を行い、値を四捨五入して画素点の位置を求めて色付けを行ないます。

まず、円の方程式は以下の通りです。

$$X^2 + Y^2 = R^2$$

これを変形すると、以下の二つの式が導出できます。

$$\begin{aligned} Y &= \pm\sqrt{R^2 - X^2} \\ X &= \pm\sqrt{R^2 - Y^2} \end{aligned}$$

これらの式を用いて色付け画素点の位置を求めます（図 4.62 参照）。

ところで、円の形状の対称性から、もし (X, Y) が色付けする画素点であるならば、$(-X, Y)$、$(X, -Y)$、$(-X, -Y)$、(Y, X)、$(-Y, X)$、$(Y, -X)$、$(-Y, -X)$ も色付け画素点です。よって、色付け画素点は全周の 1/8 だけ計算すれば済むことになります（図 4.63 参照）。そこでここでは第一象限（$X \geq 0,\ Y \geq 0$）の角度 $0 \leq \theta \leq \pi/4$ の範囲内 — これは $0 \leq Y \leq X$ と等価な条件です — で具体的に (X, Y) を求めることを考えます。

ラスタライズのプログラムは図 4.64 です。$0 \leq \theta \leq \pi/4$ の範囲の円の接線の傾きは 1 以上のため、X ではなく、Y でループを回していることに注意してください。なお、円の半径は 200、円の中心は $(300, 250)$ と設定しました。なお、実行結果は自明なので示しません。

高速化アルゴリズム

線分におけるブレゼンハムのアルゴリズムのように、円周をラスタライズするプログラムから `float` 型データを除去し、アルゴリズムを整数演算化する方法が知られています。前提条件や使用する漸化式によって数種類の方法が知られていますが、ここではそのひとつを紹介します。

整数演算化　図 4.65 を用いて解説します。今、 $(R, 0)$ の画素点から色付けを始めて図 4.65 の画素点 $A = (X_i, Y_i)$ まで色付けを終えたと仮定します。そうすると、次に色付けすべき画素点は $B = (X_i, Y_i + 1)$ または $C = (X_i - 1, Y_i + 1)$ の 2 択であり、より円周に近い方を選択すべきです。

基本アルゴリズムでは、$Y = Y_i + 1$ のときの円周上の点を $D = (\sqrt{R^2 - (Y_i + 1)^2}, Y_i + 1)$ と計算し、

- B と D との距離 $|BD|$ と C と D との距離 $|CD|$ を比べ、

その距離が短い方を色付け点としました（距離の比較を四捨五入で代替しました）。しかし、この判定法には点 D の X 座標値の平方根の計算が必要とされ、整数演算化の大きな障害となります。この平方根は単純な式の変形では除去できません。そこで考え方を少し変えます。まず、円の中心点 O と点 B を通る直線が円周と交わる点を新たに B' とします。また、O と C を通る直線が円周と交わる点を C' とします。そして、

- 距離 $|BO|$ と距離 $|B'O|$（実は半径 R に等しい）との二乗の誤差[25]

$$\mathrm{E}_B = ||BO|^2 - |B'O|^2| = ||BO|^2 - R^2|$$

と 距離 $|CO|$ と距離 $|C'O|$（やはり半径 R に等しい）との二乗の誤差

$$\mathrm{E}_C = ||CO|^2 - |C'O|^2| = ||CO|^2 - R^2|$$

を比べ、その差が小さい方を色付け点として選択する

という方法を採用します。なお、整数演算化の都合とは言え、画素点を選択する条件を基本アルゴリズムから変えてしまったため、ラスタライズの結果も基本アルゴリズムのそれとは厳密には同じではありません。しかし半径 R が十分大きい場合にはその差はわずかです。

[25] ここで言う「二乗の誤差」は二乗誤差とは異なります。二乗誤差は、誤差の二乗を意味しますが、ここで用いる二乗の誤差は、二乗した値の誤差です。

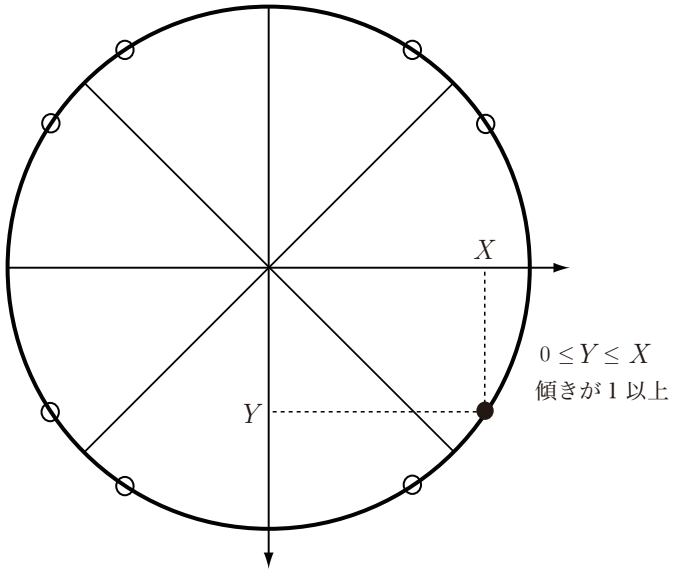


図 4.63: 円の対称性に基づく省力化（黒点の位置が決まれば、その他の 7 個の白点の位置は簡単に求められる）

```
size(600,500);
final int Xoffset = 300;
final int Yoffset = 250;
final int R = 200;

for (int Y = 0, X = R; Y <= X; Y++){
  X = round(sqrt(float(R*R-Y*Y)));

  point( X+Xoffset,  Y+Yoffset);
  point(-X+Xoffset,  Y+Yoffset);
  point( X+Xoffset, -Y+Yoffset);
  point(-X+Xoffset, -Y+Yoffset);
  point( Y+Xoffset,  X+Yoffset);
  point(-Y+Xoffset,  X+Yoffset);
  point( Y+Xoffset, -X+Yoffset);
  point(-Y+Xoffset, -X+Yoffset);
}
```

図 4.64: 円周を point() でラスタライズするプログラム（基本アルゴリズム）

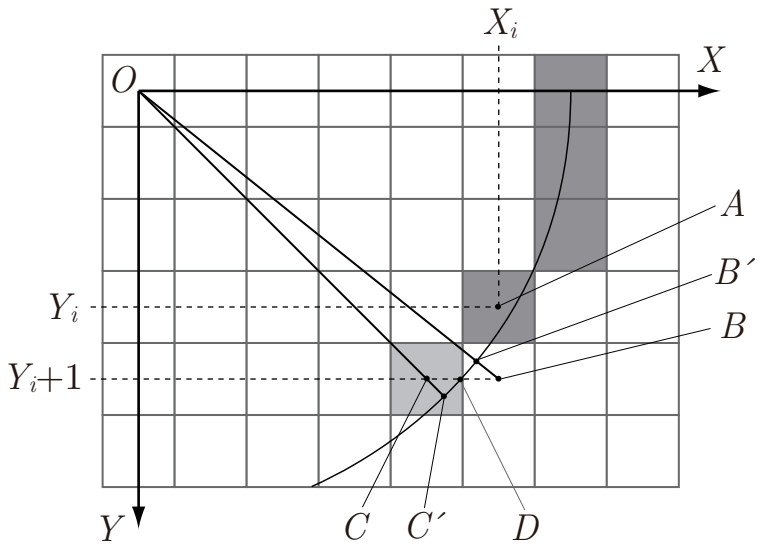


図 4.65: 高速化アルゴリズムによる円周のラスタライズ（円周との二乗の誤差が小さい画素点を選択）

さて、それぞれの誤差は具体的に以下のように書き下すことができます。

$$\begin{aligned} \mathrm{E}_B &= |X_i^2 + (Y_i+1)^2 - R^2| \\ \mathrm{E}_C &= |(X_i-1)^2 + (Y_i+1)^2 - R^2| \end{aligned}$$

B が選択される条件は $\mathrm{E}_B < \mathrm{E}_C$ です[26]。この条件を、以下のように変形していきます。

$$\begin{aligned} \mathrm{E}_B &< \mathrm{E}_C, \\ \mathrm{E}_B^2 &< \mathrm{E}_C^2, \qquad \text{(両辺を二乗)} \\ |X_i^2 + (Y_i+1)^2 - R^2|^2 &< |(X_i-1)^2 + (Y_i+1)^2 - R^2|^2, \qquad \text{(式を展開)} \\ (X_i^2 + (Y_i+1)^2 - R^2)^2 &< ((X_i-1)^2 + (Y_i+1)^2 - R^2)^2, \qquad \text{(}|\ |\text{を除去)} \end{aligned}$$

さらに右辺を左辺に移項し、整理すると

$$\begin{aligned} (X_i^2 + (Y_i+1)^2 - R^2)^2 - ((X_i-1)^2 + (Y_i+1)^2 - R^2)^2 &< 0, \\ (2X_i-1)(2X_i^2 + 2Y_i^2 - 2R^2 - 2X_i + 4Y_i + 3) &< 0, \end{aligned}$$

ここで $X_i > 0$ より、常に $(2X_i-1) > 0$ が成り立ちますから、両辺を正数 $(2X_i-1)$ で割り、以下の式を得ます。

$$2X_i^2 + 2Y_i^2 - 2R^2 - 2X_i + 4Y_i + 3 < 0$$

[26] $\mathrm{E}_B \leq \mathrm{E}_C$ でも大差ありません。

```
for (int Y = 0, X = R; Y <= X; Y++){
  point( X+Xoffset,  Y+Yoffset);
  point(-X+Xoffset,  Y+Yoffset);
  point( X+Xoffset, -Y+Yoffset);
  point(-X+Xoffset, -Y+Yoffset);
  point( Y+Xoffset,  X+Yoffset);
  point(-Y+Xoffset,  X+Yoffset);
  point( Y+Xoffset, -X+Yoffset);
  point(-Y+Xoffset, -X+Yoffset);

  if (2*X*X+2*Y*Y-2*R*R-2*X+4*Y+3 > 0) {
    X--;
  }
}
```

図 4.66: 円周を point() でラスタライズするプログラム（整数演算化後）

つまり、上の式が成り立つとき、点 B を選択し、さもなくば点 C を選択すればよいのです。B を選択するときには X は不変（$X_{i+1} = X_i$）のままにし、C を選択するときには X を 1 減らします（$X_{i+1} = X_i - 1$）。これをプログラム化したものが図 4.66 です。

条件式の高速演算　条件式 $2X_i^2 + 2Y_i^2 - 2R^2 - 2X_i + 4Y_i + 3$ は多数の乗算、加減算を含む高コストな計算式です。これを高速化するために、この条件式を一旦

$$E_i = 2X_i^2 + 2Y_i^2 - 2R^2 - 2X_i + 4Y_i + 3$$

と置きます。そして、数列 E_i に関する漸化式を求めると以下が成り立ちます。

$$\begin{aligned} E_0 &= 3 - 2R, \qquad (\because X_0 = R,\ Y_0 = 0) \\ \mathrm{E}_{i+1} &= \begin{cases} E_i + 4Y_i + 6, & \text{もし } E_i < 0 \text{ ならば,} \\ E_i + 4(Y_i - X_i) + 10, & \text{もし } E_i \geq 0 \text{ ならば.} \end{cases} \end{aligned}$$

この漸化式を用いれば演算数が激減します。

これをプログラム化したものが図 4.67 です[27]。実行結果は自明なため載せません。

ここまで円周のラスタライズを解説してきましたが、円周の一部（円弧）をラスタライズすることも重要です。また楕円のラスタライズも重要です。それらの方法は上に解説した内容をさらに拡張改良することになります。本質的な難しさはありませんが、これ以上の説明は省略します。

寄り道：高速化アルゴリズムは本当に高速なのか？

円周についても、高速化アルゴリズムが本当に効果的なのか調べてみました。

[27] ここに紹介した手法は Michener's circle algorithm、Bresenham's circle algorithm などと呼ばれていますが、呼び方の最も確からしい情報にたどり着けなかったため、この授業では名付けて呼ばないこととします。

```
int e = 3-2*R;
for (int Y = 0, X = R; Y <= X; Y++){
  point( X+Xoffset,  Y+Yoffset);
  point(-X+Xoffset,  Y+Yoffset);
  point( X+Xoffset, -Y+Yoffset);
  point(-X+Xoffset, -Y+Yoffset);
  point( Y+Xoffset,  X+Yoffset);
  point(-Y+Xoffset,  X+Yoffset);
  point( Y+Xoffset, -X+Yoffset);
  point(-Y+Xoffset, -X+Yoffset);

  if (e < 0) {
     e += 4*Y+6;
  } else {
    e += 4*(Y-X)+10;
    X--;
  }
}
```

図 4.67: 円周を point() でラスタライズするプログラム（条件式計算の高速化後）

表 4.2: 円周をラスタライズするプログラムの実行時間

プログラムの種類	実行時間（1 画素当たり）	比率
基本アルゴリズム	6.54	1.00
高速化アルゴリズム（整数演算化）	4.62	0.71
高速化アルゴリズム（条件式の高速化）	2.35	0.36

実行時間の単位はナノ秒 $= 10^{-9}$ 秒

詳細な実験方法は省略します[28]が、図 4.64、図 4.66、図 4.67 の三つのプログラムの実行時間を測定しました。その結果を表 4.2 に載せました。

プログラムの実行速度が段階毎に高速になっていることが分かります。最大 3 倍の高速化になっており、高速化のための工夫は有意義であったと言えます。次に、線分の場合の実験結果（表 4.1）と比較してみます。線分では最速 0.67 ナノ秒/画素であったのに対して、円周では 2.35 ナノ秒/画素と 3 倍以上の時間です。円周の場合、線分の場合に比べ、演算内容が本質的に複雑であるためです。

[28] アップル Mac Mini（Intel Core i7、2.6GHz）の UNIX 環境上において、point() の呼び出しを配列への値の書き込みで代用し、c++ -O3 でコンパイルし、半径 10000 の円周を描いたときのマシンサイクル数をカウントしました。実際のフレームメモリへの画素値の書き込みは変数（メインメモリ）への書き込みよりも遅いため、表 4.2 の実行時間は実際の描画よりも小さい値が出ていると予想されます。

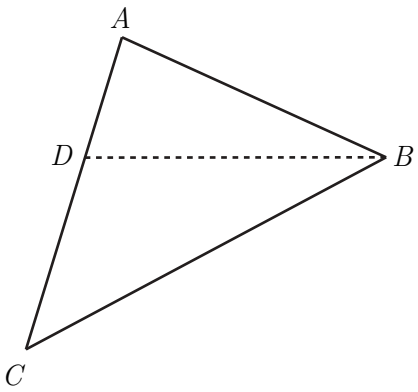


図 4.68: 任意の三角形のラスタライズ（水平線で三角形を分割）

考察（やや難易度高い）　上述の円周の高速ラスタライズ・プログラムの導出の手法 — (1) まず、次の画素点を 2 択で選ぶ判別式を作り、(2) その判別式を漸化式に直す — を線分について適用すれば図 4.61 と等価なプログラムを導出できます。その導出過程を示しなさい。

考察（やや難易度高い）　放物線 $y = x^2$ の高速なラスタライズ法について検討しなさい。

4.4.4　面のラスタライズ（授業対象外）

線のラスタライズの発展させ、面をラスタライズする（面の塗りつぶしを複数の画素点の色付けで行なう）アルゴリズムを検討します。

面の形状は多種多様ですが、最も基本的な形状は三角形です。その周囲が線分で構成される多角形は、それにどんな凹凸があろうとも、複数枚の三角形の組み合わせで必ず表すことができます。3 次元 CG（6 章）の立体形状も、それが曲面を含まないならば、三角形を組み合わせて表すことができます。その意味で、三角形のラスタライズは CG の基本技術です。

ところで、三角形の 3 頂点を、その Y 座標値の大小順に A、B、C と並べ、頂点 B の Y 座標値で三角形 ABC を切断するならば、ABC は 2 つの三角形 ABD と CBD に分割できます（図 4.68 参照）。この 2 つの三角形の特徴は、3 辺中の 1 辺が必ず水平（X 座標軸と平行）であることです。このような三角形のラスタライズ・アルゴリズムは以下のように単純になります。以下では、図 4.68 の三角形 ABD について検討しましょう。

1 辺が水平な三角形の定義

図 4.69 のような三角形の、X 軸に平行な辺を底辺（bottom line）、それに対峙する頂点を頂上点（top point）と呼ぶこととします. そして、底辺の両端点を左から (X_{bs}, Y_b)、(X_{be}, Y_b) と表し、

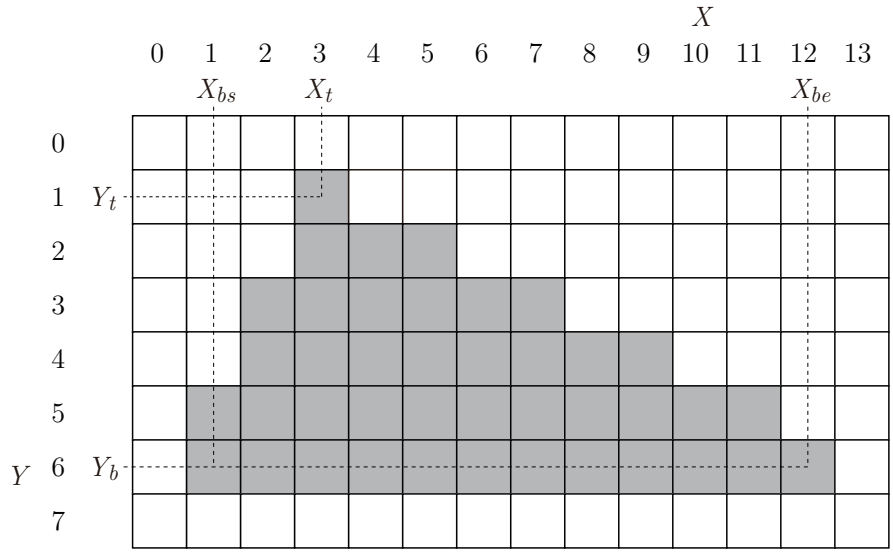


図 4.69: 三角形のラスタライズ例

頂上点の座標値を (X_t, Y_t) と表します[29]。ここに、$X_{bs} \leq X_{be}$、$Y_t \leq T_b$ が成り立つものと約束します[30]。なお、$Y_t \geq T_b$ の場合（たとえば図 4.68 の三角形 CBD）は、図 4.69 の三角形が上下逆転したような形状となるだけですから、本質的に同じです。

三角形のラスタライズとは、この底辺と頂点で囲まれた領域内の画素点を全て色づけすることです。

基本アルゴリズム

塗りつぶしのアイディアは、図 4.70 に示しているように、水平方向に画素を塗りつぶす処理を Y 軸方向に繰り返すことです。そのためには、各 Y について塗りつぶしの先頭の画素（図中の各行の最左の画素）と末尾の画素（図中の各行の最右の画素）の位置を求める必要があります。これには 4.4.2 節のラスタライズ・アルゴリズムを応用します。

4.4.2 節では線分の傾きが 1（角度で言えば 45°）よりも小さい場合にはループを X で回し、大きい場合には Y で回しました。しかし、ここでは各 Y について水平方向に塗りつぶしを実行するため、ループは必ず Y で回すことに固定します。そうすると、4.4.2 節で述べたように、頂点から底辺の端点へ至る線分の傾きが 1 よりも小さいときにはラスタライズされた画素がつながらない現象が起きてしまいます。しかし、これは面のラスタライズでは問題になりません。たとえば図 4.71 では右辺は傾きが 1 よりも小さいため、ループを Y で回してラスタライズすると、図中の赤色の画素点が抜け落ちて、線分としてはつながりません。しかし、その赤い画素点は三角形の内部

[29] b は bottom、t は top の頭文字です。s は start、e は end の頭文字です。
[30] Y 座標値は図の下方へ増えることに注意してください。

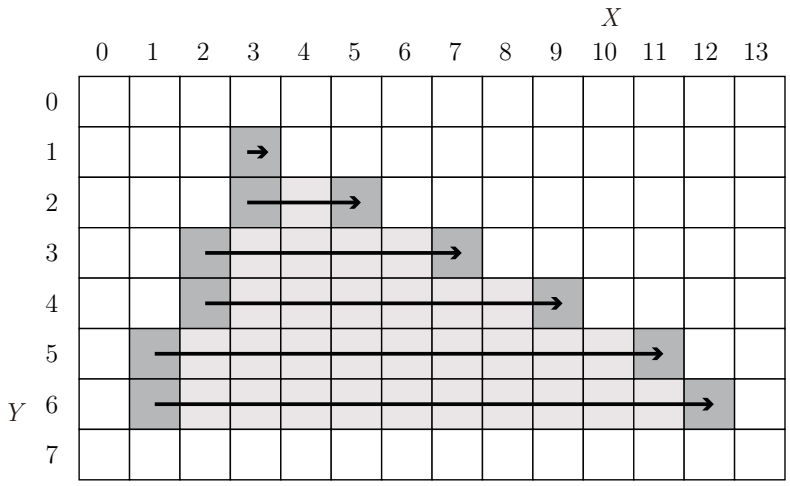


図 4.70: 三角形を水平方向へ走査

点として色付けされるため、三角形のラスタライズとしては問題が生じることはありません。

煩雑なので、三角形をラスタライズする Processing のプログラムは省略します。

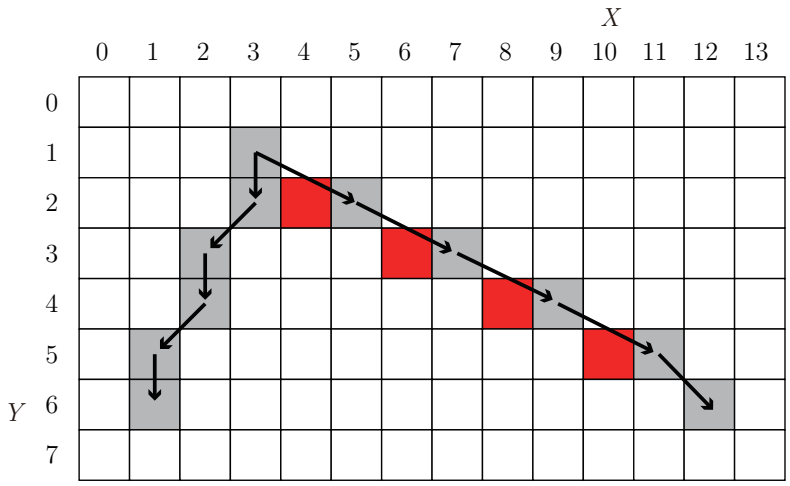


図 4.71: 三角形の左辺と右辺のラスタライズ（図中の赤い画素点が抜けますが、問題を生じません）

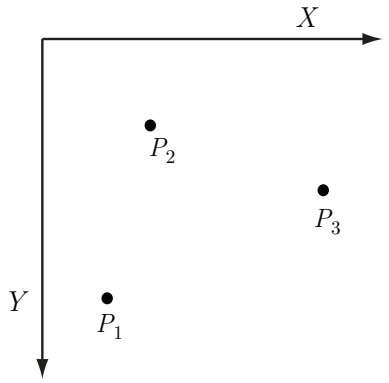


図 4.72: 点列の例

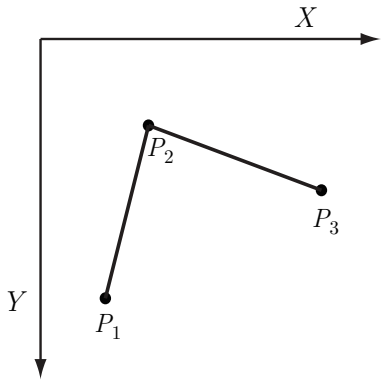


図 4.73: 線形補間（1 次補間）の例

4.5 滑らかな曲線

ここまでは折れ線（始点と終点が連続する複数の線分）を描画してきました。しかし、たとえば動植物など自然界に存在する被写体を CG で描く場合、折れ線では不都合で、曲線が必要となる場合が多々あります。短い線分を多数用いれば擬似的に曲線を描くこともできますが、その場合、線分のためのデータ量が膨大になってしまいます。これを解決するために、限られた数の点を用いて滑らかな曲線を定義する技術が昔から研究されています。そのような曲線を特に**スプライン曲線**（spline curve）[31]と呼んでいます。

4.5.1 問題の定義

スプライン曲線の技術は関数の多項式補間の技術と見なすことができます。まず、その観点から問題を整理します。

今、X-Y 平面上に座標点 $P_1 = (X_1, Y_1)$、$P_2 = (X_2, Y_2)$、...、$P_N = (X_N, Y_N)$ が N 個与えられているとします。たとえば図 4.72 のような点列です（この場合、$N = 3$）。しかし、この点列のままでは線画になりませんから、点列の間を直線で結びます（図 4.73 参照）。これを線形補間（または 1 次補間）と呼びます。線形補間では描像が滑らかではありません。CG の観点からいえば、そのままでは描画された形状が不自然に角張って見えてしまうこととなり、不都合です。

そこで、多少、数学的な厳密性を欠いたとしても[32]、関数形を人の目で見て滑らかに見えるように強制的に補正することを考えます。これを一般に**スプライン補間**（spline interpolation）と呼んでいます。図 4.74、図 4.75 はそのような例です。スプライン補間は技術的には 2 次式、3 次式など

[31] 「スプライン」はもともと曲線を引くために作られたしなる（撓る、あるいはたわむ）ようにできた薄板を指した言葉です。

[32] 何をもって厳密性とするかを論じていないので、実はこの言い方自体が厳密ではないのですが。

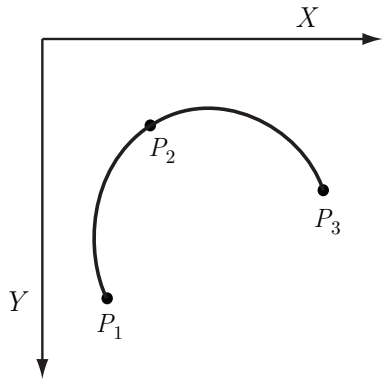


図 4.74: スプライン補間の例（その 1）

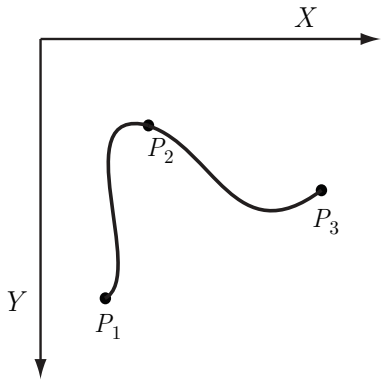


図 4.75: スプライン補間の例（その 2）

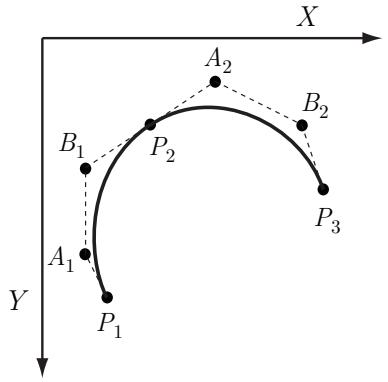


図 4.76: ベジエ曲線の制御点（その 1）

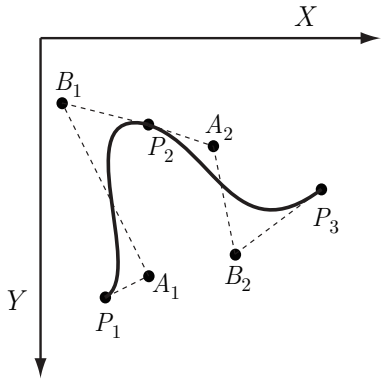


図 4.77: ベジエ曲線の制御点（その 2）

の高次多項式[33]を用いた補間の一種です。多項式補間の方法は多項式の選び方によって無数に存在しますが、CG の分野では、(1) 形状を人間が直感的に取り扱うことができる、(2) 形状が感覚的に自然である[34]、(3) 計算量が少ない等の理由から、以下に述べる方式がよく利用されています。

4.5.2 ベジエ曲線

フランス人技術者ベジエ[35]（Bézier）が考案した多項式近似によって生成される曲線をベジエ曲線と呼んでいます。

ベジエ曲線では、点の間を曲線で結ぶために、いくつかの制御点（線の曲がり具合を制御する点）が必要です。最もよく利用されている 3 次ベジエ曲線の場合には 2 個の端点と 2 個の制御点が必要です。たとえば図 4.76、図 4.77 では点 P_i と点 P_{i+1} の間に制御点 A_i と B_i を与えていま

[33] 数学的には 1 次のスプラインも存在しますが、それは先に述べた 1 次補間に他なりません。
[34] 曲線が不必要に曲がり過ぎていないことが重要です。
[35] しばしば「ベジェ」とも表記されます。

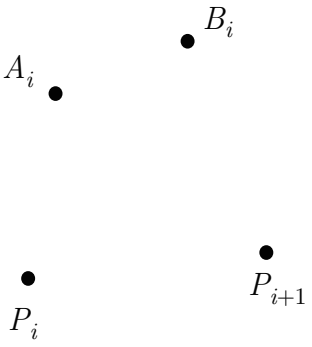
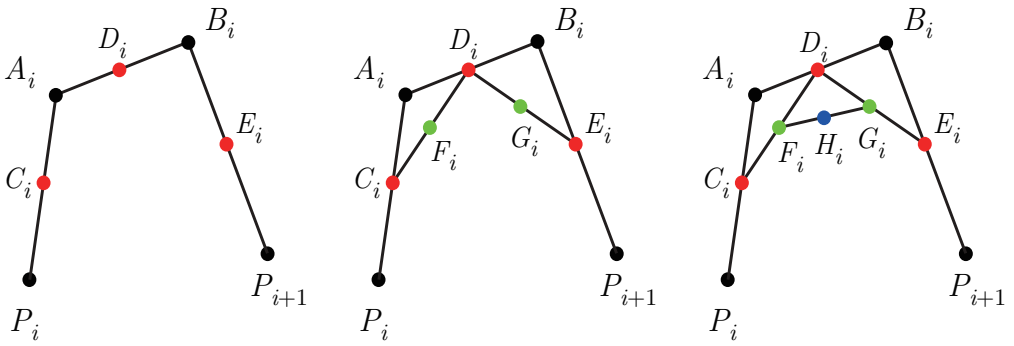


図 4.78: 4 点 P_i、A_i、B_i、P_{i+1} の例



(a) 第 1 ステップ　(b) 第 2 ステップ　(c) 第 3 ステップ

図 4.79: ド・カステリョのアルゴリズムによる 3 次ベジエ曲線の計算例（中点 $t = 0.5$ の場合

す[36]。なお、生成されるベジエ曲線は必ず P_i と P_{i+1} を通りますが、制御点 A_i と B_i の上を通ることは保証されません。

ド・カステリョのアルゴリズム

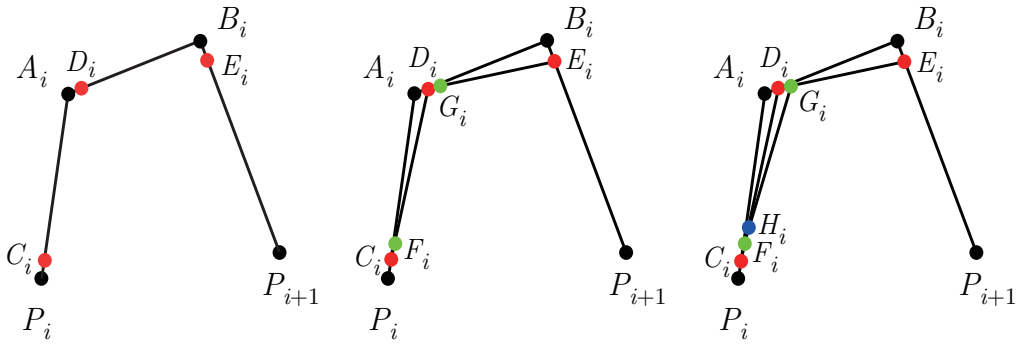
四つの点 P_i、A_i、B_i、P_{i+1} から生成されるベジエ曲線の内容は、フランス人技術者ド・カステリョ（de Casteljau）[37]が提案したアルゴリズムによる説明が直感的です。四つの点 P_i、A_i、B_i、P_{i+1} から具体的にベジエ曲線を求める方法はド・カステリョのアルゴリズムと呼ばれており、以下に述べる再帰的な計算方法で理解するのが直感的です。

まず、図 4.78 のように 4 点が与えられたと仮定します。そうすると、求める曲線は P_i を始点とし、P_{i+1} を終点とするものになります。

まずその曲線の中点を求めてみます。そのために、

[36] 一般に 3 次方程式は $a_3x^3 + a_2x^2 + a_1x + a_1x$ という式で与えられ、4 個の係数 a_3、...、a_0 を持ちます。よってそれら 4 個の係数を一意に決定するためには 4 個の点が必要となるのです。

[37] ベジエもカステリョも共に自動車会社在職中にこの曲線を提案しており、自動車の形状などの工業デザインへ応用することを意図していたのでしょう。



(a) 第 1 ステップ　　(b) 第 2 ステップ　　(c) 第 3 ステップ

図 4.80: ド・カステリョのアルゴリズムによる 3 次ベジエ曲線の計算例（$t = 0.1$ の場合）

第 1 ステップ： 与えられた P_i、A_i、B_i、P_{i+1} を用いて（図 4.79 (a) 参照）

1. 線分 P_iA_i について、その中点 C_i を求めます。
2. 線分 A_iB_i について、その中点 D_i を求めます。
3. 線分 B_iP_{i+1} について、その中点 E_i を求めます。

第 2 ステップ： 第 1 ステップで求めた C_i、D_i、E_i を用いて（図 4.79 (b) 参照）

1. 線分 C_iD_i について、その中点 F_i を求めます。
2. 線分 D_iE_i について、その中点 G_i を求めます。

第 3 ステップ： 第 2 ステップで求めた F_i、G_i を用いて（図 4.79 (c) 参照）

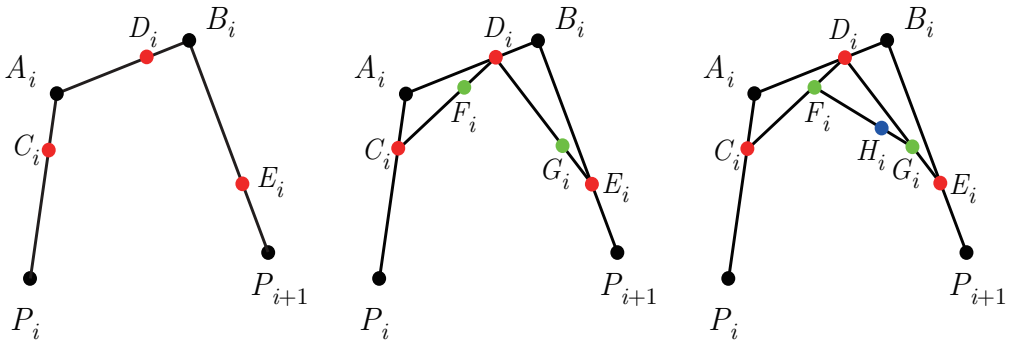
1. 線分 F_iG_i について、その中点 H_i を求めます。

上の手順で求めた点 H_i を曲線の中点とするのです。

中点でない場合も同様です。曲線の上に 媒介変数 t を設定し、$t = 0$ を始点 P_i、$t = 1.0$ を終点 P_{i+1} に対応させておきます。$t = 0.5$ は上に求めた中点 H_i です。そして、たとえば $t = 0.1$ の場合の点を求めるには、上記手順で線分の中点を求めたところを、線分を始点から終点方向へ 10% だけ進行した点を求めるように変更するだけです（図 4.80 参照）。また、$t = 0.7$ の場合には線分を始点から終点方向へ 70% だけ進行した点を求めるように変更します（図 4.81 参照）。このようにして $0 \leq t \leq 1$ の範囲で点を求め、それらをつないだ軌跡（図 4.82）をベジエ曲線とします。

級数による計算

ド・カステリョのアルゴリズムの図的理解はベジエ曲線の本質を理解する上で直感的で分かりやすいものです。しかし、実際の計算を式として厳密に定義しておかねばなりません。



(a) 第 1 ステップ　(b) 第 2 ステップ　(c) 第 3 ステップ

図 4.81: ド・カステリョのアルゴリズムによる 3 次ベジエ曲線の計算例（$t = 0.7$ の場合）

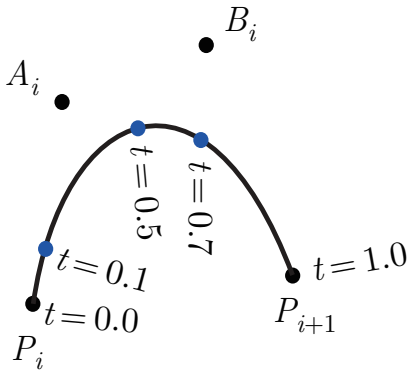


図 4.82: ド・カステリョのアルゴリズムによる 3 次ベジエ曲線の軌跡

今、2 点 P と P' が線分の始点と終点として与えられたとき、始点から終点へ向かって $(100 \cdot t)$ % だけ進行した点 Q の位置は以下の式で与えることができます。

$$\begin{aligned} Q &= (P' - P) \cdot t + P \\ &= (1 - t) \cdot P + t \cdot P' \end{aligned}$$

この式をド・カステリョのアルゴリズムに適用し、図 4.79〜図 4.81 の C_i、...、H_i を数式で表すと以下の通りです。

$$\begin{aligned} C_i &= (1 - t) \cdot P_i + t \cdot A_i \\ D_i &= (1 - t) \cdot A_i + t \cdot B_i \\ E_i &= (1 - t) \cdot B_i + t \cdot P_{i+1} \\ F_i &= (1 - t) \cdot C_i + t \cdot D_i \\ G_i &= (1 - t) \cdot D_i + t \cdot E_i \\ H_i &= (1 - t) \cdot F_i + t \cdot G_i \end{aligned}$$

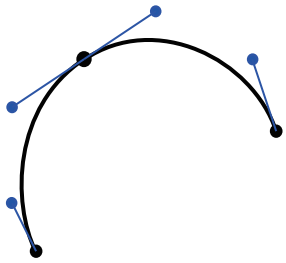


図 4.83: Adobe Illustrator におけるベジエ曲線の編集の様子（その 1）

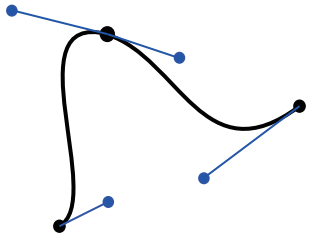


図 4.84: Adobe Illustrator におけるベジエ曲線の編集の様子（その 2）

これを整理すると

$$H_i = (1-t)^3 \cdot P_i + 3(1-t)^2 \cdot t \cdot A_i + 3(1-t) \cdot t^2 \cdot B_i + t^3 \cdot P_{i+1}$$

となり、t に関する 3 次式になっていることが分かります。ベジエ曲線の実際の描画において上の 3 次式を用いるべきです。なお、上の式の係数 1、3、3、1 が 2 項係数（binomial coefficient）であることに気づくでしょう。一般に、$N+1$ 個の点列 $P_{i,0}$、$P_{i,1}$、...、$P_{i,N}$ が与えられたとき（ここまでの 3 次曲線の議論では $N=3$ として $P_{i,0}=P_i$、$P_{i,1}=A_i$、$P_{i,2}=B_i$、$P_{i,3}=P_{i+1}$）、N 次のベジエ曲線は以下の式で与えられます。

$$Q_i(t) = \sum_{k=0}^{N} {}_N C_k \cdot (1-t)^k \cdot t^{N-k} \cdot P_{i,k}$$

ここに ${}_N C_k$ は 2 項係数の一般形です。

ドローイングツールにおけるベジエ曲線

ドローイング・ツールで曲線を取り扱うときにベジエ曲線はとても便利です。たとえば図 4.83、図 4.84 は、Adobe Illustrator でそれぞれ 2 本のベジエ曲線を編集しているときのスナップショットです。両図の 3 端点（図中の黒い点）は幾何学的に同じ配置にしていますが、制御点（図中の青い点）の位置をマウスで対話的に調整することで様々な曲線に変更することができます。

p5.js でのベジエ曲線の描画

p5.js には四つの座標点を引数として与えてベジエ曲線を描画する関数が用意されています。

`bezier(x1, y1, x2, y2, x3, y3, x4, y4)` ... 2 次元 X-Y 座標上の 4 点 $P_i = (\texttt{x1}, \texttt{y1})$、$A_i = (\texttt{x2}, \texttt{y2})$、$B_i = (\texttt{x3}, \texttt{y3})$、$P_{i+1} = (\texttt{x4}, \texttt{y4})$ で定義されるベジエ曲線を描画す、かつその曲線の囲む領域内に色を塗ります。曲線の設定を `stroke()`、`strokeWeight()` などで行い、領域の色の設定を `fill()` で行うことはこれまでと同様です。

```
function setup() {
  createCanvas(300, 300);
  background(200);
  fill(255, 0, 0);
  strokeWeight(10);
  bezier(100, 100,
           0, 300,
         300, 300,
         200, 100);
}
```

図 4.85: 簡単なベジエ曲線の描画プログラム：

図 4.86: 左のプログラムの実行結果

図 4.85 のプログラムは、$P_i = (100, 100)$、$A_i = (0, 300)$、$B_i = (300, 300)$、$P_{i+1} = (200, 100)$ の 4 点で膨らみのある曲線を描いたものです。曲線の色はデフォルトの黒、線幅は 10 画素、領域内は赤で描いています。

図 4.87 のプログラムは、$P_i = (100, 100)$、$B_i = (200, 200)$...、$P_{i+1} = (100, 200)$ に固定し、A_i をマウスの位置とするときのベジエ曲線を逐次、描画します。4 点の位置には `ellipse()` を用いてそれぞれ小円を描画します。

4.5.3 様々なスプライン曲線

ベジエ曲線は **B スプライン**（B spline）曲線と呼ばれるより広い概念の特殊形であることが知られています。B スプラインにはさらにその上位概念である **NURBS**（非一様有理 B スプライン、Non Uniform Rational B-Spline）曲線が存在します。現在、コンピュータを用いた形状のモデリングにはこの NURBS が広く利用されています。

```
function setup(){
  createCanvas(300, 300);
  noFill();
}

function draw(){
  background(200);

  ellipse(100, 100, 10, 10);
  ellipse(mouseX, mouseY, 10, 10);
  ellipse(200, 200, 10, 10);
  ellipse(100, 200, 10, 10);

  bezier(100,    100,
         mouseX, mouseY,
         200,    200,
         100,    200);
}
```

図 4.87: マウスで制御点のひとつを動かす場合のベジエ曲線の描画プログラム

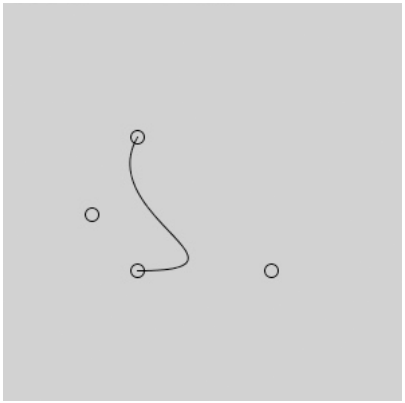


図 4.88: 左のプログラムの実行結果

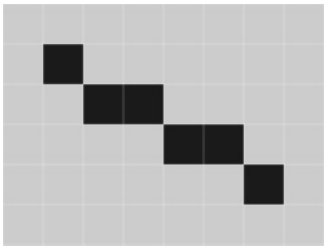


図 4.89: 図 4.54 のプログラムの実行結果

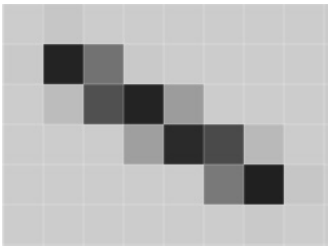


図 4.90: `line(1,1,6,4)` の実行結果

4.6　アンチエイリアシング

図 4.89 は、線分をライスタライズするプログラム（図 4.54）の実行結果です。図 4.52 (f) の通りの描画です。これに対して、図 4.90 は p5.js の関数 `line()` を用いて同じ線分を描画したものです。

遠目から見れば、図 4.90 の描画の方が線分として滑らかに見えます。

図 4.89 の描画の方では階段状のギザギザ — これを**ジャギー**（Jaggy）と呼びます — が目立ちます。これは、本来、無限の精度で定義されているベクタ・データを有限の大きさを持つ画素の上にラスタライズしたことによる、一種の雑音であって、原理的にはジャギーは避けられません。

しかし、画像生成を工夫することでジャギーを目立たなくさせることは可能です。この処理を一般に**アンチエイリアシング**（anti-aliasing）と呼んでいます[38]。これは高品質の CG を作成する上で重要な技術です。図 4.90 の p5.js の生成する線分にはアンチエイリアシングが施されている訳です。

最も単純なアンチエイリアシングの手法は、表示装置の解像度よりも高い解像度でラスタライズを行い、装置への表示では隣接する複数の画素の情報を平均化して表示する方法です。これは**スーパーサンプリング**（super sampling）と呼ばれます。この方法によって CG 画像上の規則的な高周波の雑音をかなり除去することができます。この方法の欠点は、仮想的とはいえ、解像度を上げて画像を生成するため、計算量が大きくなる点です。たとえば解像度を縦横 2 倍に想定すれば、演算時間は 4 倍掛かってしまいます。描画コストと画質は多くの場合、トレードオフの関係にある訳です。

p5.js では、2 次元グラフィックスではデフォルトでアンチエイリアシングを行い、3 次元グラフィックスではデフォルトではアンチエイリアシングを行わない仕様になっています。

考察　p5.js のアンチエイリアシングを制御する関数 `smooth()`、`noSmooth()` について調べなさい。

[38] alias には「雑音」という意味があります。

第VII部

第5章　3次元グラフィックス：線画

この章から3次元コンピュータグラフィックス（3-Dimensional Computer Graphics、以下、3D CG）に入ります。

5.1　3D CGとは

3D CGとは、コンピュータ内に仮想的に構築された3次元空間に仮想的な被写体を配置し、その被写体を仮想的なカメラで撮影して得られる仮想的な写真画像をコンピュータを用いた数値計算で求める画像生成技術です。

それを達成するには様々な技術を組み合わせねばなりませんが、この章の目的に限って言えば、たとえば立方体を構成する8個の頂点と12本の線分が3次元空間内に与えられたときに、図5.1のような線画を描くことです。このような線画を**ワイヤーフレーム**（wireframe、輪郭線）画像と呼びます。この描画を実現するには**3次元空間内の点** (x, y, z) **を2次平面上の点** (X, Y) **へ対応させる**ことが出発点となります。本章もそこから話を始めていきます。

ところで、もし立方体が単に骨組みだけではなく、中身の詰まった立体であるならば、立方体の裏側の線も描かれている図5.1は不自然です。そのような立方体は図5.2のように描画されるべきです。物体の裏側の、本来見えないはずの線を描かないことを**陰線消去**（hidden line removal）と呼びますが、これを実現する技術はそれほど単純ではありません。立方体程度のものならば比較的簡単なアルゴリズムで解決しますが、トーラス体（ドーナツの形状、図5.3参照）をワイヤフレームで表す場合、どのような種類の線を残し、どの線を消すべきか、そう簡単な問題ではありません。

線（ベクタ）の消去は、ベクタ・グラフィックスが主流であった時代には重要な技術でしたが、現在のラスタ・グラフィックスの3D CG技術では重要ではありません。というのも、ラスタ・グラフィックスでは物体を線で描画するのではなく画素の色付けで描画するため、線分の見え隠れではなく、個々の画素の見え隠れを検討すればよいからです。これを**隠面消去**（hidden surface removal）と呼びます。これについては次章で解説します。

この章では図5.1のように、物体の輪郭を単純に線で描画する方法について基礎的な事項、特に3次元座標点を次々と座標変換していく仕組みについて詳しく述べていきます。

p5.jsは3D CGをサポートしており、比較的容易に3D CG画像を作ることができます。このテキストではそれを利用しながら、随所にp5.jsのプログラムを紹介していきます。

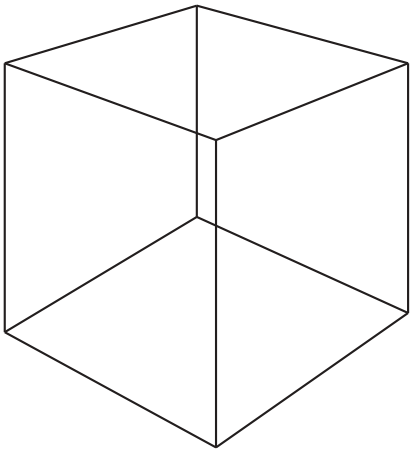


図 5.1: 立方体の 3 次元線画（図 3.1 の再掲、陰線消去なし）

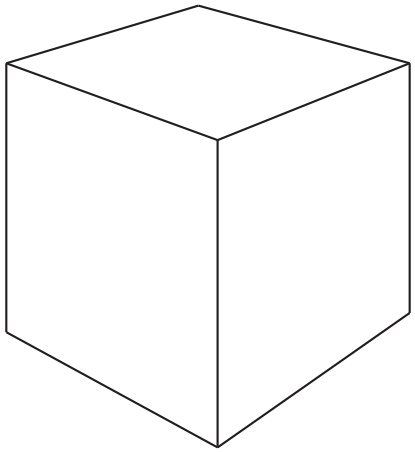


図 5.2: 立方体の 3 次元線画（陰線消去あり）

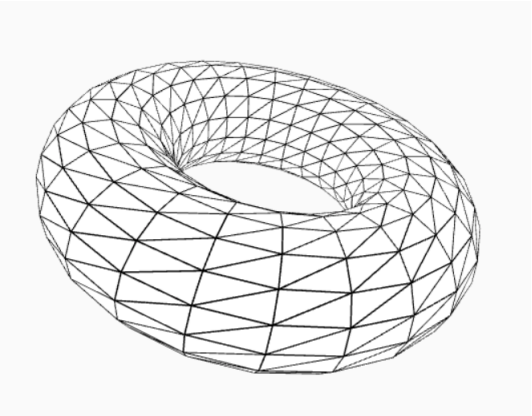


図 5.3: トーラスの 3 次元線画（陰線消去あり）

5.2　3D CG の基本モデル

上に「... 仮想的なカメラで撮影して...」と述べました。3D CG ではそのカメラの原理として**ピンホールカメラ**（pinhole camera）を用います。図 5.4 は、被写体から発射された光線がピンホール（針穴）を通過し、カメラの投影面に映る様子をイラスト化したものです。ピンホールカメラの投影像は倒立しているのですが、しかし CG でわざわざ倒立した像を扱う必要はありません。そこで、図 5.5 のように、ピンホールについて点対称な位置に**投影面**（projection plane）を仮想的に作り、これを 3D CG の基本モデルに採用するのが一般的です。このとき、ピンホールに相当する位置を新たに**視点**（viewpoint）と呼びます。そしてこのモデルでは**被写体から発射されて視点に入射する光線が投影面と交差する位置にその被写体の像を結ぶ**ことを考えます。

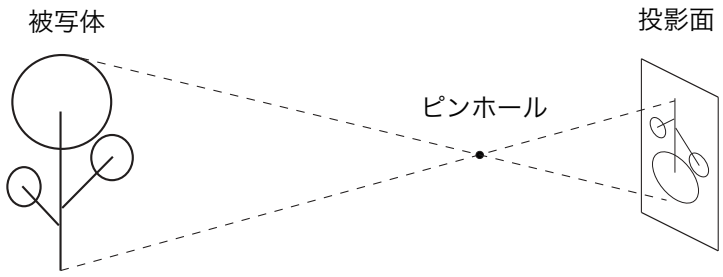


図 5.4: ピンホールカメラの構造

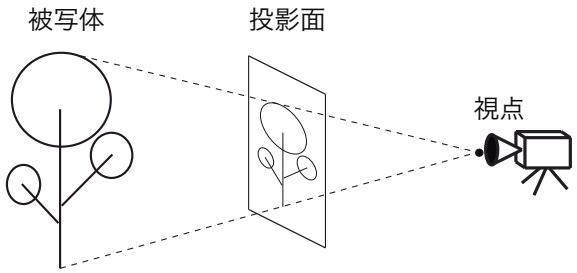


図 5.5: ピンホールカメラを簡略化した CG モデル

その様子をより精密化するために、図 5.5 に座標軸を追加したものが図 5.6 です。同図を別の方向から表したものが図 5.7、図 5.8 です。幾何学関係を 3 次元的に理解してください。

以下はこれらの図の説明であり、この章以降における基本設定です。

1. 3D CG の仮想世界は 3 次元の x-y-z 座標系で表すと仮定します。この座標系は、私たちが通常用いている右手座標系ではなく、**左手座標系**（left-handed coordinate system）です。
   (a) x 軸の正の方向は視点から見て右方向です。
   (b) y 軸の正の方向は視点から見て下向きです（視点から見た x-y 軸は前章と同じ見え方です）。
   (c) z 軸の正の方向は投影面から視点へ向かう方向です。逆に言えば、z 軸の負の方向が視点前方方向です。

   3D CG で右手座標系を用いるか、左手座標系を用いるかは CG の教科書、CG ライブラリ毎に様々です。ちょうど前章で Y 軸が上向きなのか、下向きなのかを議論したことと同じ状況です。それぞれ長所短所があって優劣付けがたいのが現状です。非常に悩ましいところです。左手座標系は私たちが慣れ親しんでいる座標系とは異なるため扱いにくいのですが、p5.js では左手座標系を用いているため、ここでもそれに従うこととします。

2. x-y-z 座標系内の長さ（距離）は特定の単位（メートル、キロメートル）を想定していません。

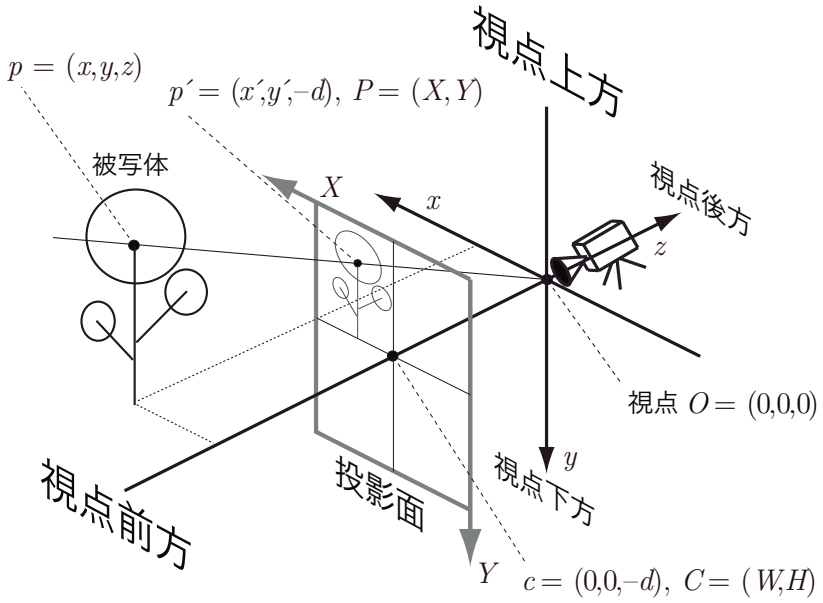


図 5.6: 投影変換と座標系の関係

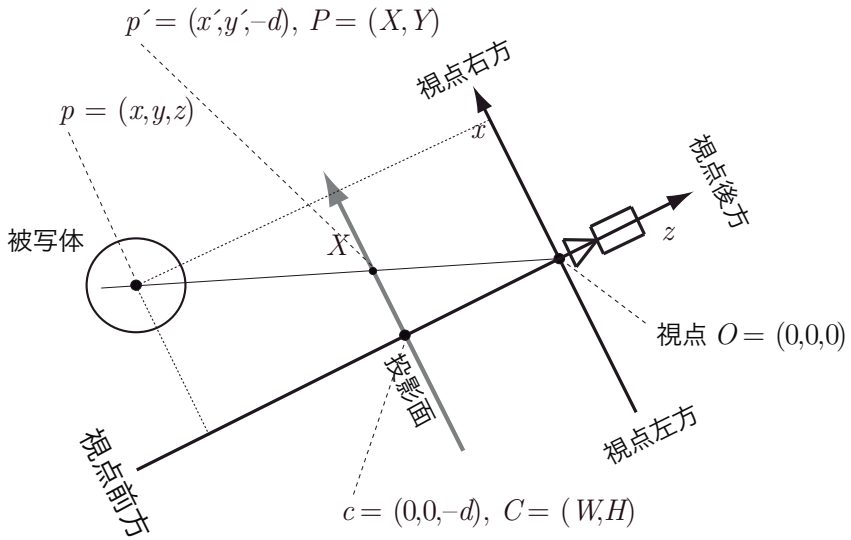


図 5.7: 投影変換と座標系の関係：図 5.6 を真上（y 軸の正の方向）から見た場合

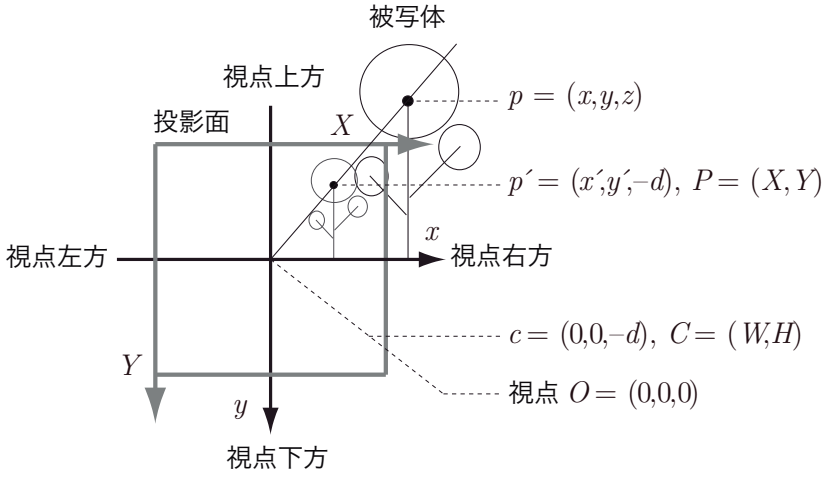


図 5.8: 投影変換と座標系の関係：図 5.6 を奥（z 軸の負の方向）から見た場合

どのような単位を用いても最終的には被写体が描画されたキャンバス上の画素間の距離に還元されるからです。x-y-z 座標系の単位長 1 を 1mm と見なしても、1km と見なしても構いません。

3. 視点は x-y-z 座標系の原点に設定します。

   視点を原点以外に設定することも可能です。実は p5.js のデフォルトの設定では視点は原点ではありません。しかし、簡単な座標変換で視点を原点に移動できるため、ここでは原点を用いることとします。

4. 正定数 d を導入し、投影面は x-y-z 座標系内の $z = -d$ の x-y 平面上とあると仮定します。

5. 投影面は有限の大きさを持つ矩形であり、その中心を x-y-z 座標系の z 軸上に設定します。具体的には、正定数 w、h を導入し、その矩形の位置を $-w \leq x \leq w$、$-h \leq y \leq h$、$z = -d$ と設定します[1]。よって、投影面の中心 c の座標値は $(0, 0, -d)$ です。

6. 投影面に映った映像がそのままキャンバス（フレームバッファ）の上の映像とみなします。

7. キャンバスは前章までと同様に X-Y 座標系で表します。X 軸と x 座標系は平行になるように設定します。Y 軸と y 座標系も平行に設定します。

8. 正定数値の W、H を導入し、キャンバスの水平方向の画素数は $2W$、垂直方向は $2H$ の大きさとします。キャンバスの縦横の比率 $H : W$ — これを**アスペクト比**（aspect ratio）と呼びます — は、投影面の縦横の比率 $h : w$ とは同じであると仮定します。キャンバスの中心 C は (W, H) であり、投影面の中心 c と一致します。

[1] w は width 、h は height の頭文字から取りました。

3D CG を行なうには、まず第一に、3 次元空間内に置かれた被写体の上の任意の頂点 $p = (x, y, z)$ が投影面へ映し出される位置 $p' = (x', y', -d)$ を知らねばなりません（図 5.6 等参照）。この計算を一般に**投影変換**（projection transformation）と呼びます。その方法には代表的な 2 通りの方法が知られており、すぐ後に詳細に解説します。

投影変換で投影面上の位置 $p' = (x', y', -d)$ が求められたならば、それに対応するキャンバス上の画素位置 $P = (X, Y)$ を次式で計算します。

$$X = \frac{W}{w} \cdot x' + W, \tag{5.1}$$

$$Y = \frac{H}{h} \cdot y' + H \tag{5.2}$$

この変換式では、$p' = (-w, -h, -d)$ の位置がキャンバスの左上点 $P = (0, 0)$ に対応し、$p' = (0, 0, -d)$ がキャンバスの中心点 $P = (H, W)$、$p' = (w, h, -d)$ が右下の点 $P = (2W, 2H)$ に対応します。

キャンバス内の画素位置 $P = (X, Y)$ が求められたならば、その位置に被写体をラスタライズします。この章では、被写体を構成する頂点と頂点を線分で結ぶワイヤフレームをラスタライズします。

5.3　投影変換

投影変換では、**透視投影**（perspective projection）とその特殊形である**平行投影**（parallel projection）が代表的です。

平行投影は人が手書きで立体図を描く場合などに無意識に用いている方法でもあるため、平行投影を先に解説する教科書、資料も多いのですが、しかし本格的な 3D CG を行なうには結局、透視投影を避けて通れませんから、ここでは透視投影を先に解説し、その特殊形（極限形）として平行投影を定義します。

5.3.1　透視投影変換

透視投影は、前述のピンホールカメラを忠実にモデル化した投影手法です[2]。3 次元世界を人の目で見るときの投影法であり、**遠近法**（perspective drawing）を実現する方法と説明されることもあります。

図 5.6（と図 5.7、図 5.8）を思い出しましょう。この図における幾何学的関係として以下が成り立ちます。

[2] 通常のカメラとピンホールカメラの違いのひとつは、通常のカメラには有限の被写界深度がありますが、ピンホールカメラにはそれがないことです。被写界深度とは被写体にピントが合う適正距離のことです。これを逆に利用し、わざとピントの合わないボケ（bokeh）を作り出すことができます。ピンホールカメラの被写界深度は無限に深いため、全ての被写体にピントが合ったような画像しか作ることができません。そのためボケを CG で再現するには特殊な CG アルゴリズムを付加する必要があります。

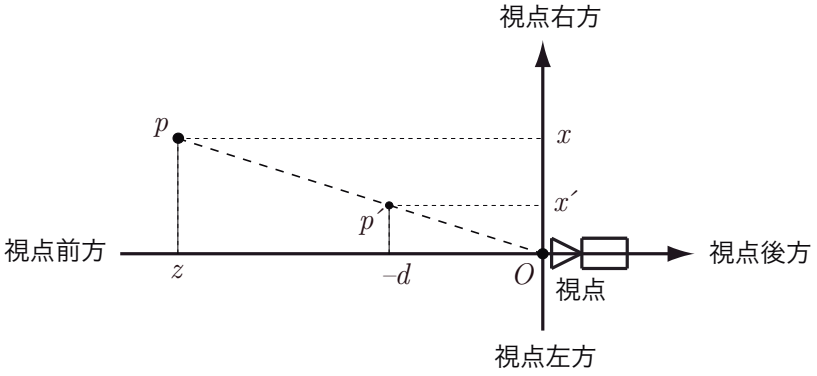


図 5.9: 透視投影における x、x'、z、d の幾何学的関係

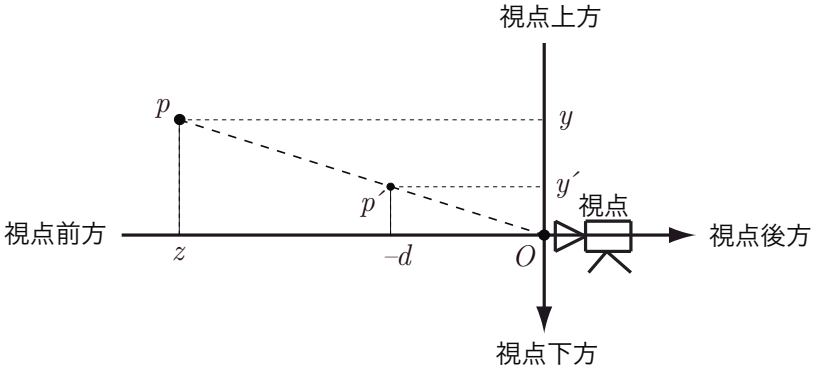


図 5.10: 透視投影における y、y'、z、d の幾何学的関係

- 3 次元空間内の被写体の上の点 $p = (x, y, z)$、その点を投影面に映した位置 $p' = (x', y', -d)$、そして視点 $(0, 0, 0)$ の 3 点は必ず一直線上に並ぶ。

このことから、x' y' を x、y、z、d の式として表すことが可能です。

図 5.9 は、図 5.6 の関係を真上（y 軸の負の位置）から見直した図であり、図 5.7 と同等の図です。y 軸は、紙面と垂直になるため省略されています。この図に含まれる二つの直角三角形の相似性から以下が成り立ちます。

$$\frac{x}{z} = \frac{x'}{-d}$$

図 5.6 の関係を左横（x 軸の負の位置）から見直した図が図 5.10 です。x 軸は、紙面と垂直になるため省略されています。この図から、x の場合と同様にして

$$\frac{y}{z} = \frac{y'}{-d}$$

が導出されます。これらを変形し、まとめると以下の通りです。この 2 式が透視投影変換の基本式

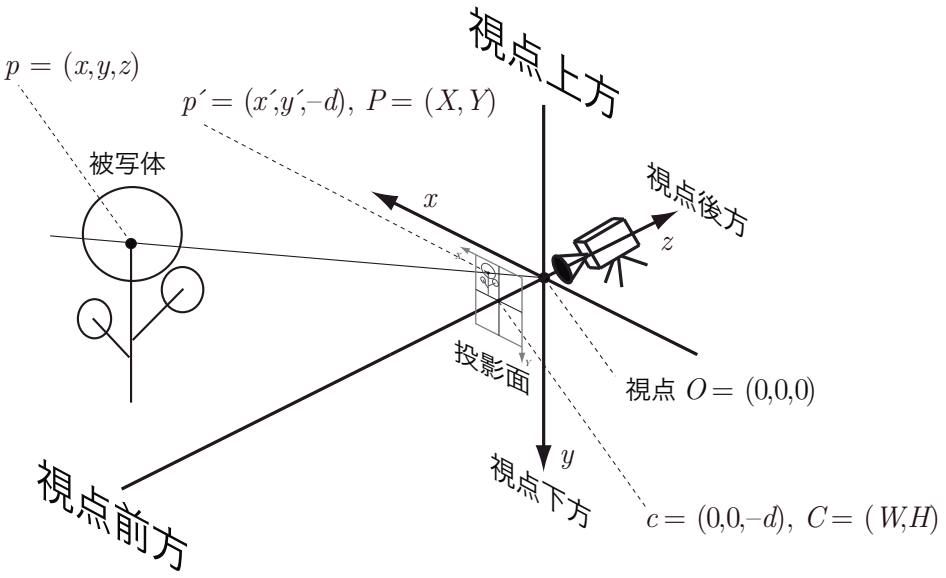


図 5.11: 投影変換と座標系の関係（d が小さくても図 5.6 と同じ 3D CG 画像を作ることができる）

です。

$$x' = -d \cdot \frac{x}{z}, \tag{5.3}$$

$$y' = -d \cdot \frac{y}{z} \tag{5.4}$$

これら 2 式は x、y の正負に関係なく成り立つ式です。

ここで図 5.6 を見直しましょう。この図からは、被写体頂点は視点から見て投影面の奥に位置することが自然に思えます。つまり、$z \leq -d$ であるべきと思えます。しかし実際には $-d < z < 0$ であっても画像描画に問題は生じません。もし視点の間近に被写体を置きたいならば、d を十分小さく設定し、小さな投影面をその被写体の前に置けばよいのです。

これを数式で知るために式 (5.1)、式 (5.2) と 式 (5.3)、式 (5.4) をまとめると、以下の式を得ます。

$$X = -W \cdot \frac{d}{w} \cdot \frac{x}{z} + W, \tag{5.5}$$

$$Y = -H \cdot \frac{d}{h} \cdot \frac{y}{z} + H \tag{5.6}$$

上の式は仮に d が変更されても d/w、d/h が不変であれば投影位置 (X,Y) は不変であり、よって CG 画像も不変であることを意味します。画像生成に d の値それ自体は重要ではなく、必要ならば d はどこまでも小さくできるのです。重要なのは d と w、d と h との比 d/w、d/h です[3]。参考のため、図 5.11 に d を小さくした場合の投影変換の幾何関係を図示します。投影面上の画像が図

[3] この比は画像の**画角**（view angle）と関連しているのですが、それについて後に述べます。

5.11 と図 5.6 で同じになることが分かるでしょう。必要ならば、d/w、d/h を一定のまま、d、w、h の値を 0 に限りなく近づけることができます。

しかし $0 \leq z$ は不自然です。これを認めると、視点（カメラ）の背後にある被写体が映ることになってしまいます。そのような不自然な被写体の描画を抑制する処理を**クリッピング**（clipping）と呼びます。また、これとは別に、キャンバスからはみ出した位置に描画される（よってキャンバス内には決して描画されない）被写体の描画を抑制し、描画処理の効率を上げることもクリッピングと呼ばれます。ただしここでは当面はクリッピングは検討しません。

さて、5.2 節において、

- x-y-z 座標系内の長さ（距離）は特定の単位（メートル、キロメートル）を想定していません。どのような単位を用いても最終的にキャンバス上の画素間の距離に還元されるからです。x-y-z 座標系の 1 の長さを 1mm と考えても、1km と考えても構いません。

と述べたことを思い出してください。式 (5.6) では x-y-z 座標系に関係する値 d、w、h、x、y、z は全て d/w、d/h、x/z、y/z という比率の計算の中で用いられています。つまり、x-y-z 座標系の長さの単位に何を用いてもよいことが式 (5.6) から分かるのです。

さらに式 (5.6) について次の性質が成り立ちます。

- x-y-z 座標系の中の任意の線分は透視投影変換（上の式 (5.3)、式 (5.4)）によって X-Y 平面（キャンバス）上の線分に投影されるか、あるいはひとつの点に投影される。

考察　上の性質を証明しなさい（x-y-z 座標系上の 2 点 $p_1 = (x_1, y_1, z_1)$、$p_2 = (x_2, y_2, z_2)$ を通過する直線を考え、p_1、p_2 に透視投影変換を適用し、得られた X-Y 平面上の 2 点を P_1、P_2 とします。p_1、p_2 を通過する直線の方程式に透視投影変換を適用すると、P_1、P_2 を通過する直線の方程式が得られることを示せばよいでしょう。ただし、$P_1 = P_2$ となるには線分は 1 点に投影されます。）

この性質から、多数の線分で構成されたワイヤフレーム画像を生成するには、

1. まず、被写体を構成する各頂点を X-Y 平面上の頂点へ透視投影変換し、
2. X-Y 平面上の頂点を線分で結べばよい

ことになります。

5.3.2　p5.js における透視投影変換

実際に 3D CG 画像を作ってみましょう。

p5.js で 3D CG を行うには、以下の関数呼び出しを行う必要があります。

```
function setup() {
  createCanvas(500, 500, WEBGL); //3D CG の指定
  background(200);
  translate(0, 0, 0.866 * height); //この授業での補正

  const zoffset = -500;
  const s = 100;

  line(-s, -s, -s + zoffset, +s, -s, -s + zoffset);
  line(+s, -s, -s + zoffset, +s, +s, -s + zoffset);
  line(+s, +s, -s + zoffset, -s, +s, -s + zoffset);
  line(-s, +s, -s + zoffset, -s, -s, -s + zoffset);

  line(-s, -s, +s + zoffset, +s, -s, +s + zoffset);
  line(+s, -s, +s + zoffset, +s, +s, +s + zoffset);
  line(+s, +s, +s + zoffset, -s, +s, +s + zoffset);
  line(-s, +s, +s + zoffset, -s, -s, +s + zoffset);

  line(-s, -s, -s + zoffset, -s, -s, +s + zoffset);
  line(+s, -s, -s + zoffset, +s, -s, +s + zoffset);
  line(+s, +s, -s + zoffset, +s, +s, +s + zoffset);
  line(-s, +s, -s + zoffset, -s, +s, +s + zoffset);
}
```

図 5.12: 立方体のワイヤーフレーム画像を描画するプログラム

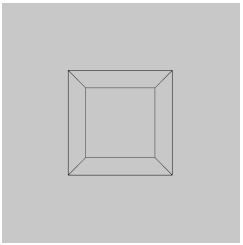


図 5.13: 左のプログラムの実行結果

`createCanvas(width, height, renderer)` ... 第 1、第 2 引数については既に説明した通り、キャンバスの幅、高さを指定します。`width`=`height`=100 がデフォルト値です。新たに第 3 引数 `renderer` にマクロ定数 `WEBGL` を指定することで、p5.js の描画環境が 3D CG 用に設定されます。

3 次元空間に線分を引く関数は、前章で用いてきた `line()` を拡張した以下の関数です。

`line(x1, y1, z1, x2, y2, z2)` ... 3 次元 x-y-z 座標上の点 $p_1 = (\mathtt{x1}, \mathtt{y1}, \mathtt{z1})$ から点 $p_2 = (\mathtt{x2}, \mathtt{y2}, \mathtt{z2})$ へ直線を描きます。具体的には、線分の両端点 p_1、p_2 がキャンバス上の画素位置 P_1、P_2 へ透視投影変換され、P_1 と P_2 を両端点とする線分がキャンバスにラスタライズされます。

立方体の 3D 描画

上の関数を用いて立方体のワイヤーフレーム画像を作成するプログラムが図 5.12 です。なお、説明の都合上、

```
  translate(0, 0, 0.866 * height);
```

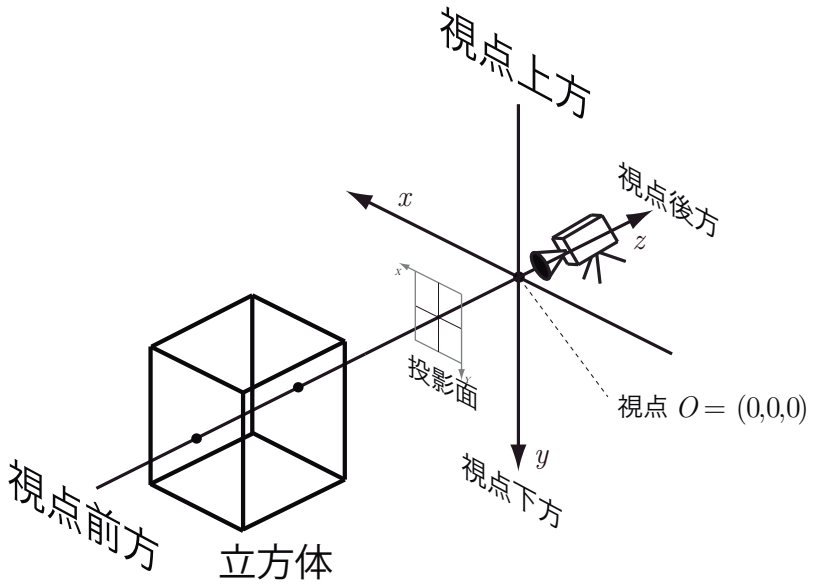


図 5.14: 視点、投影面、被写体（立方体）の位置関係

は視点の位置を補正するための処理を追加しています。というのも、p5.js ではデフォルトでは視点の位置は、このテキストで前提としている座標原点ではなく $(0, 0, \texttt{height}/\left(2\tan\left(\pi/6\right)\right)$ に設定されます。上の関数呼び出しはこれを補正し、視点を原点に移動させる処理です[4]。`translate()` の詳細および視点の位置の自由な設定については後節で述べます。

このプログラムで設定している視点、投影面、被写体（立方体）の位置関係を図 5.14 に示します。ここでは、一辺の長さが 200 の立方体を $(0, 0, -500)$ の周りに仮想的に作り、その 8 個の頂点

$$
\begin{aligned}
(-100, -100, -100) &+ (0, 0, -500),\\
(+100, -100, -100) &+ (0, 0, -500),\\
(+100, +100, -100) &+ (0, 0, -500),\\
(-100, +100, -100) &+ (0, 0, -500),\\
(-100, -100, +100) &+ (0, 0, -500),\\
(+100, -100, +100) &+ (0, 0, -500),\\
(+100, +100, +100) &+ (0, 0, -500),\\
(-100, +100, +100) &+ (0, 0, -500)
\end{aligned}
$$

の間を 12 本の線分で結んだ画像を作ります。図 5.12 のプログラムでは一辺の長さの半分 100 を変数 `s` に設定し、-500 を変数 `zoffset` に設定しています。視点は 座標原点から z 軸の負の方向を見ていますから、$(0, 0, -500)$ の周りの立方体は画像の中心に描画されます。実行結果は図 5.13

[4] $1/2 = 0.5$、$1/\left(2\tan\left(\pi/6\right)\right) \approx 0.866$ であることに注意。

です。視点、投影面、被写体の配置が幾何学的に完全に対照になっているため、図 5.1 のような奥行き感のある立方体としては描画されません。

回転する立方体の 3D 描画

3D 描画された立方体の奥行き感を実感するために、図 5.12 のプログラムをアニメーション化したものが図 5.15 のプログラムです。

まず、初期化関数 `setup()` では `createCanvas(500,500,WEBGL)` を実行しておきます。

描画関数 `draw()` では、初めに `translate(...)` で p5.js の透視投影変換のデフォルト設定を変更します。次にキャンバスを `background(200)` でクリアします。立方体を回転させるために、頂点の座標値を三角関数を用いて計算し、図 5.12 同様に線分を引きます、実行結果は図 5.16 の通りです。

回転しながら上下する星の 3D 描画

もうひとつ例題として、星が上下に動きながら回転する CG アニメーションを示します。元にしたプログラムは 2D CG で星が回転するプログラム（図 4.40）と先出の立方体が回転するプログラム（図 5.15）です。

3D CG を作る際の視点、投影面、被写体（星型）の位置関係を図 5.19 に示します。これは図 5.14 とほとんど同じです。

```
function setup() {
  createCanvas(500, 500, WEBGL);
}

let d = 0;

function draw() {
  translate(0, 0, 0.866 * width);
  background(200);

  const zoffset = -500;
  const s = 100;
  const cs = s * sqrt(2) * cos(d);
  const sn = s * sqrt(2) * sin(d);

  line(-cs, -s, -sn + zoffset,
       +sn, -s, -cs + zoffset);
  line(+sn, -s, -cs + zoffset,
       +sn, +s, -cs + zoffset);
  line(+sn, +s, -cs + zoffset,
       -cs, +s, -sn + zoffset);
  line(-cs, +s, -sn + zoffset,
       -cs, -s, -sn + zoffset);

  line(-sn, -s, +cs + zoffset,
       +cs, -s, +sn + zoffset);
  line(+cs, -s, +sn + zoffset,
       +cs, +s, +sn + zoffset);
  line(+cs, +s, +sn + zoffset,
       -sn, +s, +cs + zoffset);
  line(-sn, +s, +cs + zoffset,
       -sn, -s, +cs + zoffset);

  line(-cs, -s, -sn + zoffset,
       -sn, -s, +cs + zoffset);
  line(+sn, -s, -cs + zoffset,
       +cs, -s, +sn + zoffset);
  line(+sn, +s, -cs + zoffset,
       +cs, +s, +sn + zoffset);
  line(-cs, +s, -sn + zoffset,
       -sn, +s, +cs + zoffset);

  d += 0.01;
}
```

図 5.15: 回転する立方体のワイヤーフレーム画像を描画するプログラム

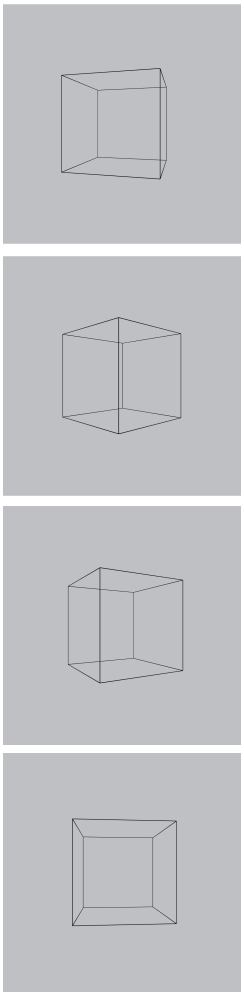


図 5.16: 左のプログラムの実行結果

```
function setup() {
  createCanvas(500, 500, WEBGL);
}

let d = 0;

function draw() {
  translate(0, 0, 0.866 * width);
  background(200);

  const yoffset = d % 400 - 200;
  const zoffset = -500;
  const r = 200.0;
  const n = 23;
  const m = 7;
  const incAngle = 360.0 /n;

  for (let i = 0; i < m * n; i += m) {
    const t1 = radians(incAngle * float(i) + d);
    const x1 = r * cos(t1);
    const z1 = r * sin(t1);

    const t2 = radians(incAngle * float(i + m) + d);
    const x2 = r * cos(t2);
    const z2 = r * sin(t2);

    line(x1, yoffset, z1 + zoffset,
      x2, yoffset, z2 + zoffset);
  }

  d += 1;
}
```

図 5.17: 回転しながら上下する星型を描画するプログラム

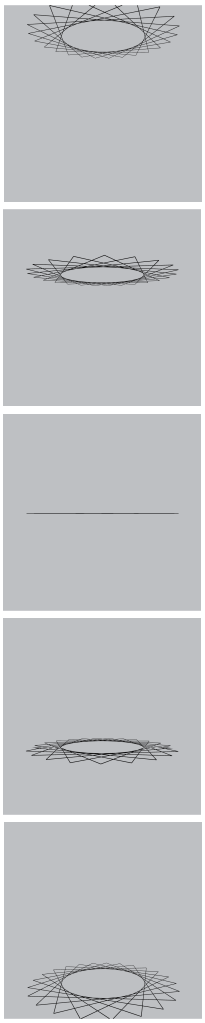


図 5.18: 左のプログラムの実行結果

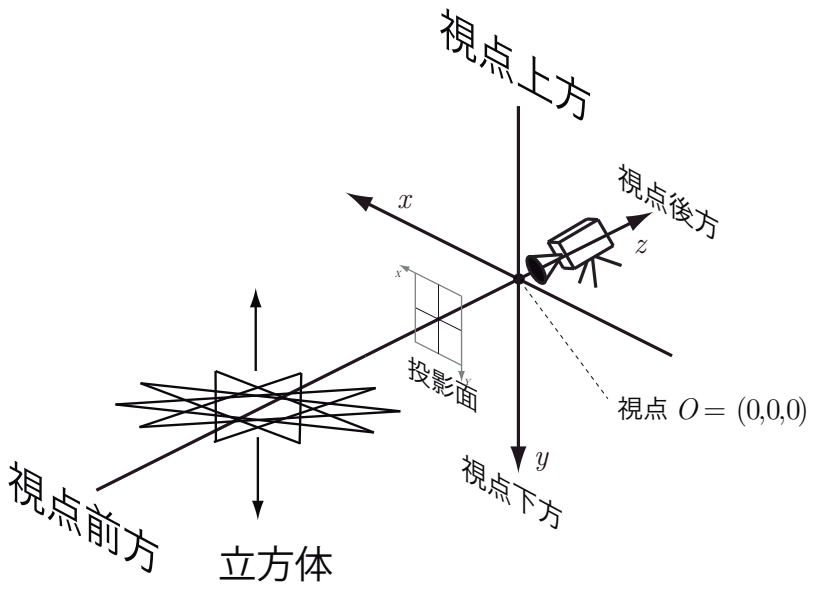


図 5.19: 視点、投影面、被写体（星型）の位置関係

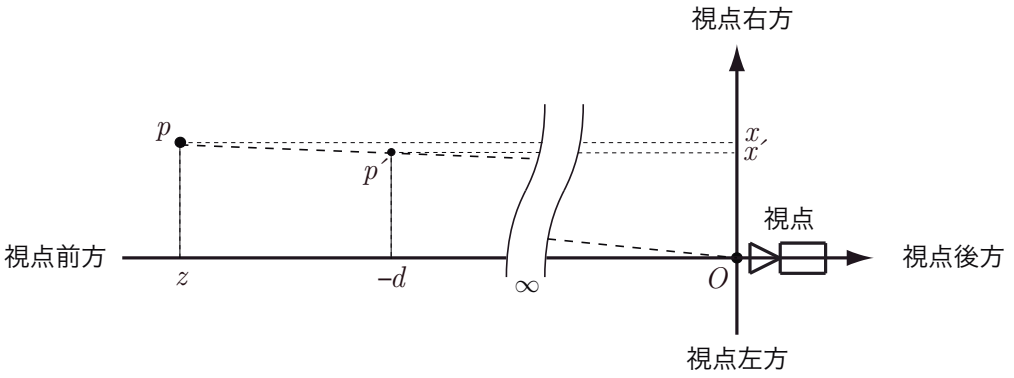


図 5.20: 平行投影における x、x'、z、d の幾何学的関係（d が十分大きな値の場合）

5.3.3　平行投影変換

図 5.9 を再度検討します。この図式において、もし**視点と投影面の間が十分遠い**、すなわち d が十分大きな値であるならば、幾何関係は図 5.20 のように変わります。そして、被写体の点 $p = (x, y, z)$ とそれが投影面へ映る位置 $p' = (x', y', -d)$ について極限において $x' \approx x$ が成り立ちます。変換式について言えば、式 (5.3)、式 (5.4) において $z = -d + z'$ と置き、$d \to \infty$ の場合を求めた以下の式が成り立ちます。

$$x' = \lim_{d\to\infty} -d \cdot \frac{x}{-d + z'} = x, \tag{5.7}$$

$$y' = \lim_{d\to\infty} -d \cdot \frac{y}{-d + z'} = y, \tag{5.8}$$

視点へ入射する光線が z 軸にほとんど平行になっているため、この投影変換を特に**平行投影変換**と呼んでいます。生成される 3D CG 画像は、ちょうど望遠鏡を通して眺めたような、遠近感に乏しいものになるのが特徴です。

変換が単純であることから、手書きで立体図を描くときに多くの人は無意識にこの投影法を利用しているはずです。その意味で重要な投影変換法のひとつですが、リアルな立体感を求める 3D CG には適しません。

5.3.4　p5.js における平行投影変換

p5.js では以下の関数で平行投影変換を実現できます。

`ortho()` [5]... 3D CG の投影方法をデフォルトの透視投影から平行投影に変更します。初期設定で 1 度だけ呼び出します。

[5] 平行投影の英語呼称は parallel projection ですが、別の呼び方では orthographic projectio です。関数名は後者に由来します。

```
function setup() {
  createCanvas(500, 500, WEBGL);
  ortho();                        // 追加
}

let d = 0;

function draw() {
  中略

  const zoffset = -300;       // クリッピングの都合
で変更

  中略
}
```

図 5.21: 回転する立方体のワイヤーフレーム画像を平行投影で描画するプログラム（図 5.15 のプログラムの改造）

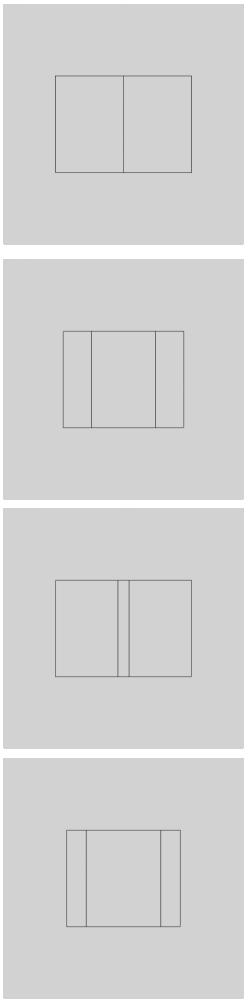


図 5.22: 左のプログラムの実行結果

回転する立方体の 3D 描画

回転する立方体を描画するプログラム（図 5.15）を平行投影に変えるには、図 5.21 のように初期化関数 `setup()` に `ortho()` を追加します。なお、詳細は述べませんが、p5.js では透視投影の場合と平行投影の場合でクリッピングの方法がやや異なるため `zoff` の値を -500 から -300 へ変更しました[6]。実行結果は図 5.22 の通りです。

[6] `zoff=-500` の場合、立方体の辺が描画されない場合があります。その理由は次節のクリッピング空間に関連します。

第VIII部

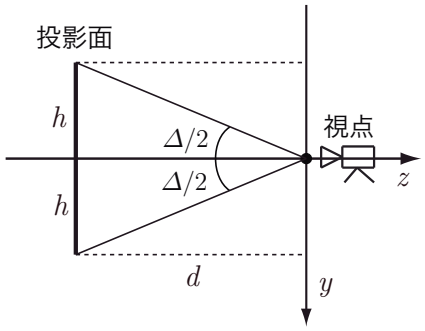


図 5.23: 水平画角 θ と d、h の関係

5.4 画角、キャンバスのアスペクト比、クリッピング空間

ここでは透視投影変換の様々なパラメータを紹介します。

5.4.1 画角

画角（view angle[7]、視野角とも言います）とは、視点から見た投影面の大きさ、広がりを角度で表したものです。投影面の水平方向、垂直方向、対角方向のそれぞれについて**水平画角**（horizontal view angle）、**垂直画角**（vertical view angle）、**対角画角**（diagonal view angle）の 3 種類が考えられます。

図 5.23 には、視点から投影面までの距離 d、投影面の垂直方向の高さ h と垂直画角 Δ_h の関係を図示したものです。この図から直ちに以下の関係が分かります。

$$\frac{h}{d} = \tan\frac{\Delta_h}{2} \tag{5.9}$$

同様に、投影面の水平方向の幅 w と水平画角 Δ_w には以下の関係が成り立ちます。

$$\frac{w}{d} = \tan\frac{\Delta_w}{2} \tag{5.10}$$

ここで以下は先に学んだ透視投影の変換式（式 (5.5)、式 (5.6)）です。

$$X = -W \cdot \frac{d}{w} \cdot \frac{x}{z} + W, \tag{5.11}$$

$$Y = -H \cdot \frac{d}{h} \cdot \frac{y}{z} + H \tag{5.12}$$

この式の d/w、d/h はそれぞれ $1/\tan(\Delta_w/2)$、$1/\tan(\Delta_h/2)$ で置き換えることができます。よっ

[7] または angle of view、viewing angle。

て、投影変換は画角を用いて以下のように定式化できます。

$$\begin{aligned} X &= -\frac{W}{\tan(\Delta_w/2)} \cdot \frac{x}{z} + W, \\ Y &= -\frac{H}{\tan(\Delta_h/2)} \cdot \frac{y}{z} + H \end{aligned}$$

なお、水平（垂直）画角が 180° のときには tan() の値は無限大となり、不都合であるため、画角は 180° よりも小さい値でなければなりません。

さて、画角はどのような大きさが適正なのでしょうか。

人が、特定の対象に意識を集中するのではなく、自然体で景色を視るときの画角はレンズ交換式 35mm カメラの標準レンズの画角に近いと言われています。その標準レンズの画角については諸説ありますが、おおむね水平画角は約 40° 、垂直画角は約 27° です[8]。カメラの用語ではこれよりも画角が大きい場合を**広角**（wide angle）、小さい場合を**望遠**（long focus）と呼んでいます。

p5.js のデフォルトの垂直画角は 60° です。よって上の議論からはかなり広角です。後に述べるように、p5.js では画角をプログラムで変更可能です。

5.4.2 アスペクト比

キャンバス、投影面の水平方向と垂直方向の大きさの比 $2W/2H$、$2w/2h$ を**アスペクト比**（aspect ratio）と呼びます。通常、$2W/2H$ と $2w/2h$ は同じ比率になるように設定します。異なる比率を用いると、描画像が縦長、横長の映像になってしまいますから注意してください（5.4.4 節参照）。

3.3.2 節の表 3.2 に示すように、テレビ、PC ディスプレイのアスペクト比には 4/3 (= 1.33)、3/2 (= 1.5)、16/10 (= 1.6)、16/9 (= 1.78) が用いられています。一般に、横長のほど没入感、臨場感が高くなることが知られています[9]。

p5.js では、キャンバスのアスペクト比は関数 `createCanvas()` で指定する水平方向、垂直方向の大きさの比です。投影面のアスペクト比は、デフォルトではキャンバスのアスペクト比と同じです。後に述べるように、p5.js ではアスペクト比をプログラムで変更可能です。

5.4.3 クリッピング空間

5.3.1 節において $z < 0$ が成り立つ任意の頂点 (x, y, z) を透視投影し、キャンバスに描画できると述べました。

原理的にはそうなのですが、たとえば視点から非常に遠いところにある立方体を考えましょう。その立方体はキャンバス上に芥子粒のような小さな点として描画されるだけで CG 画像への寄与

[8] 35mm カメラで使用する 1 枚のフィルムの横幅は 36mm、縦幅は 24mm です。標準レンズの焦点距離を 50mm と考えると、水平画角は $2\arctan((36/2)/50) \approx 39.59°$、垂直画角は $2\arctan((24/2)/50) \approx 26.99°$ となります。

[9] 「アスペクト比&没入感」で検索してみてください。

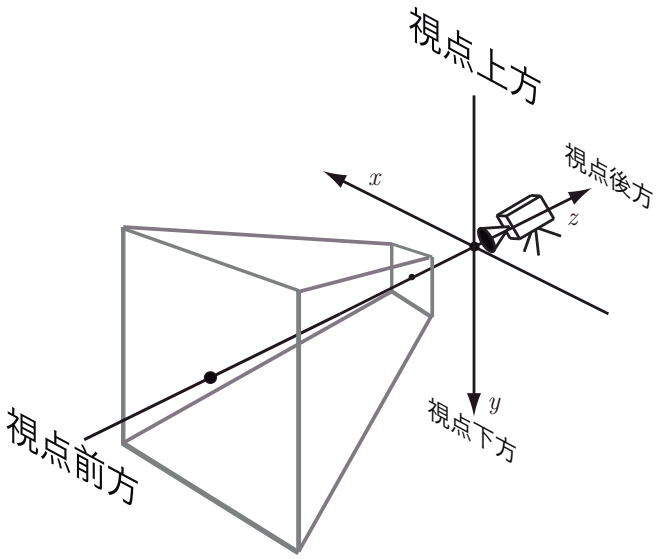


図 5.24: クリッピング空間（錐台の内部のみを描画対象とする）

はほとんどありません。しかし立方体は 8 個の頂点と 12 本の線分から構成されていますから、8 個の点を透視投影し、12 本の線分を描画せねばならないことに変わりはありません[10]。これは全く非効率であって、描画速度を無駄に落とす原因になります。

これを解決するために（処理を端折るために）、正定数値 **far** を導入し、

- 座標値が $-$**far** よりも小さいならば、その頂点を含む被写体を描画しない

ように制限することを考えます。**far** を適切に決めれば、描画像への影響を最小限に抑えつつ描画速度を向上されることができます。実際、これはビデオゲームで活用されています。ゲームの背景が急に現れたり、急に消えたりする現象がそれです。

上とは逆に、立方体が視点に非常に近いところにある場合も不都合です。立方体が CG 画像の枠一杯に描画されるかもしれません。さらに近づくと視点が立方体の内部に入り込んでしまいます。このような事態を避けるために、正定数値 **near** を導入し、

- 座標値が $-$**near** よりも大きいならば、その頂点を含む被写体を描画しない

ように約束しましょう。

このように **far**、**near** で描画を制限すると、描画の対象となる頂点が含まれる領域は図 5.24 のような錐台（すいだい、frustum）になります。これをクリッピング空間（clipping frustum）[11]などと呼んでいます。

[10] 距離に応じて被写体の描画アルゴリズムを変える技術もありますが、ここの内容を超えます。
[11] clipping frustum の直接の日本語訳はクリッピング錐台ですが、錐台という言葉の認知度が低いため、「錐台」は「空間」に置き換えられている場合がほとんどです。

p5.js では、**near** を キャンバスの高さ `height` の 0.1 倍、**far** を同じく 10 倍にデフォルト設定しています。これはプログラムで変更可能です。

5.4.4 p5.js における透視投影変換のパラメータ変更

p5.js では、垂直画角、アスペクト比、クリッピング空間を以下の関数で変更可能です。

`perspective(fovy, aspect, near, far)` ... `fovy` は垂直画角（ラジアンモード）、`aspect` は投影面のアスペクト比、`near`、`far` はクリッピング空間のパラメータです。この関数を呼ばない場合のデフォルトの値が、`fovy` $= 60°$、`aspect` は `createCanvas()` での設定値、`near` はキャンバスの高さ `height` の 0.1 倍、`far` は同じく 10 倍に設定されています。

`perspective(fovy)` ... 垂直画角（ラジアンモード）のみ変更できます。その他のパラメータはデフォルト値のままです。

図 5.25 にこの関数を用いた例題を示します。これは図 5.15 のプログラムを一部改造したもので、立方体が回転しながら前後に運動するアニメーションです。実行の様子が図 5.26 です。

このプログラムでは関数呼び出し：

```
perspective(PI/2, 2, 200, 800);
```

によって透視投影の設定を変更しています。

まず、第 1 引数 `PI/2` は垂直画角を $\pi/2 = 90°$ に設定しています。これはデフォルト値の $60°$ よりもかなり広角です。

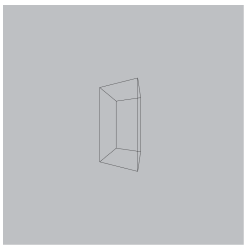
第 2 引数 2 は、投影面のアスペクト比を横に 2 倍長くなるように設定しています。しかしキャンバスのアスペクト比は、このプログラムの場合は $500/500 = 1$ であり、正方形の形状です。結果、横長の画像を正方形のキャンバスに押し込めるため、キャンバス上の CG 画像は縦長になります（図 5.26(a) 参照）。

第 3、4 引数 200、800 は、**near** を 200、**far** を 800 に設定しています。デフォルト値は、それぞれ $0.1 \times$ `height` $= 0.1 \times 500 = 50$、$010 \times$ `height` $= 10 \times 500 = 5000$ ですから、200、800 はクリッピング空間をかなり狭くしています。

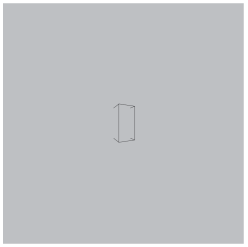
結果、図 5.26(b) では視点から遠ざかった立方体の一部が描画されていません。図 5.26(c) では視点に近づいた立方体の一部が描画されていません。

```
function setup(){
  createCanvas(500,500,WEBGL);
  perspective(PI/2, 2, 200, 800);  // (1)
}

let d = 0;

function draw(){
  translate(0, 0, 0.866*width);
  background(200);

  const zoffset = -500-500*sin(3*d); // (2)
  const s = 100;
  const cs = s * sqrt(2) * cos(d);
  const sn = s * sqrt(2) * sin(d);

  line(-cs, -s, -sn + zoffset,
       +sn, -s, -cs + zoffset);
  line(+sn, -s, -cs + zoffset,
       +sn, +s, -cs + zoffset);
  line(+sn, +s, -cs + zoffset,
       -cs, +s, -sn + zoffset);
  line(-cs, +s, -sn + zoffset,
       -cs, -s, -sn + zoffset);

  line(-sn, -s, +cs + zoffset,
       +cs, -s, +sn + zoffset);
  line(+cs, -s, +sn + zoffset,
       +cs, +s, +sn + zoffset);
  line(+cs, +s, +sn + zoffset,
       -sn, +s, +cs + zoffset);
  line(-sn, +s, +cs + zoffset,
       -sn, -s, +cs + zoffset);

  line(-cs, -s, -sn + zoffset,
       -sn, -s, +cs + zoffset);
  line(+sn, -s, -cs + zoffset,
       +cs, -s, +sn + zoffset);
  line(+sn, +s, -cs + zoffset,
       +cs, +s, +sn + zoffset);
  line(-cs, +s, -sn + zoffset,
       -sn, +s, +cs + zoffset);

  d += 0.01;
}
```

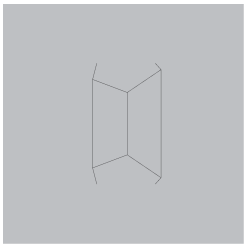
図 5.25: 図 5.15 のプログラムの改造：(1) 透視投影変換パラメータをデフォルト値から変更、(2) 立方体の視点からの距離が時間変化するように変更



(a)



(b)



(c)

図 5.26: 左のプログラムの実行結果：本文参照のこと

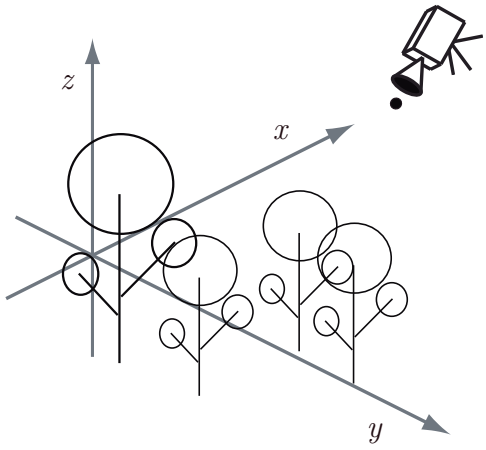


図 5.27: 森を上空のカメラから撮影する様子

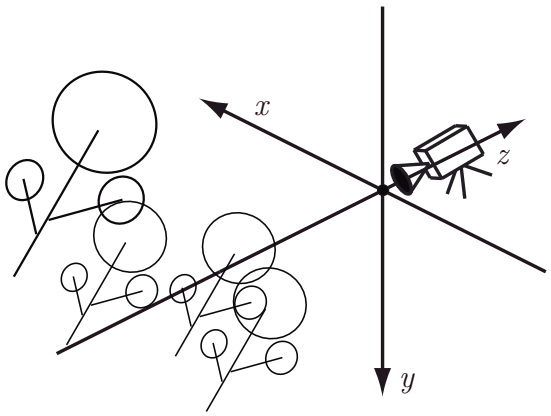


図 5.28: 左の図を前節の座標系へ変換した結果

5.5 視野変換

視点は言わばカメラです。そのカメラを図 5.5 のように原点に固定し、しかもカメラの向きを z 軸の負の方向へ固定したままにすることは制約が強すぎます。カメラを自由な位置に置き、自由な方向へ向けて CG 描画できるべきです。また、現実の撮影ではわざとカメラを左右に傾けて撮影することも行われます。

たとえば図 5.27 は、森を上空のカメラから撮影する様子を考えたものです。x-y 平面を地面に見立て、木が z 軸の上方向に伸びるような座標配置を設定しています。ただし、左手座標系を用いているので、x 軸と y 軸の配置が通常の右手座標系とは逆です。

この図 5.27 と同じ状況を、前節の座標系で表現したものが図 5.28 です。カメラは座標原点にあり、z 軸の負の方向を見ています。森は斜めに配置せざるをえません。

私たちは x-y 平面を水平面とする空間配置に慣れているため、図 5.27 と図 5.28 を比較すれば、図 5.27 の配置が記述しやすいと感じます。しかし、透視投影変換は図 5.28 の座標系を用いて定義する方が単純明快です。そこで、世の中のほとんど全ての CG システムでは

- カメラ、被写体の配置はプログラマの自由な方法を許し、
- 透視投影変換には前節の単純な定式化を利用する

方法を用います。そして両者の間を座標変換で結びます。この節では図 5.27 を図 5.28 へ 3 次元座標変換する手順を解説するものです。

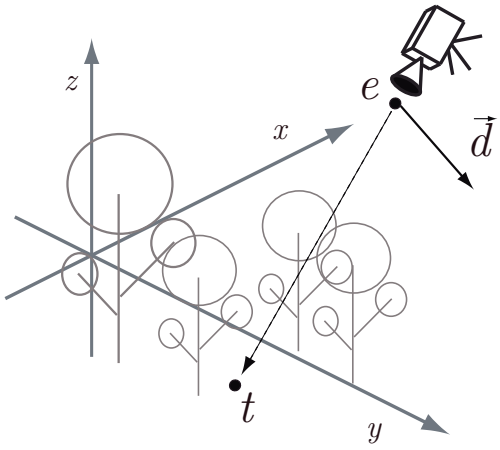


図 5.29: ワールド座標系上の e、t、$\vec{d}$

5.5.1 ワールド座標系

プログラマが視点（カメラ）を自由に設定する座標系を**ワールド座標系**（world coordinate system）[12]と呼びます。この座標系上での視点の状態を決めるパラメータは以下の三つです（図 5.29 参照）。

視点の位置 ... $e = (e_x, e_y, e_z)$ で表すこととします。自由に設定して構いません。

視線の先の位置 ... $t = (t_x, t_y, t_z)$ で表すこととします。視線（view direction）とは視点が向いている正面方向のことです。視線の先の位置は、視点の位置 e から視線方向へ伸びる直線上に無数に存在しますが、その直線上の、プログラマが指定しやすい 1 点を任意に選んで指定して問題ありません。

視点の下向き方向ベクトル [13] ... $\vec{d} = (d_x, d_y, d_z)$ で表すこととします。厳密に視点の真下方向である必要はなく、方向が前後にずれて斜め前下、斜め後ろ下の方向でも構いません。ただし、左右にずれて斜め右下、斜め左下の方向を指定すると、その分だけ投影面が視線の周りに回転します（カメラが傾きます）から注意してください。$\vec{d}$ は正規化されている（ベクトルの長さが 1 である）必要はありません。

さて、上の三つの量を基に、座標変換に必要ないくつかのベクトルを計算しておきましょう。

視線の先の位置 t から視点の位置 e を引くと視線方向ベクトルを求めることができます。

$$\vec{f} = (f_x, f_y, f_x) = (t_x - e_x, t_y - e_y, t_z - e_z)$$

[12] 世界座標系、グローバル座標系（global coordinate system）などとも呼びます。
[13] カメラの上下方向を指定するときには、「上向き方向ベクトル」を指定するのが一般的ですが、p5.js では下向き方向ベクトルを与える仕様になっているため、それに従います。これについては後に補足説明します。

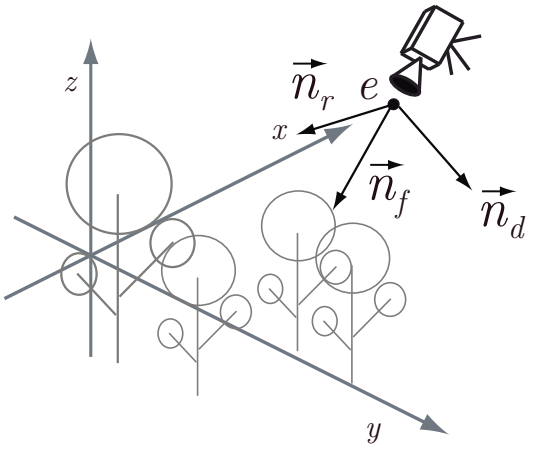


図 5.30: ワールド座標系上の e、$\vec{n}_f$、$\vec{n}_r$、$\vec{n}_d$

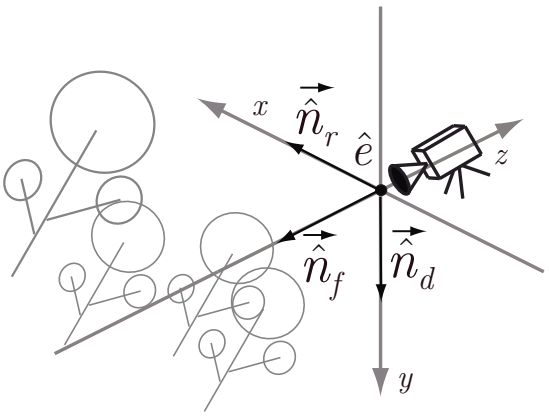


図 5.31: 視点座標系上の $\hat{e}$、$\vec{\hat{n}}_f$、$\vec{\hat{n}}_r$、$\vec{\hat{n}}_d$

さらにこれを以下のように正規化（単位ベクトル化）しましょう。ここに | | はベクトルの大きさを表します。

$$\vec{n}_f = \vec{f}/|\vec{f}|$$

これが視線方向の単位ベクトルです。

上に述べたように、$\vec{d}$ は $\vec{n}_f$ と直交している必要はなく、斜め前の下方向、斜め後ろの下方向でも構いません。というのも、$\vec{d}$ から視線方向 $\vec{n}_f$ に平行なベクトル成分を取り除くと垂直下向きの成分が抽出できるからです。具体的に計算してみます。まず、二つのベクトル $\vec{d}$ と $\vec{n}_f$ に垂直なベクトルで、かつ視線の右方向のベクトルは、ベクトルの外積演算 $\times$ を用いて以下の式で定義できます。

$$\vec{r} = \vec{d} \times \vec{n}_f$$

さらにこれを正規化します。

$$\vec{n}_r = \vec{r}/|\vec{r}|$$

これは視線ベクトル $\vec{n}_f$ に直交する右方向の単位ベクトルです。

次に、$\vec{n}_f$ と $\vec{n}_r$ に垂直で、d と同じ方向（d との内積が正値）であるようなベクトルは、以下のように計算できます。

$$\vec{n}_d = \vec{n}_f \times \vec{n}_r$$

これが視線ベクトル $\vec{n}_f$ および視線の右方向ベクトル $\vec{n}_r$ に直交する下向き方向の単位ベクトルです。

このようにして求められた三つのベクトル $\vec{n}_f$、$\vec{n}_r$、$\vec{n}_d$ は互いに直交し、正規化されています（図 5.30 参照）。

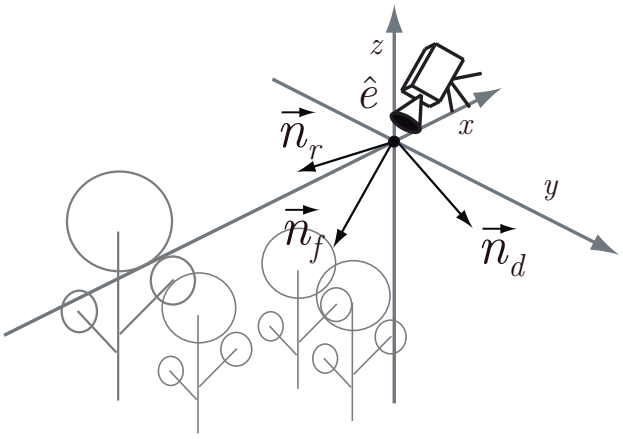


図 5.32: 視点の位置の原点への平行移動（ただし方向ベクトルはこの座標変換に対して不変）

実は前節までの議論は以下の条件が成り立つ座標系を用いています。ただし、一般の場合と区別するため、各記号の上に ˆ を付けました（図 5.31 参照）。

$$\hat{\vec{n}}_f = (0,0,-1) \tag{5.13}$$

$$\hat{\vec{n}}_r = (1,0,0) \tag{5.14}$$

$$\hat{\vec{n}}_d = (0,1,0) \tag{5.15}$$

$$\hat{e} = (0,0,0) \tag{5.16}$$

上の 4 式を満たす座標系（前節の座標系）を**視点座標系**（view coordinate system）[14]と呼びます。

この節の目的は、ワールド座標系上に任意に与えられた e、t、$\vec{d}$ から計算される $\vec{n}_f$、$\vec{n}_r$、$\vec{n}_d$ 、e を基に、ワールド座標系を視点座標系に変換することです。この座標変換を**視野変換**（viewing transformation、view transformation）と呼びます。

視野変換では視点、投影面、被写体を含めた 3 次元仮想世界全体を一斉に座標変換します。一斉変換ですから、仮想世界の視点と被写体の相対的な幾何学関係は変換の前後で不変です。図 5.27 と図 5.28、あるいは図 5.30 と図 5.31 を比較してください。変換後は前節の幾何学関係（図 5.6）が成り立ちますから、前節の透視投影変換の式 (5.3)、式 (5.4) がそのまま利用できます。

5.5.2 視点の位置に関する座標の平行移動

もし視点の位置 $e = (e_x, e_y, e_z)$ が原点 $(0,0,0)$ でないならば、まず e を原点へ**平行移動**（parallel translation）します。このとき、視点以外の座標位置も全て同じく平行移動します。ただし、方向と大きさしか持たないベクトルは平行移動に対して不変ですから、ベクトルについては平行移動の必要はありません。たとえば図 5.32 は、図 5.30 の視点を平行移動した後の状況です。

[14] カメラ座標系（camera coordinate system）、視座標系、viewing coordinate system、eye coordinate system などとも呼びます。

今、この平行移動変換を $P : (x, y, z) \mapsto (x', y', z')$ と表すとき、P は以下の式で定義できます。

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \tag{5.17}$$

この変換によって、式 (5.16) が成り立ちます。

5.5.3 視点の方向に関する基底変換

残る変換は、図 5.32 から図 5.31 への座標変換です。ベクトル $\vec{n}_f$、$\vec{n}_r$、$\vec{n}_d$ が $\vec{\hat{n}}_f$、$\vec{\hat{n}}_r$、$\vec{\hat{n}}_d$ に変換され、それらが式 (5.13)、式 (5.14)、式 (5.15) を満たさねばなりません。

この変換は簡単です。線形代数の知見によれば、x-y-z 直交座標系の上に互いに直交する三つの単位ベクトル：

$$\begin{aligned} \vec{a} &= (a_x, a_y, a_z), \\ \vec{b} &= (b_x, b_y, b_z), \\ \vec{c} &= (c_x, c_y, c_z) \end{aligned}$$

が与えられたとき、$\vec{a}$、$\vec{b}$、$\vec{c}$ を基底ベクトルとする x'-y'-z' 直交座標系への座標変換 $R : (x, y, z) \mapsto (x', y', z')$ は以下の式：

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = R \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{5.18}$$

で与えることができます。上の式に現れる行列を**基底変換行列**（change-of-basis matrix）と呼びます。x-y-z 直交座標系がワールド座標系に相当し、x'-y'-z' 直交座標系が視点座標系に相当します。よって、ここでは $\vec{a}$、$\vec{b}$、$\vec{c}$ は以下のように与えます。

$$\begin{aligned} \vec{a} &= \vec{n}_r = (n_{r,x}, n_{r,y}, n_{r,z}), \\ \vec{b} &= \vec{n}_d = (n_{d,x}, n_{d,y}, n_{d,z}), \\ \vec{c} &= -\vec{n}_f = (-n_{f,x}, -n_{f,y}, -n_{f,z}) \end{aligned}$$

上のように定義された変換の正しさを検証します。まず、$\vec{n}_f$ を変換してみましょう。

$$\begin{aligned}
\vec{\hat{n}}_f &= R \cdot \vec{n}_f \\
&= \begin{pmatrix} n_{r,x} & n_{r,y} & n_{r,z} \\ n_{d,x} & n_{d,y} & n_{d,z} \\ -n_{f,x} & -n_{f,y} & -n_{f,z} \end{pmatrix} \begin{pmatrix} n_{f,x} \\ n_{f,y} \\ n_{f,z} \end{pmatrix} \\
&= \begin{pmatrix} n_{r,x}n_{f,x} + n_{r,y}n_{f,y} + n_{r,z}n_{f,z} \\ n_{d,x}n_{f,x} + n_{d,y}n_{f,y} + n_{d,z}n_{f,z} \\ -n_{f,x}n_{f,x} - n_{f,y}n_{f,y} - n_{f,z}n_{f,z} \end{pmatrix} \\
&= \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}
\end{aligned}$$

となり、変換後のベクトル $\vec{\hat{n}}_f$ は式 (5.13) を満たします。同様に、

$$\vec{\hat{n}}_r = R \cdot \vec{n}_r = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \vec{\hat{n}}_d = R \cdot \vec{n}_d = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

となり、式 (5.14)、式 (5.15) も成り立ちます。

視野変換の手順をまとめます。

1. ワールド座標系上に e、t、$\vec{d}$ が与えられたとします。
2. ベクトル $\vec{n}_f$、$\vec{n}_r$、$\vec{n}_d$ を求めます。
3. 平行移動変換 P を求めます。
4. 基底変換 R を求めます。
5. 全ての被写体の座標位置を $R \cdot P$ で変換します（P を適用し、次に R を適用します)。

そして、最後に投影変換を行います。

5.5.4 p5.js における視野変換

p5.js では、デフォルトでは視野変換は恒等変換です。しかし、以下の関数を呼び出すことで上に述べた平行移動 P および基底変換行列 R が自動的に計算され、p5.js のシステム内に設定されます。そして、その後の被写体の描画時には自動的にこの視野変換 $R \cdot P$ が適用されます。

`camera(eyeX, eyeY, eyeZ, targetX, targetY, targetZ, downX, downY, downZ)` ... 第 1〜第 3 引数 `eyeX`、`eyeY`、`eyeZ` で視点の位置（上述の e）を指定します。第 4〜第 6 引数 `targetX`、`targetY`、`targetZ` で視線の先の位置（上述の t）を指定します。第 7〜第 9 引数 `downX`、`downY`、`downZ` で視点の下向き方向ベクトル（上述の $\vec{d}$）を指定します。

```
function setup() {
  createCanvas(500, 500, WEBGL);
  background(200);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);

  box(200);
}
```

図 5.33: 斜め上から原点にある立方体を見るプログラム

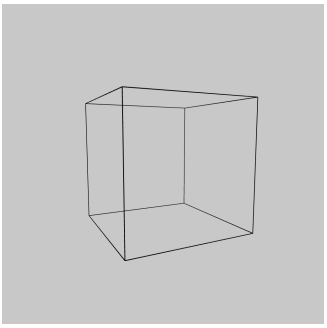


図 5.34: 左のプログラムの実行結果

この関数は、`setup()` から呼び出しても構いませんし、`draw()` から呼び出しても構いません。この関数が呼ばれたときに、直前までの視野変換は無効となり、再設定されます。

立方体の描画

実は p5.js には立方体を描画する関数が用意されています。以下がそれです。

`box(size)` ... 座標原点を中心として、辺が座標軸に平行で、一辺の長さが `size` の立方体を描画します。

`box(xsize, ysize, zsize)` ... 座標原点を中心として、辺が座標軸に平行で、x 軸方向の長さが `xsize`、y 軸方向の長さが `ysize`、z 軸方向の長さが `zsize` の直方体を描画します。

前節までこの関数を用いてこなかった理由は、**`box()` で描画される立方体の中心は 3 次元座標原点にあり、立方体を原点以外の場所に描画できない**制約があるためです。ここまでの議論では立方体を原点以外に描画する必要がありましたから、この関数を利用できませんでした。`box()` を用いて原点以外に立方体を描画するには「モデリング変換」が必要なのですが、それはもう少し先の話になります。ここでは原点に描画される立方体について p5.js の視野変換を解説していきます。

また、`box()` で描く立方体は面画（平面で構成された被写体）に対応しています。しかし面画は次章で解説しますから、この章では線分だけを描画することとし、初期化時には `noFill()` を実行することとします。試しに `noFill()` をコメントアウトしてみてください。立方体の各面が白く描画され、視点から見て裏側の線分が隠れるはずです。

さて、図 5.33 は、辺の長さが 200 の立方体を原点に置き、$(400, 200, 50)$ の位置の視点が原点方向を見ているときの CG 画像です。視点の下方向は $(0, 0, -1)$ です。視点、被写体、座標系の位置関係を図 5.35 に図示しましたので参考にしてください。

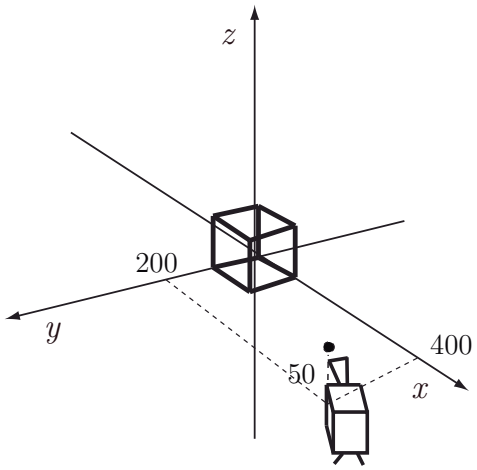


図 5.35: 図 5.33 の視点、被写体、座標系の位置関係

視点が球の表面を移動する

図 5.36 は、図 5.33 のプログラムを改造したもので、視点が原点方向を見ながら球の表面を渦巻き状に回転していくプログラムです。

原点を中心とする半径 r の球の表面の座標は、二つの角度 θ、ϕ を用いて以下のように記述できます。

$$\begin{aligned} x &= r\sin\theta\sin\phi \\ y &= r\sin\theta\cos\phi \\ z &= r\cos\theta \end{aligned}$$

そこで、$r = 500$、$\theta = 0.1$d、$\phi =$ d とおき、d の値を一定間隔で増加させてみました。

補足： 以下、非常に微細な話なのですが、授業担当者も少し自信がないので以下を補足しておきます。p5.js のサイトの関数 camera() の解説では、第 7〜第 9 引数の引数名は upX、upY、upZ となっており（down ではなく up）、上向きベクトルを指定するように書かれてます。実際、p5.js 以外の多くの CG ライブラリでもカメラの上向きベクトルを指定するのが普通で、下向きベクトルを指定するものは見たことがありません。しかし、このテキストでは下向きベクトルを指定すると解説しています。逆にした理由は、p5.js のサイトの解説の文章には “with the Y axis as up” と書かれており、これは y 軸の上向き方向を指定するという意味と考えています。y 軸はキャンバス上では下向きに値が増えますから、結局、CG 画像を観る者の立場では下向きベクトルのことを意味すると授業担当者は解釈しています。実際、その解釈で現時点では矛盾がないのですが、もし問題を発見したならば連絡してください。

```
function setup() {
  createCanvas(500, 500, WEBGL);
  noFill();
}


let d = 0;

function draw() {
  background(200);

  const csph = cos(d);
  const snph = sin(d);
  const csth = cos(0.1 * d);
  const snth = sin(0.1 * d);
  camera(500*snth*csph, 500*snth*snph, 500*csth,
    0, 0, 0,
    0, 0, -1);

  box(200);

  d += 0.05;
}
```

図 5.36: 図 5.33 のプログラムの改造

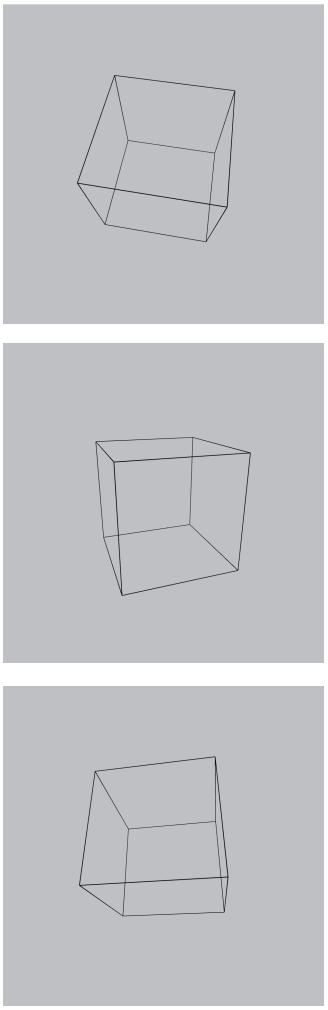


図 5.37: 左のプログラムの実行の様子

第IX部

5.6　同次座標系による高速化

今、複数の座標変換 f_1、f_2、...、f_k を用いて座標点 X を次々と変換する以下の式について考えてみます。

$$X' = f_k(\cdots(f_2(f_1(X)))\cdots)$$

もしこれら座標変換が行列 M_1、M_2、...、M_k で与えられており、かつその全ての行列要素が座標点に依存しない不変な定数値であるならば、演算の順序を変更し、

$$\begin{aligned} &= M_k(\cdots(M_2(M_1X))\cdots) \\ &= (M_k\cdots M_2\cdot M_1)X \\ &= \hat{M}X \end{aligned}$$

と変形できます。ここに $\hat{M} = M_k\cdots M_2\cdot M_1$ は、$k-1$ 回の行列間の積を事前に計算し、ひとつにまとめた行列を表します。これによって、k 回の行列とベクトルの積の計算

$$X_1 = M_1X,\quad X_2 = M_2X_1,\quad \ldots\quad X_{k-1} = M_{k-1}X_{k-2},\quad X' = M_kX_{k-1}$$

を 1 回の計算

$$\hat{M}X$$

に省力化できます。同じことを前節の座標の平行移動（5.5.2 節）、基底変換（5.5.3 節）、投影変換（5.3.1 節）、キャンバス座標への変換（5.2 節）に適用できるでしょうか。もしできるならば、膨大な計算を必要とする 3D CG の計算は大幅に高速化できます。

直ちに気づくことは、前節までの議論では基底変換以外は行列形式で表されていないことです。これを解決する手段として、**同次座標系**（homogeneous coordinate、斉次座標系とも呼びます）を用います。興味深いことに、同次座標系では 3D CG の座標変換を全て行列として表すことができます。

ここまでの p5.js のプログラムには行列の話は全く出ていません。しかし、次節のモデリング変換はこの行列による座標変換を想定していますので、次節に進む前にこれを知っておく必要があります。

5.6.1　定義

同次座標系は、対象とする座標系に次元をひとつ追加して融通性を持たせ、数学的な操作を容易にした座標系です。

3 次元座標系には 4 次元同次座標系が対応します。通常の座標系との違いを明確にするために、同次座標系上の点を、以下では

$$[\bar{x}, \bar{y}, \bar{z}, \bar{w}]$$

という形式で表すこととします[15]。$\bar{x}$、$\bar{y}$、$\bar{z}$、$\bar{w}$ のそれぞれは実数値です。そして、$[\bar{x}, \bar{y}, \bar{z}, \bar{w}]$ は、通常の 3 次元座標系の点

$$(\bar{x}/\bar{w}, \bar{y}/\bar{w}, \bar{z}/\bar{w})$$

に対応すると見なします[16]。ここではこの対応関係を以下のように表すこととします。

$$[\bar{x}, \bar{y}, \bar{z}, \bar{w}] \mapsto \left(\frac{\bar{x}}{\bar{w}}, \frac{\bar{y}}{\bar{w}}, \frac{\bar{z}}{\bar{w}}\right) \tag{5.19}$$

逆に、通常の 3 次元座標系上の点 (x, y, z) は、任意の値 w を用いて、4 次元同次座標系上の点 $[wx, wy, wz, w]$ に対応させることができます。つまり

$$\forall w,\ \ (x, y, z) \mapsto [wx, wy, wz, w] \tag{5.20}$$

ここに w が任意であるため、通常の座標系の上の 1 点が同次座標系の上の複数の点（より詳しく言えば、4 次元座標系上の原点を通る直線）に対応することに注意してください。なお、多くの場合には単純に $w = 1$ と置いて以下を用います。

$$(x, y, z) \mapsto [x, y, z, 1]$$

純粋数学的には、同次座標系の議論は射影空間の理論へ拡がっていくのですが、それはこの授業の趣旨からは外れます。私たちは同次座標系を 3D CG の高速描画へ応用する立場で用います。

5.6.2 平行移動

同次座標系の興味深い点は、座標系上の平行移動を行列形式で表現できることです。以下は視点が原点にない場合の平行移動変換を表す式 (5.17) の再掲です。

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix}$$

この変換を同次座標系の変換 $P : [\bar{x}, \bar{y}, \bar{z}, \bar{w}] \to [\bar{x}', \bar{y}', \bar{z}', \bar{w}']$ と表すとき、以下のような行列形式で表現できます。

$$\begin{bmatrix} \bar{x}' \\ \bar{y}' \\ \bar{z}' \\ \bar{w}' \end{bmatrix} = P \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} \tag{5.21}$$

[15] これ以外にも $(\bar{x} : \bar{y} : \bar{z} : \bar{w})$ と表す方法もあります。通常のベクトル表記 $(\bar{x}, \bar{y}, \bar{z}, \bar{w})$ を用いる教科書もあります。

[16] CG とは直接関係しませんが、同次座標系の興味深い性質のひとつとして、もし $\bar{w} = 0$ ならば、$\bar{x}/\bar{w}$、$\bar{y}/\bar{w}$、$\bar{z}/\bar{w}$ は無限大となります。3 次元座標系では座標値が無限大になるような特殊な座標点を同次座標系では有限値として取り扱うことができるという利点があります。

ここに行列とベクトルの積の計算方法は通常の行列計算と同じです。実際、式 (5.20) の座標点を用いて上の式の右辺を計算すると、任意の w について

$$\begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix} = \begin{bmatrix} wx - we_x \\ wy - we_y \\ wz - we_z \\ w \end{bmatrix} \mapsto \begin{pmatrix} x - e_x \\ y - e_y \\ z - e_z \end{pmatrix}$$

となり、確かに平行移動の式と等価であることが分かります。

5.6.3 基底変換

5.5.3 節の式 (5.18) はもともと行列の形ですから、同次座標系の変換行列を作るには単純に式 (5.18) を 4×4 行列に埋め込めばよいだけです。この変換を同次座標系の変換 $R : [\bar{x}, \bar{y}, \bar{z}, \bar{w}] \to [\bar{x}', \bar{y}', \bar{z}', \bar{w}']$ と表すとき、以下の通りです。

$$\begin{bmatrix} \bar{x}' \\ \bar{y}' \\ \bar{z}' \\ \bar{w}' \end{bmatrix} = R \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} \tag{5.22}$$

5.6.4 透視投影変換

次に透視投影変換を同次座標系の行列計算で実現します。同時にクリッピングに関する補助的な計算も追加します。

変換行列の基本形

以下は 5.3.1 節の式 (5.3)、式 (5.4) の透視投影変換式の再掲です。

$$\begin{aligned} x' &= -d \cdot \frac{x}{z}, \\ y' &= -d \cdot \frac{y}{z} \end{aligned}$$

これを同次座標系の変換行列に埋め込むことを考えます。上の式の右辺の分母に z が現れている点に注意してください。行列計算では除算はできませんが、式 (5.19) の右辺では $\bar{w}$ で割る計算を行っています。そこに着目し、以下のような変換行列を用意します。この変換を変換 $J : [\bar{x}, \bar{y}, \bar{z}, \bar{w}] \to [\bar{x}', \bar{y}', \bar{z}', \bar{w}']$ と表しましょう[17]。

[17] 記号 P、R は既に平行移動変換、基底変換に用いているため、透視変換を proJection の J で表すことにしました。

$$\begin{bmatrix} \bar{x}' \\ \bar{y}' \\ \bar{z}' \\ \bar{w}' \end{bmatrix} = J \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} \tag{5.23}$$

なお、定数値 A、B については後に議論することとします。式 (5.20) を用いて上の式の右辺を計算すると以下の通りです。

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix} = \begin{bmatrix} wdx \\ wdy \\ Awz + Bw \\ -wz \end{bmatrix} \mapsto \begin{pmatrix} -dx/z \\ -dy/z \\ -A - B/z \end{pmatrix} \tag{5.24}$$

式 (5.3)、式 (5.4) の x'、y' が正しく計算できています。

クリッピングの指標の計算

次に式 (5.24) の最右辺の第 3 成分 $-A - B/z$ の使い方を考えます。「この値を CG 描画に用いない」と割り切るのもひとつの考え方です。その場合、A、B は任意に設定して構いませんし、単純に 0 と置いてもよいでしょう。しかし多くの CG システムは $-A - B/z$ をクリッピング（描画の抑制）（5.4.3 節参照）に利用しています。

クリッピングは、

- 視野変換後の、被写体の位置の z 座標値が $-\mathbf{far} \leq z \leq -\mathbf{near}$ を満たすならば被写体を描画し、さもなくば描画しない

というルールでした（5.4.3 節参照）。これを以下のように読み替えてみます。

- 視野変換後の、被写体の位置の z 座標値が $-1 \leq -A - B/z \leq 1$ を満たすならば被写体を描画し、さもなくば描画しない

これによって描画の判定の範囲が $[-1, 1]$ に正規化されるため、様々な計算が単純になるという利点があります。この方針のもと、境界条件：

$$-A - B/(-\mathbf{far}) = -1, \qquad -A - B/(-\mathbf{near}) = 1 \tag{5.25}$$

を設け、これを解くと定数 A、B が以下のように定まります。

$$A = \frac{\mathbf{far} + \mathbf{near}}{\mathbf{far} - \mathbf{near}}, \qquad B = \frac{2 \cdot \mathbf{far} \cdot \mathbf{near}}{\mathbf{far} - \mathbf{near}} \tag{5.26}$$

参考までに z に関する $-A - B/z$ のグラフを図 5.38 に載せます。z が $[-\mathbf{far}, -\mathbf{near}]$ の範囲内ならば、そのときに限り、$-1 \leq -A - B/z \leq 1$ であることを確認してください。

このようにして求めた $-A - B/z$ を**正規化された深さ値**（normalized depth value）などと呼びます[18]。

[18] 正規化された深さ値の決め方を $[-1, 1]$ ではなく、$[0, 1]$ に設定している CG システムもあります。その

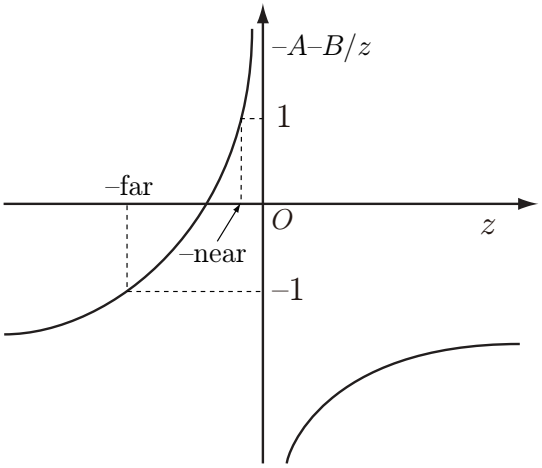


図 5.38: クリッピング指標 $-A - B/z$ のグラフ

5.6.5 キャンバス座標への変換

投影面上の点 (x', y') をキャンバス上の画素点 (X, Y) へ変換するには 式 (5.1)、式 (5.2) を用います。この変換を同次座標系の変換 $C : [\bar{x}, \bar{y}, \bar{z}, \bar{w}] \to [\bar{x}', \bar{y}', \bar{z}', \bar{w}']$ で表すとき、以下の通りです。ただし $\bar{z} = \bar{z}'$、$\bar{w} = \bar{w}'$ とします。

$$\begin{bmatrix} \bar{x}' \\ \bar{y}' \\ \bar{z}' \\ \bar{w}' \end{bmatrix} = C \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} = \begin{bmatrix} W/w & 0 & 0 & W \\ 0 & H/h & 0 & H \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} \tag{5.27}$$

5.6.6 高速化

ここまで定義してきた四つの変換 P、R、J、C を合成しましょう。つまり、視点の位置、方向が決まったならば、以下の行列の合成：

$$\hat{M} = C \circ J \circ R \circ P \tag{5.28}$$

$$= \begin{bmatrix} W/w & 0 & 0 & W \\ 0 & H/h & 0 & H \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \circ \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \circ \begin{bmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \circ \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.29}$$

場合、式 (5.25) の条件を変えて A、B を求めます。

表 5.1: 座標変換の実行時間

プログラムの種類	実行時間（1 座標点当たり）	比率
同次座標系を用いず、座標変換を事前に合成しない場合	13.46	1.00
同次座標系を用い、座標変換を事前に合成する場合	6.73	0.50

実行時間の単位はナノ秒 $= 10^{-9}$ 秒

をあらかじめ計算しておきます。そして、ワールド座標系における被写体の頂点の座標位置 (x, y, z) 毎に、

1. それを同次座標系へ対応させた点 $[x, y, z, 1]$（この場合、$w = 1$ を採用）を求め、
2. 行列 $\hat{M}$ を用いて座標変換を行い、変換後の点 $[\bar{x}', \bar{y}', \bar{z}', \bar{w}']$ を求め、
3. 3 次元座標点 $(x', y', z') = (\bar{x}'/\bar{w}', \bar{y}'/\bar{w}', \bar{z}'/\bar{w}')$ へ戻し、
4. 正規化された深さ値 z' が $[-1, 1]$ の範囲ならば、キャンバス上の位置 (x', y') に被写体を描画します。さもなくば描画しません。

p5.js では関数 `createCanvas()`、`perspective()`、`camera()` が呼び出される度に上の行列 $\hat{M}$ が再計算され、処理系内部に保持されます。

4 次元同次座標系上の行列を用いた、この計算手法は、全ての CG システム、CG ライブラリ、GPU のハードウェアが採用している方法です。p5.js においても採用されています。行列内の具体的な行列要素の値を知る必要はありませんが、このような行列によって 3 次元座標点が 2 次元画素点へ一気に座標変換されていることを知っておく必要があります。

寄り道：同次座標系を用いた高速化は本当に有効なのか？

4.4.2 節の最後にブレゼンハムのアルゴリズムの有効性を検証したように、同次座標系を用いないプログラムと同次座標系を用いるプログラムの実行速度を C++を用いて比較しました[19]。

詳細な実験方法は省略します[20]が、表 5.1 がその実験結果です。同次座標系を用いた方が 2 倍高速であることが分かります。四つの変換をひとつに合成したにも関わらず、2 倍の高速化しか得られていない理由は、コンパイラが同次座標系を用いないプログラムの四つの変換をアセンブリコードレベルで自動合成する最適化を行なっているためです[21]。

[19] p5.js では同次座標系を用いるアルゴリズムが標準でシステムに組み込まれているため、検証には不向きです。

[20] アップル Mac Mini（Intel Core i7、2.6GHz）の UNIX 環境上において `c++ -O3` でコンパイルし、被写体の座標点をキャンバスの画素点へ変換する部分のみの実行時間を 8 万回繰り返し、マシンサイクル数をカウントしました。

[21] コンパイルオプション `c++ -O3 -S` でコンパイラは生成するアセンブリコードをファイル出力し、そのコードの処理内容を調べました。

```
function setup() {
  createCanvas(500, 500, WEBGL);
  background(200);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);

  box(200);
}
```

図 5.39: 図 5.33 の再掲

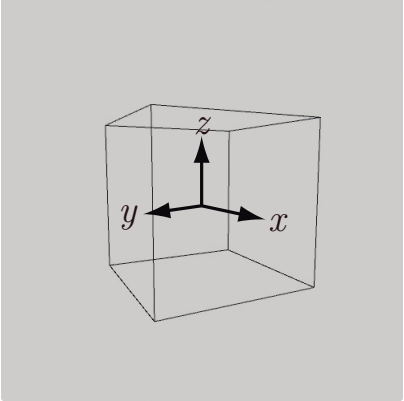


図 5.40: 左のプログラムの実行結果（座標軸はこれ以降の説明のために記載している）

5.7　モデリング変換

複雑な被写体は、多くの部品を組み合わせることで作ることができます。個々の部品は、単純な形状を変形し、あるいはさらに単純な部品を組み合わせて作ることができます。被写体や部品を設計するときには、それぞれを座標の中心に置いて考えた方が取り扱いが容易です。また、同じ形状の複数の被写体、部品を用いるときには、それぞれに個別の座標系で同じ形状を描画すれば、プログラムを単純化できます。

それら被写体や個々の部品の作るときに想定する座標系を、それぞれの被写体、部品に関する**モデル座標系**（modeling coordinate system、model coordinate system）、**ローカル座標系**（local coordinate system）などと呼びます。5.5.1 節で定義したワールド座標系が、文字通り、仮想世界全体を表し、その中に様々な被写体が配置するための座標系であるのに対して、モデル座標系はそれぞれの被写体、部品の形状、配置に利用する座標系です。部品の座標系、部品の部品の座標系など、幾重にも階層化された座標系群をまとめてモデル座標系と呼んでいます。

この節の内容は少し分かりにくいため、できるだけ多くの例題を通して丁寧に解説していきます。

5.7.1　モデル座標系

ここではモデル座標系の概念を複数の立方体の描画を例に順に解説していきます。

原点の回りの立方体の 3D CG

図 5.39 は図 5.33 の再掲です。なお、これ以降の説明のため、図 5.40 にはワールド座標系の座標軸を書き込みました。既に説明しているように、x-y 座標系を水平面に設定し（ただし左手座標系

```
function setup() {
  createCanvas(500, 500, WEBGL);
  background(200);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);

  translate(-400, 110, -250);
  box(200);
}
```

図 5.41: 図 5.39 のプログラムに平行移動を追加したプログラム

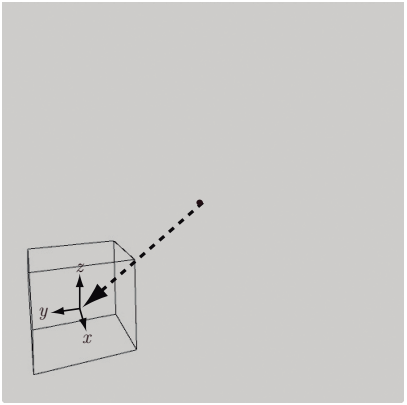


図 5.42: 左のプログラムの実行結果

なので、x 軸と y 軸の配置が通常の逆)、z 軸の上方向をカメラの上方向に設定しています。

座標系の平行移動

関数 box() は、立方体、直方体を原点の周りにしか描画しません。実際、上述の box() の仕様には立方体の 3 次元座標位置を指定する引数はありません。

原点以外の場所に立方体を描画したいときには、p5.js では座標変換を行い、座標原点を描画したい位置へ移動させる方法を用います。以下の関数が、座標系を平行移動させます。

translate(x, y) ... 標準の 2 次元 CG の場合に、座標系を $(\mathtt{x}, \mathtt{y})$ だけ平行移動します。関数呼び出し前の $(\mathtt{x}, \mathtt{y})$ の位置が、呼び出し後の座標原点になります。

translate(x, y, z) ... 3D CG の場合に、座標系を $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ だけ平行移動します。関数呼び出し前の $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ の位置が、呼び出し後の座標原点になります。なお、この移動では視点は移動しませんから、関数呼び出し後の視点の座標位置は呼び出し前の座標位置から逆に $(-\mathtt{x}, -\mathtt{y}, -\mathtt{z})$ だけ移動します。

図 5.41 は、座標原点を $(-400, 110, -250)$ へ平行移動し、その周りに立方体を描画するプログラムです。このプログラムの実行結果（図 5.42）では、画像の奥（x 軸の負の方向）へ 400、左（y 軸の正の方向）へ 110、下（z 軸の負の方向）へ 250 の座標原点の平行移動を行い、そこを中心として立方体を描画しています。

三つの立方体の組み合わせ

次に図 5.43 は、関数 translate() を 3 回呼び出して三つの立方体をピラミッド状に配置するプログラムです。ただし、立方体どうしは密着せず、距離にして 20 だけ離すことにし、x 方向か

```
function setup() {
  createCanvas(500, 500, WEBGL);
  background(200);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);

  translate(-400, 110, -250);
  box(200);

  translate(0, -110, 220);
  box(200);

  translate(0, -110, -220);
  box(200);
}
```

図 5.43: 図 5.41 のプログラムに 2 個の立方体を追加したプログラム

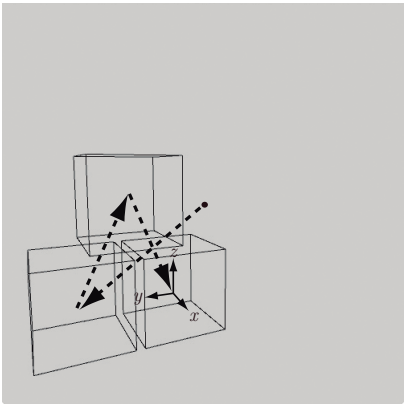


図 5.44: 左のプログラムの実行結果

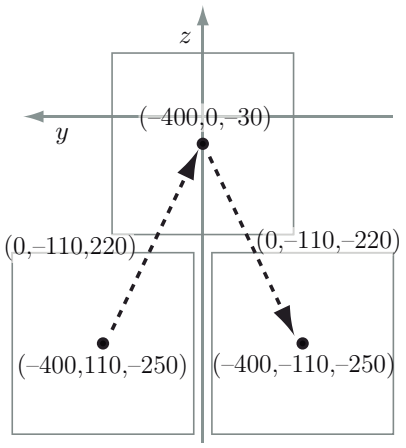


図 5.45: 小さなピラミッドの設計図

ら見た立方体の配置は図 5.45 のように設定します。左下の立方体は、図 5.41 のプログラムと同じ配置にしました。そうすると上の立方体の中心位置は、左下のそれから見て $(0, -110, 220)$ だけ移動した位置にあります。また、右下の立方体の中心位置は、上のそれから見て $(0, -110, -220)$ だけ移動した位置にあります。そこで、図 5.43 のプログラムでは、座標の平行移動と立方体の描画を 3 回繰り返しています。分かりやすさのため、実行結果の図 5.44 にも座標の平行移動を書き加えました。

第X部

```
function setup() {
  createCanvas(500, 500, WEBGL);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);

  translate(-400, 110, -250);
  box(200);

  translate(0, -110, 220);
  box(200);

  translate(0, -110, -220);
  box(200);

  translate(0, -220, 0);
  box(200);

  translate(0, -110, 220);
  box(200);

  translate(0, -110, -220);
  box(200);
}
```

図 5.46: 図 5.43 のプログラムにピラミッドを追加したプログラム

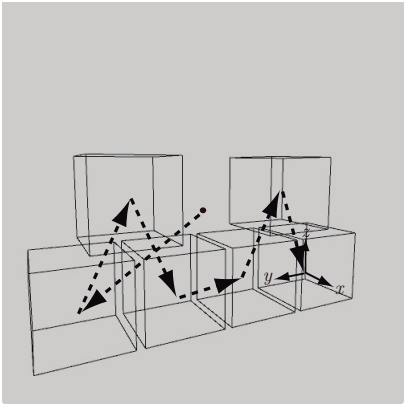


図 5.47: 左のプログラムの実行結果

二つのピラミッド

さらに、上に描画した三つの立方体からなる小さなピラミッドをもうひとつ追加し、横に並べることを検討しましょう。ここまでの方法を適用し、座標の平行移動と描画をさらに繰り返すことも可能です。図 5.46 がそのプログラムです。

しかしこの方法が早晩行き詰まることは明らかです。

まず、類似の処理内容を繰り返し記述することは良いプログラミングスタイルではありません。類似の処理はモジュール化（関数化）して集中管理し、再利用すべきです。図 5.46 のプログラムの場合、三つの立方体のピラミッドの描画をひとつの関数にまとめるべきです。しかし、関数化してもなお以下の問題は残ります。

図 5.46 のように単純に平行移動を繰り返す方法では、移動が増えれば増えるほど各時点での座標原点の位置を把握しにくくなっていきます。原理的にはその時点までに実行された平行移動の総和がその時点での座標原点の位置になりますが、プログラムに条件分岐やループが含まれていたり、あるいは時間と共に位置が変換する CG アニメーションを行う場合には座標原点の位置の管理

```
function setup() {
  createCanvas(500, 500, WEBGL);       // A
  background(200);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);
                                       // B

  translate(-400,110,-250);            // C
  tripleBox();                         // D

  translate(0,-440,0);                 // E
  tripleBox();                         // F
}

function tripleBox(){
  push();                              // G

  box(200);                            // H

  translate(0,-110,220);               // I
  box(200);                            // J

  translate(0,-110,-220);              // K
  box(200);                            // L

  pop();                               // M
}
```

図 5.48: 設定スタックを想定し、push()、pop() で座標変換を制御しているプログラム

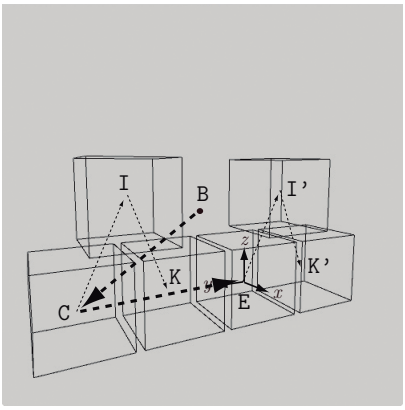


図 5.49: 左のプログラムの実行結果：B→C→E の平行移動に注目

は困難になります。この問題は深刻です。これを解決するために、p5.js には座標変換をスタック[22]を用いて階層的に管理する方法が用いられます。次の例題では図 5.46 のプログラムをその方法で書き直します。

描画設定スタック

ここで検討する図 5.48 のプログラムの実行結果（図 5.49) は、図 5.46 のプログラムの実行結果（図 5.47）と同じものですが、プログラムの構造は両者で異なります。

まず図 5.48 で呼び出されている push()、pop() について解説します。

push() ... この関数が呼び出された時点での様々な描画の設定をスタックへ積みます[23]。スタック

[22] スタックは「最初に格納したデータが最後に取り出される」（first-in last-out）構造のデータの並びのことです。

[23] p5.js のホームページの Reference によると、push() は以下の関数の設定をスタックに格

に格納された設定は push() 実行後も継続して描画に利用されます。以下、このスタックを**描画設定スタック**（簡単に設定スタック）（rendering setting stack）と呼ぶこととします。

pop() ... この関数が呼び出された時点での描画の設定を破棄し、設定スタックの先頭（頂上）に積まれている設定を復元します。

上の説明だけでは分かりにくいため、実際に図 5.48 のプログラムの動作を A から M までの 1 行ずつ解説します。まず、A の実行が終わった時点での設定スタックと現在の使用中の設定を以下のように表すとします。これはキャンバスが作られた直後の初期設定です。

位置	設定スタック	使用中の描画設定	コメント
A		$\hat{S}_0$	初期設定

次に B の行を実行すると

B		$\hat{S}_1$	視野変換の設定

camera() は使用中の設定を上書きします。この時点では、座標原点は図 5.49 の B の位置にあります。次に C を実行すると

C		$\hat{S}_2$	平行移動

C の行の translate() による平行移動は使用中の設定に追加されます。この時点で座標原点は図 5.49 の C の位置にあります。

次に D を実行すると、関数 tripleBox() へ実行が移動します。関数呼び出しでは描画の設定は不変です。そして、G を実行すると、

G	$\hat{S}_2$	$\hat{S}_2$	設定の push

つまり、push() が実行されて、使用中の設定が設定スタックの頂上にコピーされて積まれます。次に H を実行すると、図 5.49 の C の位置に立方体を描画します。描画自体では設定は不変です。そして次に I を実行すると、

I	$\hat{S}_2$	$\hat{S}_3$	平行移動

translate() による平行移動が設定に追加されます。座標原点は図 5.49 の I の位置に移動しますから、J を実行するとこの位置に立方体が描画されます。次に K を実行すると、

K	$\hat{S}_2$	$\hat{S}_4$	平行移動

納し ま す。fill(), noFill(), noStroke(), stroke(), tint(), noTint(), strokeWeight(), strokeCap(), strokeJoin(), imageMode(), rectMode(), ellipseMode(), colorMode(), textAlign(), textFont(), textSize(), textLeading(), applyMatrix(), resetMatrix(), rotate(), scale(), shearX(), shearY(), translate(), noiseSeed(), setCamera(), ambientLight(), directionalLight(), pointLight(), texture(), specularMaterial(), shininess(), normalMaterial() and shader()。Reference には載っていませんが、perspective()、camera() などの関数による座標変換の設定もちろん含まれます。この授業で扱っていない関数の詳細は各自で調べてください。

座標原点は図 5.49 の K の位置に移動しますから、L を実行することでこの位置に立方体が描画されます。

関数 tripleBox() の最後の行 M を実行すると、

M		$\hat{S}_2$	設定の pop

つまり、pop() が実行されて、設定スタックの頂上に保存されている設定が取り出され、それが使用中の設定を上書きします。これによって関数 tripleBox() が呼び出される直前の状態が復元されます。

プログラムの実行はこの後、E、F、G、...、M と進みますが、その間の状態の変化は以下の通りです。

位置	設定スタック	使用中の描画設定	コメント
E		$\hat{S}_5$	平行移動
G	$\hat{S}_5$	$\hat{S}_5$	設定の push
I	$\hat{S}_5$	$\hat{S}_6$	平行移動
K	$\hat{S}_5$	$\hat{S}_7$	平行移動
M		$\hat{S}_5$	設定の pop

ここで重要なことは、関数 setup() のレベルで見た場合、座標原点は B、C、E へと移動しており、関数 tripleBox() 内での座標原点の移動を関知する必要がないことです。また、関数 tripleBox() の中では呼び出された時点の座標原点を起点として I、K（または I'、K'）という移動のみを考えればよく、この関数を呼び出した側の状況を知る必要はありません。この方法では座標の移動を各階層で局所的に理解すれば足りるため、プログラムの書き易さ/読み易さが格段に向上します。

既に紹介したように、このように局所的に想定される座標系をモデル座標系と呼んでいます。モデルとはひな型、模型のことです。ここでは三つの立方体で作られるピラミッドがそのひな型であり、ひな型を作る目的に特化された座標系がモデル座標系です。

大きなピラミッド

上述の立方体 3 個からなる小さなピラミッドをさらに 3 個重ねて、立方体 9 個からなるピラミッドを作ってみたプログラムが図 5.50 です。

このプログラムで取り扱う座標系は 3 階層からなります。ひとつは関数 setup() で想定している座標系です。これはワールド座標系です。関数 nonupleBox()、tripleBox() で想定されている座標系はそれぞれのレベルのモデル座標系です。表 5.2 に設定スタックの変化を時系列で示します。設定の退避/回復の様子を確認してください。

```
function setup() {
  createCanvas(500, 500, WEBGL);  // A
  background(200);
  noFill();
  camera(400, 200, 50, 0, 0, 0, 0, 0, -1);
                                  // B

  translate(-400, 110, -250);     // C
  nonupleBox();                   // D
}

function nonupleBox() {
  push();                         // E

  tripleBox();                    // F

  translate(0, -220, 440);        // G
  tripleBox();                    // H

  translate(0, -220, -440);       // I
  tripleBox();                    // J

  pop();                          // K
}

function tripleBox() {
  push();                         // L

  box(200);                       // M

  translate(0, -110, 220);        // N
  box(200);                       // P

  translate(0, -110, -220);       // Q
  box(200);                       // R

  pop();                          // S
}
```

図 5.50: push()、pop() を用いた 3 階層のプログラム

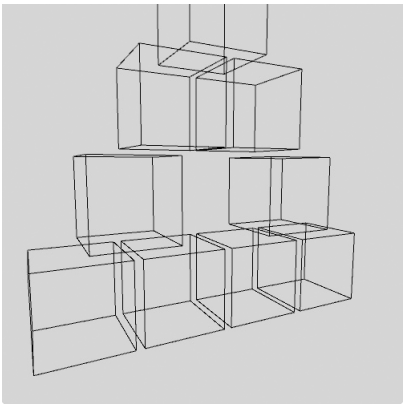


図 5.51: 左のプログラムの実行結果

表 5.2: 設定スタックの様子

位置	設定スタック	使用中の描画設定	コメント
A		$\hat{S}_0$	初期設定
B		$\hat{S}_1$	視野変換の設定
C		$\hat{S}_2$	平行移動
E	$\hat{S}_2$	$\hat{S}_2$	設定の push
L	$\hat{S}_2$ $\hat{S}_2$	$\hat{S}_2$	設定の push
N	$\hat{S}_2$ $\hat{S}_2$	$\hat{S}_3$	平行移動
Q	$\hat{S}_2$ $\hat{S}_2$	$\hat{S}_4$	平行移動
S	$\hat{S}_2$	$\hat{S}_2$	設定の pop
G	$\hat{S}_2$	$\hat{S}_5$	平行移動
L	$\hat{S}_2$ $\hat{S}_5$	$\hat{S}_5$	設定の push
N	$\hat{S}_2$ $\hat{S}_5$	$\hat{S}_6$	平行移動
Q	$\hat{S}_2$ $\hat{S}_5$	$\hat{S}_7$	平行移動
S	$\hat{S}_2$	$\hat{S}_5$	設定の pop
I	$\hat{S}_2$	$\hat{S}_8$	平行移動
L	$\hat{S}_2$ $\hat{S}_8$	$\hat{S}_8$	設定の push
N	$\hat{S}_2$ $\hat{S}_8$	$\hat{S}_9$	平行移動
Q	$\hat{S}_2$ $\hat{S}_8$	$\hat{S}_{10}$	平行移動
S	$\hat{S}_2$	$\hat{S}_8$	設定の pop
K		$\hat{S}_2$	設定の pop

```
function setup() {
  中略
}

function nonupleBox() {
  中略
}

function tripleBox() {
  push();

  box(200);
  sphere(100);            // 球の描画

  translate(0, -110, 220);
  box(200);
  sphere(100);            // 球の描画

  translate(0, -110, -220);
  box(200);
  sphere(100);            // 球の描画

  pop();
}
```

図 5.52: 立方体の中に球を描画するプログラム

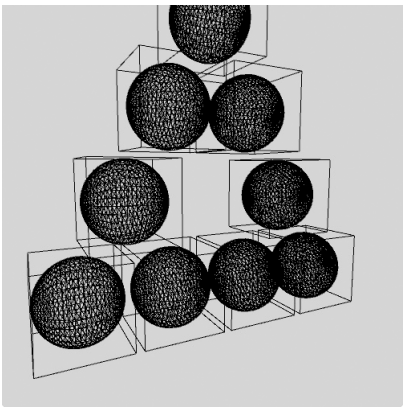


図 5.53: 左のプログラムの実行結果

球の描画

p5.js では球も、直方体（box()）と同様に、原点を中心として描画可能です。以下の関数がそれです。なお、球は 3 角形を貼り合わせた多面体[24]で近似して描画します。その多面体の数は引数で指定できます。

sphere(radius, detailX, detailY) ... 第 1 引数 radius に球の半径を指定します。第 2 引数 detailX に球の緯度を多面体で分割する数を指定します。第 3 引数 detailY に球の経度を多面体で分割する数を指定します。指定する引数は後ろから順に省略できて、全ての引数を省略した場合のデフォルト値は radius = 50、detailX = 24、detailY = 16 になるようです[25]。なお、北極、南極は y 軸上にあります[26]。

図 5.52 は、各立方体の中にそれと内接する球を描き加えるプログラムです。関数 setup()、nonupleBox() は図 5.50 のプログラムと同じです。

[24] 地球儀を思い出しましょう。
[25] デフォルト値についてホームページに記載がないため、別途調査しました。
[26] p5.js では y 軸を暗黙の上下方向とみなしているためです。このテキストの 5.5 節以降では z 軸を上下方向としています。

5.7.2　さまざまな座標変換

p5.js では平行移動以外にも様々な座標変換が用意されています。これらを用いることで被写体の見た目を変形させることができます。2 次元と 3 次元に同等の変換が定義されていますから、ここでまとめて紹介します。

回転

原点を中心とした回転変換です。p5.js では正の角度方向が時計回りと約束されています。

まず、2 次元の回転変換を行う関数は以下の通りです。

`rotate(angle)` ... X-Y 座標系を原点を中心に `angle` だけ回転させます。

図 5.54 のプログラムがその例です。並べた二つの正方形について、右側を 30° だけ回転して描画します。描画結果は図 5.55(a) です。

3 次元の回転変換は以下の関数です。

`rotateX(angle)` ... x-y-z 座標系を x 軸を中心に `angle` だけ回転させます。

`rotateY(angle)` ... x-y-z 座標系を y 軸を中心に `angle` だけ回転させます。

`rotateZ(angle)` ... x-y-z 座標系を z 軸を中心に `angle` だけ回転させます[27]。

図 5.56 のプログラムが `rotateX()` を用いた例です。視点の前に並べた二つの立方体について、右側を 30° だけ x 軸の回りに回転して描画します。描画結果は図 5.57(a) です。図 5.56 のプログラムについて `rotateX(radians(30));` をコメントアウトして、`rotateY(radians(30));` をアンコメント（uncomment）[28]したプログラムを実行した結果が図 5.57(b) です。図 5.57(c) も同様です。

拡大縮小

各座標軸の縮尺を変えることで描画する被写体を拡大縮小できます。また軸毎に拡大率/縮小率を変えると、被写体の形状を変更できます。

2 次元の拡大縮小変換を行う関数は以下の通りです。

`scale(factor)` ... 2 次元座標系の X 軸および Y 軸を `factor` だけ拡大縮小します。

`scale(xfactor, yfactor)` ... 2 次元座標系の X 軸、Y 軸をそれぞれ `xfactor`、`yfactor` だけ拡大縮小します。

[27] 2 次元回転変換用の `rotate(angle)` は、3 次元座標系では `rotateZ()` と同じ効果を持ちます。
[28] コメントから元の実行文へ戻すことをこう呼びます。

図 5.54 のプログラムについて `rotate(radians(30));` をコメントアウトして、`scale(1.5,0.7);` をアンコメントしたプログラムを実行した結果が図 5.55(b) です。

3 次元の拡大縮小変換は以下の関数です。

`scale(factor)` ... 3 次元座標系の各軸を `factor` だけ拡大縮小します。

`scale(xfactor, yfactor, zfactor)` ... 3 次元座標系の x 軸、x 軸、z 軸をそれぞれ `xfactor`、`yfactor`、`zfactor` だけ拡大縮小します。

図 5.56 のプログラムについて `rotateX(radians(30));` をコメントアウトして、`scale(2.0,0.8,0.5);` をアンコメントしたプログラムを実行した結果が図 5.57(d) です。

せん断

被写体を軸の両方向に引っ張って形状をずらすような変形を**せん断**（shear、剪断）と呼びます。文章では理解しにくいので、図 5.54(c)(d)、図 5.56(e)(f) を参考にしてください。座標変換で言い換えると、2 次元 X-Y 座標系上で X 方向へのせん断変換は、c を定数として次のような変換行列を用いることに相当します。

$$\begin{pmatrix} X' \\ Y' \end{pmatrix} = \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} \tag{5.30}$$

p5.js の 2 次元のせん断変換を行う関数は以下の通りです。

`shearX(angle)` ... 2 次元 X-Y 座標系上で X 方向へのせん断変換を行います。引数 `angle` は被写体の傾きを角度で指定します。式 (5.30) で言えば、$c = \tan$ `angle` に相当します。

`shearY(angle)` ... 2 次元 X-Y 座標系上で Y 方向へのせん断変換を行います。引数 `angle` は被写体の傾きを角度で指定します。変換行列は式 (5.30) の X と Y を入れ替えたものです。

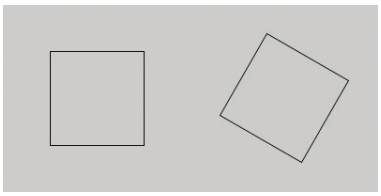
図 5.54 のプログラムについて `rotate(radians(30));` をコメントアウトして、`shearX(radians(30));` をアンコメントしたプログラムを実行した結果が図 5.55(c) です。`rotate(radians(30));` をコメントアウトして、`shearY(radians(45));` をアンコメントしたプログラムを実行した結果が図 5.55(d) です。

3 次元のせん断変換にも上の `shearX(angle)`、`shearY(angle)` を用います。図 5.56 のプログラムについて `rotateX(radians(30));` をコメントアウトして、`shearX(radians(45));` をアンコメントしたプログラムを実行した結果が図 5.57(e) です。`rotate(radians(30));` をコメントアウトして、`shearY(radians(45));` をアンコメントしたプログラムを実行した結果が図 5.55(f) です。

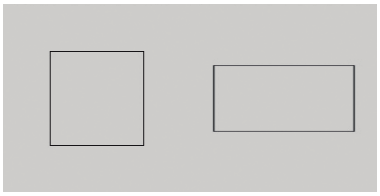
考察： 3 次元座標系における `shearY(angle)` に相当する 4 次元同次座標系での変換行列を示しなさい。

```
function setup() {
  createCanvas(400, 200);
  background(200);
  noFill();

  translate(100, 100);
  rect(-50, -50, 100, 100);

  translate(200, 0);
  rotate(radians(30)); // 回転
  //scale(1.5,0.7);       // 拡大縮小
  //shearX(radians(30));  // x 方向のせん断
  //shearY(radians(45));  // y 方向のせん断
  rect(-50, -50, 100, 100);
}
```

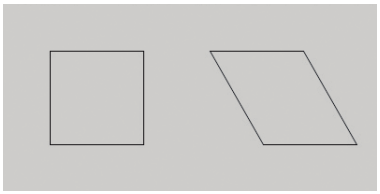
図 5.54: 正方形の様々な変形を調べるプログラム



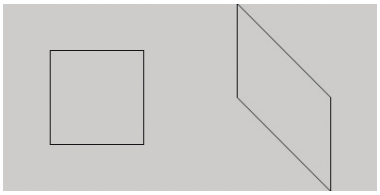
(a) `rotate(radians(30))`



(b) `scale(1.5,0.7)`



(c) `shearX(radians(30))`

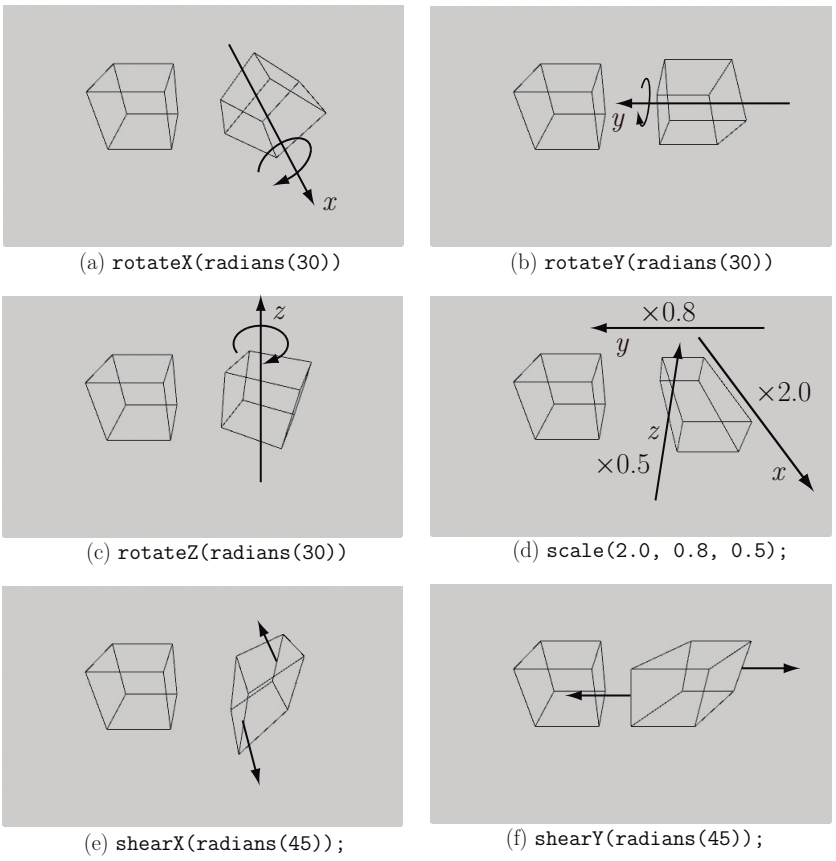


(d) `shearY(radians(45))`

図 5.55: 上のプログラムの実行結果

```
function setup() {
  createCanvas(500, 300, WEBGL);
  camera(400, 0, 300, 0, 0, 0, 0, 0, -1);
  background(200);
  noFill();

  translate(-180, 200, -50);
  box(200);

  translate(0, -400, 0);
  rotateX(radians(30));
  //rotateY(radians(30));
  //rotateZ(radians(30));
  //scale(2.0,0.8,0.5);
  //shearX(radians(45));
  //shearY(radians(45));
  box(200);
}
```

図 5.56: 立方体の様々な変形を調べるプログラム



(a) `rotateX(radians(30))`　(b) `rotateY(radians(30))`

(c) `rotateZ(radians(30))`　(d) `scale(2.0, 0.8, 0.5);`

(e) `shearX(radians(45));`　(f) `shearY(radians(45));`

図 5.57: 上のプログラムの実行結果

5.7.3　ロボットの描画

少し大きな描画例としてロボットを考えましょう。

ロボットの頭、胴体、左腕、右腕、左脚、右足を図 5.58 の平面図のように設計します。つまり、頭と胴体は直方体で作り、腕と足は球を組み合わせて作ることとします。

このロボットを描画するプログラムは図 5.60 の通りです。それぞれの部品はそれに対応する関数内で作ることとします。関数内では、その部品を座標原点の周りに配置するモデル座標系（あるいはローカル座標系）を用い、かつ関数内での座標変換が関数外（その関数を呼び出した関数）へ影響しないように `push()`、`pop()` で描画設定を退避/回復させます。作られた部品は最終的にロボットの形にまとめ上げられます。

図 5.61 は部品からロボットを作る工程の概念図です。モデル座標系上での変換操作およびモデル座標系からワールド座標系への変換操作をまとめて**モデリング変換**（modeling transformation）と呼びます。モデリング変換の内容は被写体毎にさまざまに異なるため、視野変換のように一律に述べることはできませんが、平行移動、回転、拡大・縮小、せん断などを順次適用し、目的の形状へ変形していくのが一般的です。また、通常、部品はより小さな部品の組み合わせで作られるため、モデル座標系は階層構造を成しているのが一般的です。下位のモデル座標系から上位のモデル座標系への座標変換も必要です。この変換もモデリング変換と呼びます[29]。

被写体がワールド座標系に構築されたならば、残る作業はそれを視野変換で視点座標系へ変換し、それを投影変換し、さらにキャンバス座標系へ変換し、ラスタライズすることです。これについては既に述べた通りです。結果、図 5.61 の最下部に示すような様々な 3D CG 画像を作ることができます。

[29] その意味では、最上位のモデル座標系をワールド座標系と理解することができます。

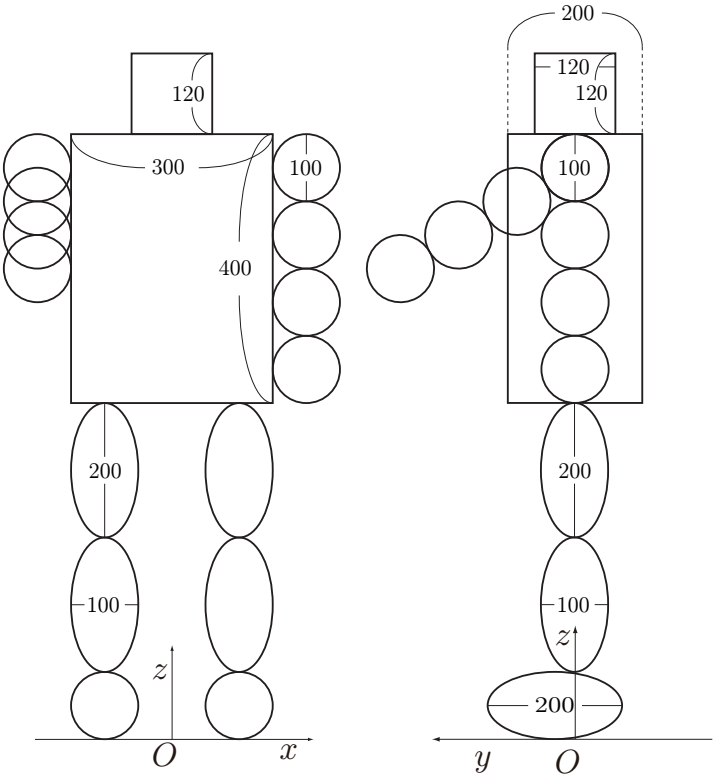


図 5.58: ロボットの設計図

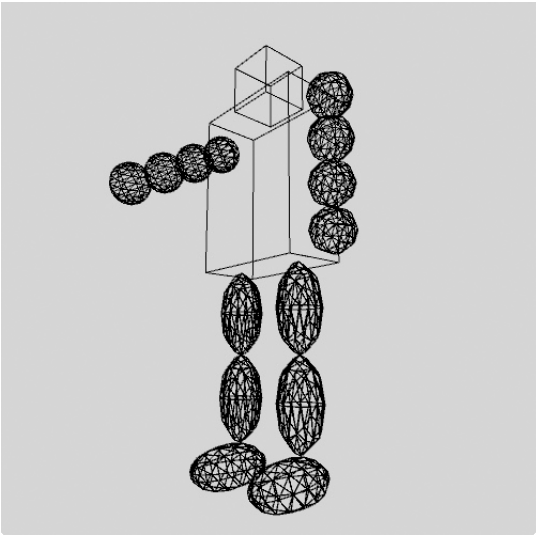


図 5.59: 図 5.60 のプログラムの実行結果

```
function setup() {
  createCanvas(500, 500, WEBGL);
  background(200);
  noFill();
  camera(700,600,300,
         0,0,500,
         0,0,-1);

  robot();
}

function robot() {
  push();

  leg(100);
  leg(-100);

  translate(0,0,700);
  body();

  translate(0,0,200);
  arm(200,0);
  translate(0,0,000);
  arm(-200,radians(60));

  translate(0, 0, 60);
  head();
  pop();
}

function head(){
  push();
  box(120);
  pop();
}

function body(){
  push();
  box(300,150,400);
  pop();
}

function arm(xposition, angle){
  print(angle);
  push();
  translate(xposition, 0, -50);
  rotateX(angle);
  sphere(50,16,8);

  translate(0, 0, -100);
  sphere(50,16,8);

  translate(0, 0, -100);
  sphere(50,16,8);

  translate(0, 0, -100);
  sphere(50,16,8);
  pop();
}

function leg(xposition) {
  push();
  translate(xposition, 30, 50);
  scale(1,2,1);
  sphere(50,16,8);

  scale(1,0.5,1);
  translate(0, -30, 150);
  scale(1,1,2);
  sphere(50,16,8);

  translate(0, 0, 100);
  sphere(50,16,8);
  pop();
}
```

図 5.60: 図 5.58 のロボットを描画するプログラム

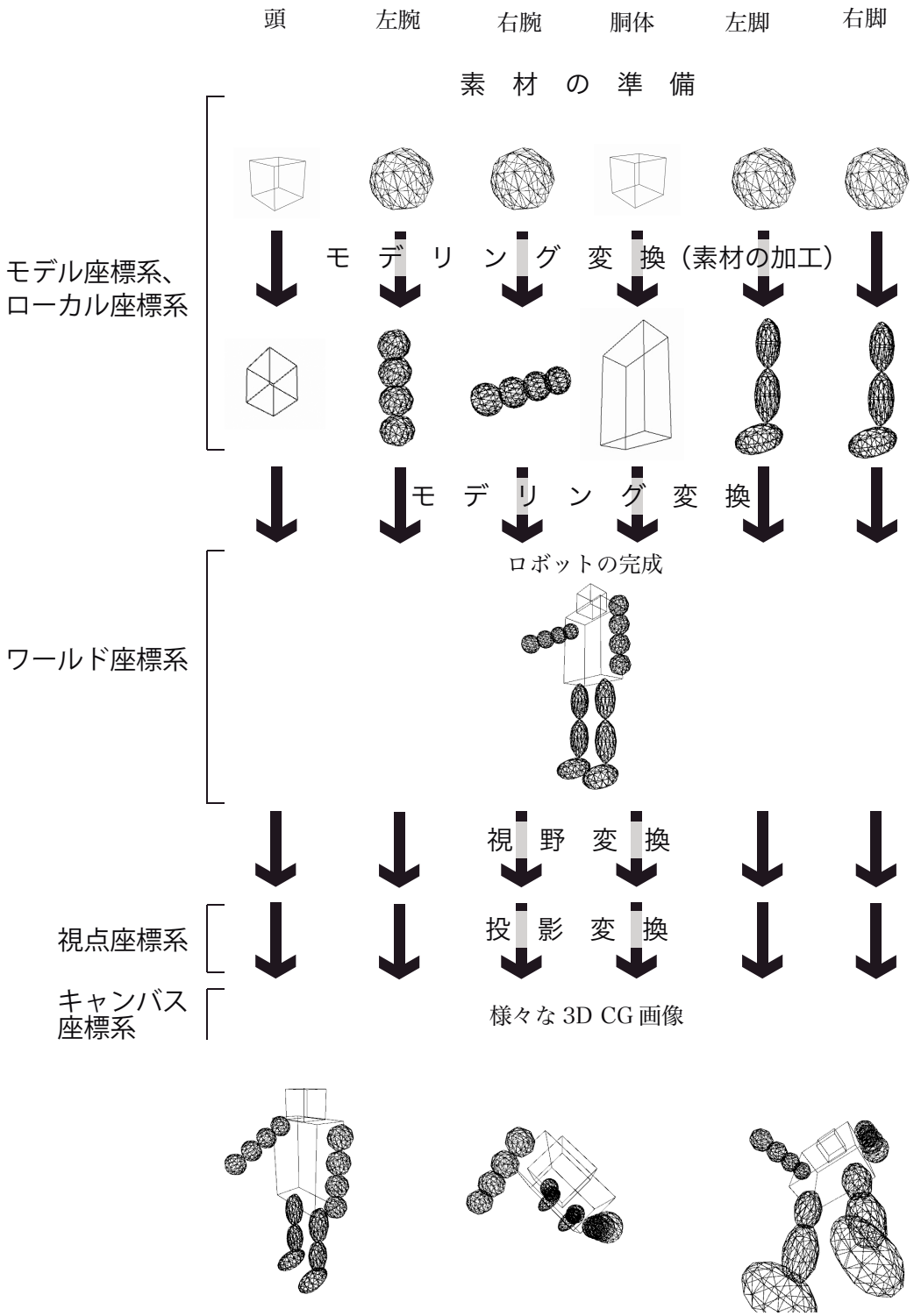


図 5.61: ロボットの構築（モデル座標系から視点座標系への変換）

第XI部

第6章　3次元グラフィックス：面画

前章では線画の作成を通して、3D CG に必要な座標変換を学んできました。この章からは線ではなく面で構成される 3D CG 画像の作成を検討していきます。

前章の図 5.39、図 5.60 のプログラムから `noFill();` を削除し、プログラムを実行すると、図 6.1、図 6.2 の画像を得ます。これらを作る上での課題は以下の二つです。

三角形のラスタライズ 3D CG で扱う多くの形状は多数の三角形を貼り合わせて表現できます[1]。四角形は 2 枚の三角形で表され、直方体は 6 枚の四角形で表され、球は緯線/経線で囲まれた四角形、三角形で近似的に表現できます。三角形を CG 画像として描画するにはそれをキャンバス上にラスタライズ（4.4 節参照）せねばなりません。

隠面消去 複数の三角形が重なっているときには視点に最も近い三角形のみを描画し、後方の三角形の重なって見えない部分は描画しないように制御すべきです。これを画素単位で行わねばなりません。

もうひとつの課題は、より写実的な 3D CG を作るための以下のものです。

陰影付け 三角形の見え方は、三角形が表す被写体の材質（金属、木材など）、三角形表面への光の当たり方（正面から、斜めから、光が当たっていないなど）、光源の色、位置、空気感などによって様々に変化し、均一ではありません。これを陰影（shade）と呼びますが、微妙な陰影が 3D CG にリアリティを与えます。それを光学的に考察し、数値計算で計算せねばなりません。

陰影付けによって、たとえば図 6.3、図 6.4 のような画像を作ることができます。

この章ではこれらを解決する手法を p5.js のプログラムを示しながら概説します。

注意：　被写体を構成する三角形のことを**三角形ポリゴン**（triangular polygon）、**三角ポリゴン**、あるい単に**ポリゴン**などとも呼びます。以下では、単なる三角形と区別するため、そのように呼ぶ場合があります。

[1] 例外として、髪、炎、波などの形状が不定形であるような被写体が挙げられます。これを描画するには別の技法は必要です。

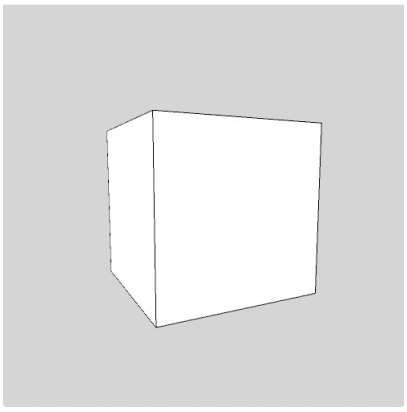


図 6.1: 面画の例（その 1）

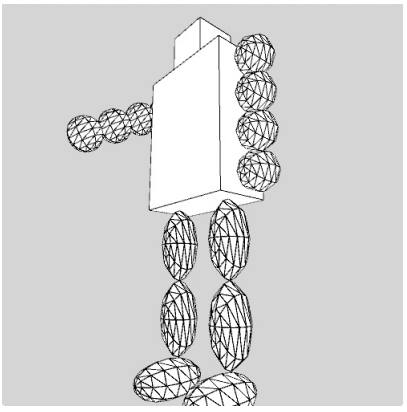


図 6.2: 面画の例（その 2）

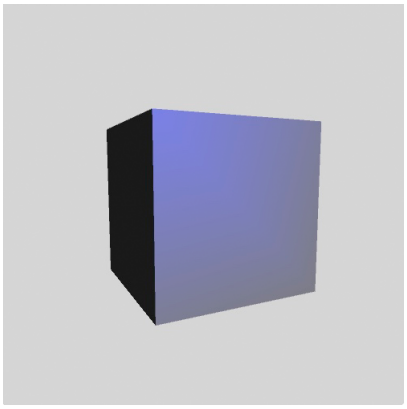


図 6.3: 面画の例（その 3）

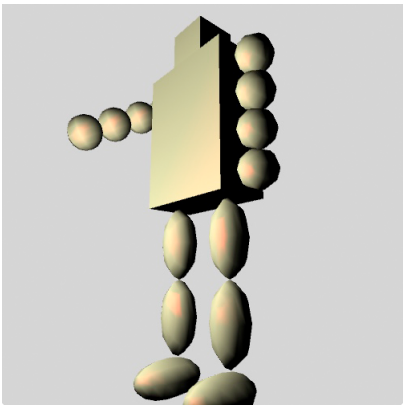


図 6.4: 面画の例（その 4）

6.1　三角形のラスタライズ

まず、次の事実に着目します。

- 3 次元座標系の中の任意の三角形は透視投影変換（式 (5.3)、式 (5.4)）によって投影面上の三角形に投影される。

3 次元座標系の中の任意の線分が投影面上で線分へ投影されることは 5.3.1 節で述べました。三角形は 3 本の線分から成りますから、結果、上の事実が成り立ちます。

そこで 1 枚の三角形ポリゴンをラスタライズする方法は以下の通りです。

1. 3 次元座標系の中の 3 頂点 a、b、c が任意に与えられたとします。
2. 前章の方法で、a、b、c をキャンバス上の画素位置 A、B、C へ座標変換（視野変換、投影変

```
function setup() {
  createCanvas(500, 500, WEBGL);
  background(200);
  camera(200, 300, 200, 0, 0, 0, 0, 0, -1);

  torus(120, 50);
  translate(-100, -100, 0);
  box(200);
}
```

図 6.5: 立方体とトーラスが重なった場合の描画プログラム

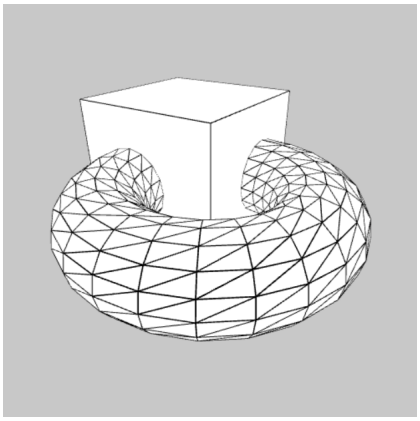


図 6.6: 左のプログラムの実行結果

換、キャンバス座標変換）を行います。特殊な場合（5.3.1 節末尾の考察を参照）を除いて、A、B、C は互いに異なる点に変換されます。

3. 4.4.4 節の方法で 3 点 A、B、C からなる三角形の内部の画素を全て求め、描画します。

GPU による高速描画は上の方法に基づいています。

6.2　隠面消去

図 6.5 と図 6.6 は立方体とトーラスの一部が同じ空間に重なった場合の描画です。図 6.1、図 6.2 と同様に、この場合も視点に近い面が描画され、視点から遠い面、見えない面は隠されています。p5.js には隠面消去、陰線消去が実装されているため、このような適切な描画ができる訳ですが、ここではその方法を解説します。

なお、トーラスは以下の関数で描画できます。

`torus(radius, tubeRadius, detailX, detailY)` ... 第 1 引数 `radius` に大円の半径を指定します。第 2 引数 `tubeRadius` に小円の半径を指定します。第 3 引数 `detailX` にトーラスの大円を分割する数を指定します。第 4 引数 `detailY` にトーラスの小円を分割する数を指定します。指定する引数は後ろから順に省略できて、全ての引数を省略した場合のデフォルト値は `radius = 50`、 `tubeRadius = 10`、`detailX = 24`、`detailY = 16` になるようです[2]。トーラスは x-y 平面に平行に描画されます。

今、視野変換後の視点座標系において図 6.7 のように、2 枚の三角形 T_1 と T_2 が視点から重なって見える配置にあると仮定します。正しく隠面消去を行う最も素朴な解決法は、複数の三角形につ

[2] `radius = 50` と `tubeRadius = 10` のデフォルト値についてホームページに記載がないため、別途調査しました。

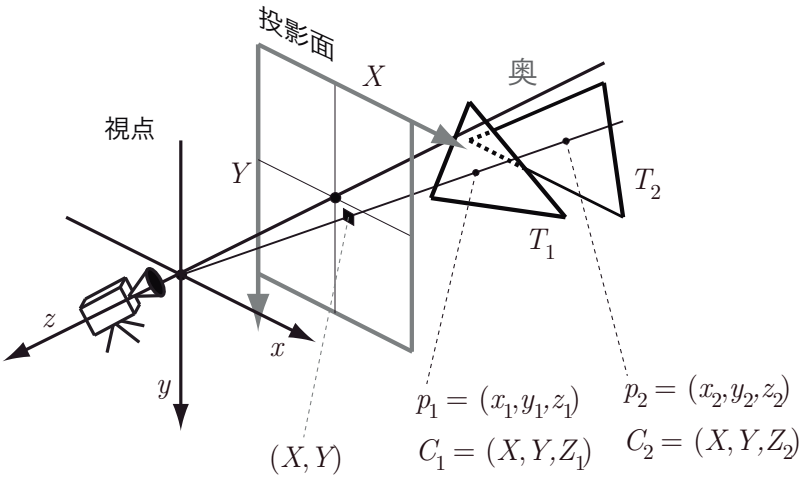


図 6.7: 視点座標系で 2 枚の三角形が重なっている場合

図 6.8: 三角形が相互に重なる場合

いて事前に前後関係を調べておき、後ろの三角形から順に描画していく方法です。図 6.7 の場合には T_2 を描画し、次に T_1 を描画します。これならば、後ろの三角形の描画像は、前の三角形の描画像で上書きされるため、本来見えない部分は消えて、正しい描画になりそうです。しかしこの方法は万全ではありません。図 6.8 のように三角形が互いに重なり合っている場合には、三角形の描画順序を決めることが原理的に不可能だからです[3]。

以下では、このような場合にも対応できる隠面消去の方法を解説します。この方法は現在、最も標準的な隠面消去の方法であり、GPU はこれをハードウェアで実装しています。

6.2.1　アイディア

図 6.7 について、三角形 T_1 上のある点 $p_1 = (x_1, y_1, z_1)$ が透視投影変換によって $C_1 = (X, Y, Z_1)$ へ変換され、T_2 上のある点 $p_2 = (x_2, y_2, z_2)$ が透視投影変換によって $C_2 = (X, Y, Z_2)$ へ変換され

[3] この問題を解決する方法としてスキャンライン法（scan line method）が知られていますが、この方法は結局、後に述べる Z バッファ法の特殊形と位置づけられますから、ここでは取り上げません。

たと仮定します。つまり、異なる 2 点 p_1、p_2 はキャンバス上の同じ画素点 (X, Y) へ投影されたと仮定します。Z_1、Z_2 は点 p_1、p_2 の正規化された深さ値とします（5.6.4 節参照）。

図 6.7 では T_1 の方が明らかに手前にありますから p_1 の点を画素点 (X, Y) へ色付けすべきです。この場合、「手前」とは、p_1 と p_2 の z 座標値 z_1 と z_2 を比較し、その値が大きい方を指します。z 座標値ではなく、正規化された深さ値 Z_1、Z_2 を用いても同じ結果になります。何故ならば、正規化された深さ値 $Z = -A - B/z$ は、$-\mathbf{far} \leq z \leq -\mathbf{near}$ の範囲では単調増加関数になるからです（図 5.38 参照）。しかも正規化された値は、描画される範囲では $-1 \leq Z \leq 1$ に限定されるため、様々な取り扱いが容易になるという利点[4]があります。5.6.4 節で正規化された値を導入した理由はこのためなのです。

もし別の三角形 T_3 の上の点 $p_3 = (x_3, y_3, z_3)$ が $C_3 = (X, Y, Z_3)$ へ透視投影されるならば、さらに比較が必要です。そこであらかじめ Z_1、Z_2 の中の大きい方の値 $Z_{\max}$ を**メモリに保存**しておき、$Z_{\max}$ と Z_3 を比較し、もし $Z_{\max} < T_3$ ならば画素点を T_3 の点で更新します。さもなくば画素点は更新しません。

6.2.2　Z バッファ法

上に述べたアイディアを特定の画素点ではなくキャンバス全体について実装するためには、正規化された深さ値をキャンバスの画素点数と同じ個数だけ保持するメモリ領域を用意します。この領域を一般に **Z バッファ**（Z buffer）[5]と呼び、Z バッファを用いる隠面消去法を **Z バッファ法**（Z buffer method）と呼んでいます[6]。この手法は単純明快です。欠点は、キャンバスと同等の大きさの記憶領域を必要とすることです。そのため、過去にメモリが高価であった時代（3.4 節末尾参照）には実装が難しい手法でした。

図 6.9 を用いて Z バッファ法をさらに詳しく紹介します。この図は、図 6.7 の 2 枚の三角形を非常に荒い解像度（10 × 10 画素）で描画する場合の過程を示したものです。

まず、キャンバス（画像）と同じ解像度の Z バッファを用意します（図 6.9(a)）。キャンバスの各画素点には背景色、Z バッファの各要素には正規化された深さ値の最小値 -1.0（あるいは -1.0 よりも小さな値）を初期設定しておきます。

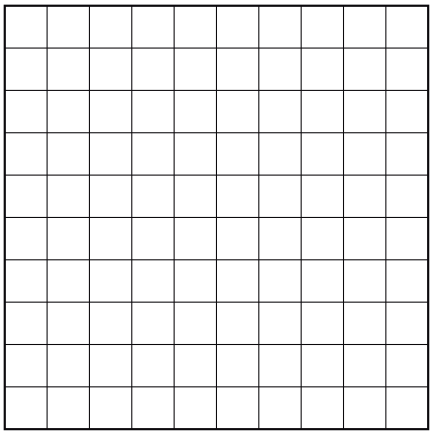
ここで、最初に三角形 T_1 を描画（ラスタライズ）すると仮定します。もし T_1 の深さ値が全て 0.3 であるならば、その深さ値は初期値 -1.0 よりも大きいので、T_1 の内部画素点は全て描画することができます。キャンバスに描画する際には対応する画素点の Z バッファの内容を 0.3 で更新します（図 6.9(b)）。

---

[4] 利点のひとつは、描画するか否か（クリッピングを行わないか行うか）の判断にいちいち **far**、**near** の値を参照する必要がなく、描画プログラムが簡潔になることです。また、詳細は述べませんが、深さ値の大小比較を浮動小数点数値で行う際に正規化しない場合に比べて高い精度で正しい比較が可能です。

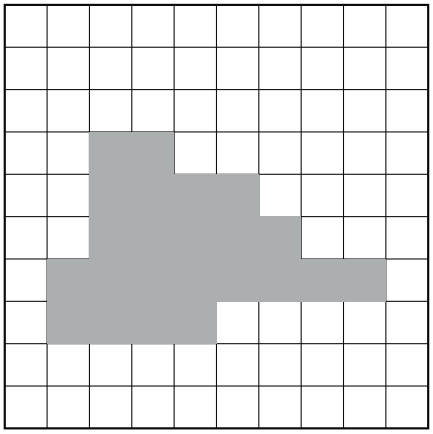
[5] Z の由来はもちろん深さ値の z または Z です。深さバッファ（depth buffer）とも呼びます。

[6] 深さバッファ法（depth buffer method）とも呼びます。



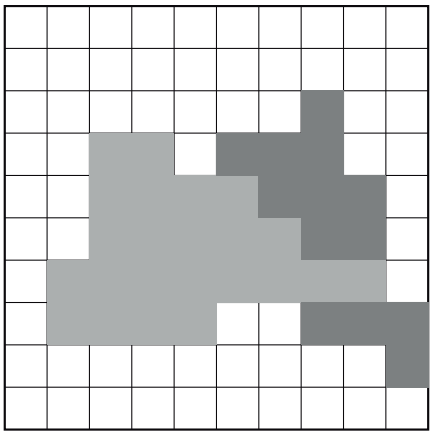
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0

(a) 初期状態のキャンバスと Z バッファの様子



-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	0.3	0.3	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	0.3	0.3	0.3	0.3	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	0.3	0.3	0.3	0.3	0.3	-1.0	-1.0	-1.0
-1.0	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	-1.0
-1.0	0.3	0.3	0.3	0.3	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0

(b) T_1 を描画した後のキャンバスと Z バッファの様子



-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0
-1.0	-1.0	0.3	0.3	-1.0	0.0	0.0	0.0	-1.0	-1.0
-1.0	-1.0	0.3	0.3	0.3	0.3	0.0	0.0	0.0	-1.0
-1.0	-1.0	0.3	0.3	0.3	0.3	0.3	0.0	0.0	-1.0
-1.0	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	-1.0
-1.0	0.3	0.3	0.3	0.3	-1.0	-1.0	0.0	0.0	0.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	0.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0

(c) T_2 を描画した後の Z バッファの様子

図 6.9: 図 6.7 の 2 枚の三角形を描画する際のキャンバスと Z バッファの変化

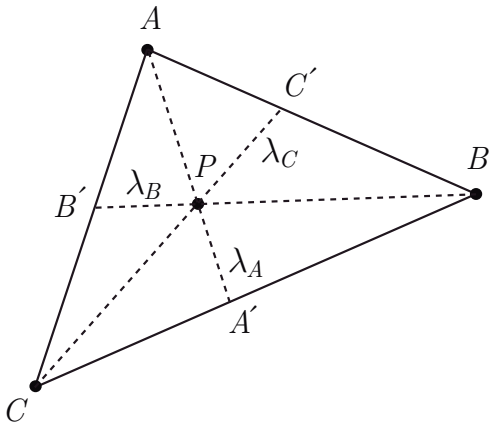


図 6.10: 重心座標系による正規化された深さ値の線形補間

次に T_2 を描画すると仮定します。もし T_2 の深さ値が 0.0 ならば、バッファに 0.3 よりも小さな値が格納されている画素位置に限り、対応する画素点を T_2 で描画することが可能です。描画の際にはバッファを 0.0 で更新します。もしバッファに 0.0 以上の値が格納されているならば、深さ値 0.0 を持つ T_2 は描画しません（図 6.9(c)）。

6.2.3　重心座標系による線形補間

上述の Z バッファ法の例題では三角形 T_1 の正規化された深さ値は全ての内部画素点について一様に 0.3、T_2 のそれは 0.0 と仮定しました。しかし一般に、三角形が投影面に平行でない場合には深さ値は一様ではありません。

標準的な 3D CG のアルゴリズムでは、それを三角形を構成する 3 頂点から線形補間で求めています。実はこの線形補間は深さ値だけではなく、三角形ポリゴンの描画に関する様々な計算に利用されています。

図 6.10 で解説します。まず、キャンバス上の三角形の 3 頂点を A、B、C とし、三角形内の任意の点を P とします。このとき、A と P を通る直線が A の対辺 BC と交わる点 A' を求めます。同様に、交点 B'、C' を図 6.10 の通りに求めます。そして三つの値 λ_A、λ_B、λ_C を以下のように計算します。

$$\lambda_A = \frac{|P - A'|}{|A - A'|}, \quad \lambda_B = \frac{|P - B'|}{|B - B'|}, \quad \lambda_C = \frac{|P - C'|}{|C - C'|} \tag{6.1}$$

この三つの値の組 $(\lambda_1, \lambda_2, \lambda_3)$ は**重心座標系**（barycentric coordinate system）[7]における重心座標値として知られており、任意の座標値 A、B、C、P について以下が成り立ちます。

$$P = \lambda_A \cdot A + \lambda_B \cdot B + \lambda_C \cdot C \tag{6.2}$$

[7] 面積座標系（area coordinate system）とも呼びます。

つまり、点 P は 3 頂点 A、B、C を重み λ_A、λ_B、λ_C で加重平均した値になっており[8]、これが「重心座標系」の名前の由来になっています。

考察（少し煩雑）： (1) 任意の P について $\lambda_A + \lambda_B + \lambda_C = 1$ が成り立つことを証明しなさい。(2) 任意の P について式 (6.2) を証明しなさい。

重心座標系の利点は、3 点 A、B、C に付加された様々な値（スカラー値、ベクトル値）$\mathcal{V}_A$、$\mathcal{V}_B$、$\mathcal{V}_C$ について点 P での線形補間値 $\mathcal{V}_P$ を

$$\mathcal{V}_P = \lambda_A \cdot \mathcal{V}_A + \lambda_B \cdot \mathcal{V}_B + \lambda_C \cdot \mathcal{V}_C \tag{6.3}$$

と定義できることです。これは 3D CG における最も一般的な補間式です。GPU にはこの線形補間を高速計算するハードウェアが搭載されています。

正規化された深さ値についても上の式が利用できます。3 頂点 A、B、C の正規化された深さ値を Z_A、Z_B、Z_C とするとき、点 P の深さ値 Z_P を以下の式で計算します。

$$Z_P = \lambda_A Z_A + \lambda_B Z_B + \lambda_C Z_C$$

三角形ポリゴンの隠面消去では、各画素点の深さ値をこの式で計算し、Z バッファ法に用いています。

考察： (1) もし $P = A$ ならば、$Z_P = Z_A$ が成り立つこと、(2) もし P が線分 BC 上ならば、$Z_P = \lambda_B Z_B + \lambda_C Z_C$ が成り立つことを確認しなさい。

[8] 別の視点では、点 A、B、C にそれぞれ λ_A、λ_B、λ_C の重量があるとき、この三角形の重心は点 P になると考えることもできます。

第XII部

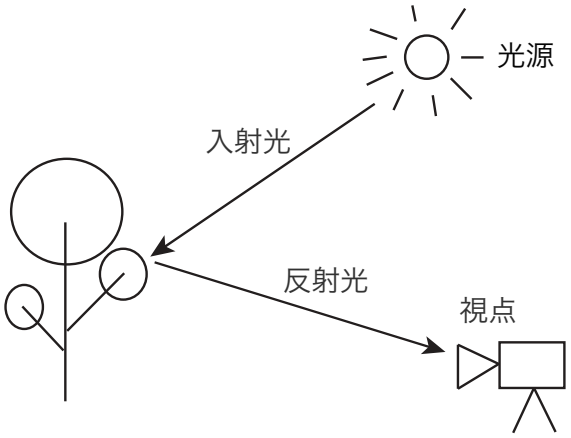


図 6.11: 陰影計算のモデル

図 6.12: 陰影の例（ウィキペディア「マルクス・ユニウス・ブルトゥス」（日本語版）の図から転載）

6.3 陰影付け

私たちが見る様々な物体は、日光や蛍光灯などの光源から発射された光が被写体の表面で反射して目に入り、画像として認識されています（図 6.11 参照）。そのとき表面の輝度は光が物体表面に当たる角度によって微妙に変化します。たとえば図 6.12 の胸像彫刻では彫刻表面の光の当たり方によって微妙な陰影が作られています。この種の陰影付けをコンピュータでシミュレートし、リアルな画像を得ることがこの節の目的です。

現在の標準的な 3D CG は以下の 3 種類の反射光を用いる光学モデルを基礎としており、このテキストでもそれを順に解説していきます。これは 1970 年代にベトナム人研究者フォン（Phong）によって提案されたため、**フォンの反射モデル**（Phong reflection model）[9]と呼ばれています。

拡散反射光 ... 図 6.12 の胸像彫刻の陰影が主にこれに相当します。

環境光 ... ぼんやりとした空気感を演出します。

鏡面反射光 ... メタリックな輝きがこれに相当します。

また、この 3 種類の反射光に加えて、p5.js では以下も表現できます。

自発光 ... 被写体そのものが光を放つ場合に用います。

ここでは p5.js におけるプログラミングと共にこれらを解説していきます。

[9] 類似した術語で「フォン・シェーディング」があり、「フォンの反射モデル」と同じ意味で用いられることもあります。しかし厳密に言えば、フォン・シェーディング（後述）は反射光計算の補間手法を指す術語であって、フォンの反射モデルとは異なります。反射モデルと補間法をまとめてフォン・シェーディングと呼ぶこともあります。

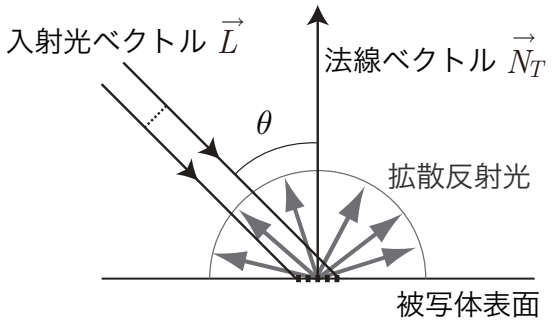


図 6.13: 拡散反射モデル

6.3.1 拡散反射

拡散反射（diffuse reflection）とは、光源から発せられ、被写体表面に入射した光のエネルギーがその点から均一に反射される現象のことです。面が滑らかではなく、ざらついた材質であるときに現れ、乱反射と呼ばれることもあります。完全に均一に拡散反射する材質は現実には存在しません[10]が、ここでは理想化した拡散反射面をモデルとして考えています。たとえば木材表面やツヤ消しの塗料面などがそれに近いとされています。

被写体表面（の単位面積）に入射する光のエネルギーは光の入射角度によって変化します。これは、夏は太陽が高いために暑く、冬は太陽が低いために寒い事実と似た現象です。反射光の輝度は入射されるエネルギーに比例し、それが被写体表面の陰影となって現れます。

一般に、面の法線ベクトルと光線ベクトルのなす角度が θ であるとき、反射光の輝度は光が正面から当たる場合（$\theta = 0$）の $\cos\theta$ 倍になります（図 6.13 参照）。つまり物体表面からの反射光の輝度値 I_d は、入射光の光強度ベクトルを $\vec{L}$ として、

$$I_d \propto |\vec{L}| \cdot \cos\theta$$

と表すことができます。これを**ランベルトの余弦則**（Lambert's cosine law）と呼びます[11]。あるいは、比例定数 K_d (> 0) を導入し、

$$I_d = K_d \cdot |\vec{L}| \cdot \cos\theta$$

ここに K_d の値は厳密には物理的に決定すべきものですが、3D CG では画像の見映えを考慮してプログラマ（あるいは CG デザイナー）が調整するのが一般的です。余弦値 $\cos\theta$ は、三角形表面の法線ベクトル を $\vec{N_T}$ とするとき、$-(\vec{L}/|\vec{L}|) \cdot \vec{N_T}$ に比例しますから、通常、輝度値の計算には

[10] ほぼ理想的な拡散反射特性を持つ素材として、スペクトラロン（spectralon）が現実に開発されています。入射した光線をほぼ均一に反射することから、様々な光源（たとえば LED）の強度を高精度で計測するためなどの応用に利用されています。

[11] ランベルトは 18 世紀のドイツ人の物理学者です。各国語による Lambert の発音の違いからランバートの余弦則と呼ばれることもあります。

```
function setup() {
  createCanvas(500,500,WEBGL);
  background(200);
  camera(200,150,0, 0,0,0, 0,0,-1);
  noStroke();

  //拡散反射に関するマテリアルと光源の設定
  fill(255,0,0);
  directionalLight(200,200,200, 0,-1,-1);

  sphere(100,100,100);
}
```

図 6.14: 球の拡散反射光を描画するプログラム

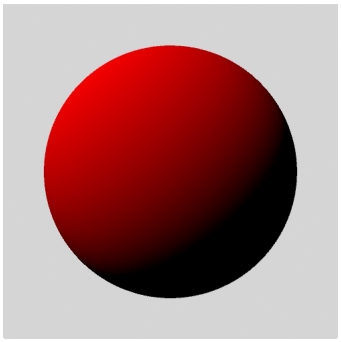


図 6.15: 左のプログラムの実行結果

三角関数を用いず、内積を用いて以下のように計算します。

$$I_d = -K_d \cdot \left(\vec{L} \cdot \vec{N_T}\right) \tag{6.4}$$

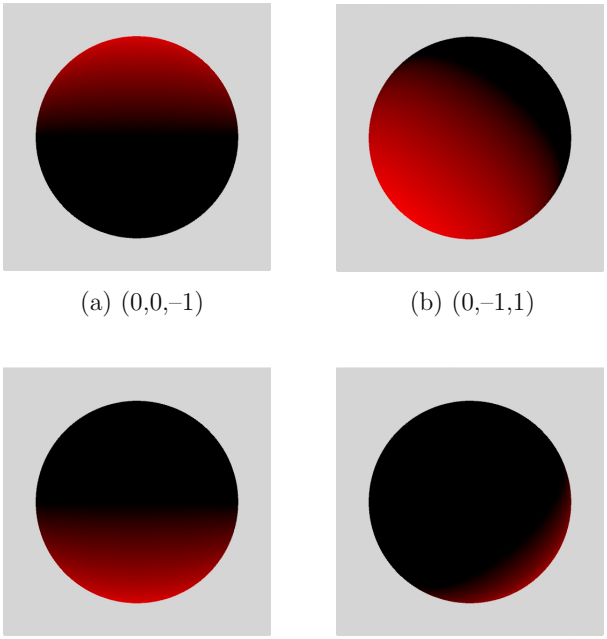
なお、上の式で計算された輝度値 I_d が負値のときには光線はその面の裏側に当たっていることを意味しますから、その場合の輝度値は 0 に置き換える必要があります。

図 6.14 は球の表面で拡散反射する光を描画する p5.js のプログラム、図 6.15 はその実行結果です。球面は多数の三角形で構成されており、三角形毎に法線ベクトルが異なります。結果、球の部分毎に異なる拡散反射が生じます。拡散反射には次の関数を用います。

`fill(mr, mg, mb)` ... 3D CG では被写体表面のことをしばしば**マテリアル**（material）と呼ぶますが、この関数は拡散反射に関するマテリアルの色（あるいは色の反射率）を (`mr`, `mg`, `mb`) で指定します。同名の関数はすでに 4 章で紹介しましたが、同じ関数が 3D CG ではマテリアルの色の定義に利用されている訳です。プログラム中でこの関数を呼び出さない場合にはマテリアルの色は白色がデフォルト値になる点も 4 章と同じです。

`directionalLight(lr, lg, lb, nx, ny, nz)` ... 3 次元空間内に降り注ぐ平行光線を指定します。(`lr`, `lg`, `lb`) が光線の色です。(`nx`, `ny`, `nz`) が光線の進行方向ベクトルです。このベクトルは単位ベクトルでなくても構いませんが、輝度計算時にはベクトルの大きさは無視されます。その点で式 (6.4) とは異なる計算になります。**後に述べる拡散反射および鏡面反射はこの関数または類似の `pointLight()`（点光源、後述します）を呼び出さない限り発動しない仕様になっています。陰影付けを行いたい場合には必ず `directionalLigh()` または `pointLight()` を呼び出してください。**

描画される色 (r, g, b) は、`fill()` で指定するマテリアルの色と `directionalLight()` で指定す



(a) (0,0,–1)　(b) (0,–1,1)

(c) (0,0,1)　(d) (0,1,1)

図 6.16: 光線の方向を変えた場合の描画

る光源の色の積として以下のように計算されます。

$$(r, g, b) = 255 \times \left(\frac{\texttt{mr}}{255}\frac{\texttt{lr}}{255}, \frac{\texttt{mg}}{255}\frac{\texttt{lg}}{255}, \frac{\texttt{mb}}{255}\frac{\texttt{lb}}{255} \right) \tag{6.5}$$

図 6.14 では、$(\texttt{mr}, \texttt{mg}, \texttt{mb}) = (255, 0, 0)$、$(\texttt{lr}, \texttt{lg}, \texttt{lb}) = (200, 200, 200)$ ですから、$(r, g, b) = (200, 0, 0)$ です。光線の方向は $(\texttt{nx}, \texttt{ny}, \texttt{nz}) = (0, -1, -1)$ ですから、光はカメラの左上後方から降り注ぐような位置関係です。

図 6.16 には、光線の方向を様々に変えた（図 6.14 のプログラムの `0,-1,-1` を様々に変えた）画像を載せます。図の上方が z 軸の正の方向、図に向かって右方向が x 軸の正の方向、左方向が y 軸の正の方向です。

補足：　一般に、拡散反射光の色は入射光の色ではなく、被写体表面の固有の色と考えられています。何故ならば、拡散反射では入射光のエネルギーが被写体の表面の分子層および内部のごく浅い層に一時的に蓄えられ、被写体の特性（分子構造）に応じて特定の波長が吸収された後に反射（放射）されるからです。その意味では p5.js の式 (6.5) の計算式は自由すぎますが、p5.js は物理法則の枠を超えてより自由な表現を許しているのでしょう。

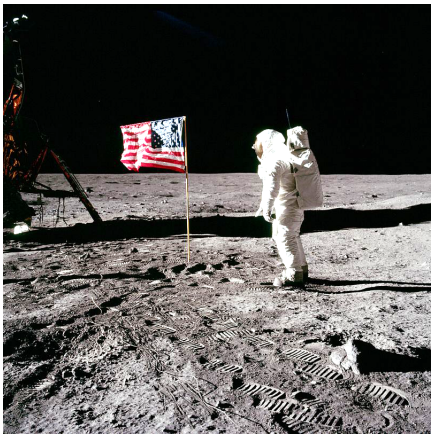


図 6.17: 大気による乱反射がない例（ウィキペディア「アポロ計画」（日本語版）の図から転載）

6.3.2 環境反射

物体表面が光源から陰になっていて、光源からの直接光が当たらない場所（たとえば図 6.15、図 6.16 の黒い領域）であっても、通常、その面は真っ黒にはならず、微かに明るいことを私たちは知っています。近くの壁に当たった光が反射し、間接光（2 次光）となって陰の面を照らすことがあるからです。また、地球上には大気（空気）があるため、光線が大気中で複雑に乱反射することで、陰の部分であっても通常、ある一定の明るさは持っています。それが画像に一種の空気感を与えています。もし大気がないならば、日陰の部分はちょうど図 6.17 の月面の写真のようになり[12]、私たちの日常の感覚とは異なります。

被写体表面が直接光以外の光を反射することを**環境反射**（ambient reflection）と呼んでいます。環境反射をコンピュータシミュレーションすることで、空気感のある自然な印象の 3D CG 画像を得ることができます。図 6.15 の画像を思い出しましょう。直接光しか考慮していないため、陰の面は真っ黒に描画されています。

環境反射の精密なコンピュータシミュレーションは容易ではないことが知られています。様々な壁に複雑に反射を繰り返す膨大な光線をひとつひとつ取り扱うには複雑なアルゴリズム[13]と膨大な計算量を必要とするからです[14]。しかし幸いなことに、私たちの日常での空気による乱反射については、日なた、日陰に関わらず、全ての被写体表面に一定の輝度を付加することで、擬似的な環境反射を表現できます。

図 6.18 は、図 6.14 のプログラムに一定の輝度の環境反射を付加するプログラムです。図 6.19 は、その実行結果です。拡散反射光に環境光が付加されています。図 6.15 において黒色だった部分に

[12] ただし、宇宙服の陰の部分には地表からの照り返しがあるため、真っ黒ではありません。
[13] 興味のある人は、たとえば「パストレーシング」でネット検索してください。
[14] それを実現する手法としてラジオシティ（Radiosity）法が知られています。コンピュータの演算能力の向上によってそれを実装する事例が増えてきています。

```
function setup() {
  createCanvas(500,500,WEBGL);
  background(200);
  camera(200,150,0, 0,0,0, 0,0,-1);
  noStroke();

  //環境反射に関するマテリアルと光源の設定
  ambientMaterial(255,0,0);
  ambientLight(60,60,60);

  fill(255,0,0);
  directionalLight(200,200,200, 0,-1,-1);

  sphere(100,100,100);
}
```

図 6.18: 環境光を加味して描画するプログラム

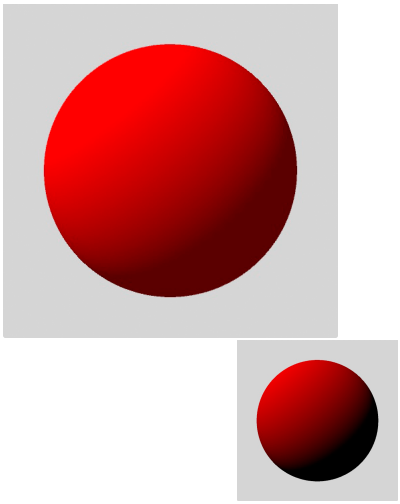


図 6.19: 左のプログラムの実行結果（比較のため、右下に図 6.15 を再掲）

赤い色が薄く付いており、明るい部屋の置いた赤いボールのように見えないでしょうか。ある種の空気感が表現できている訳です。

環境反射光（簡単に環境光）は以下の二つの関数で指定します。

`ambientMaterial(mr, mg, mb)` ... 環境反射に関するマテリアルの色を $(\mathtt{mr}, \mathtt{mg}, \mathtt{mb})$ で指定します。プログラム中でこの関数を呼び出さない場合にはマテリアルの色は白色がデフォルト値になります。

`ambientLight(ar, ag, ab)` ... 一定輝度の環境光の色を $(\mathtt{ar}, \mathtt{ag}, \mathtt{ab})$ で指定します。**環境反射はこの関数を呼び出さない限り発動しません。**

実際に描画される色 (r, g, b) は、拡散反射と同様に、`ambient()` で指定したマテリアルの色と `ambientLight()` で指定する環境光の色の積として以下のように計算されます。

$$(r, g, b) = 255 \left(\frac{\mathtt{mr}}{255}\frac{\mathtt{ar}}{255}, \frac{\mathtt{mg}}{255}\frac{\mathtt{ag}}{255}, \frac{\mathtt{mb}}{255}\frac{\mathtt{ab}}{255} \right) \tag{6.6}$$

補足： 環境反射光は、拡散反射光と同じ原理で被写体固有の色です。その意味では p5.js の式 (6.6) の計算式も自由すぎるのでしょうが、... 以下同文。

6.3.3 鏡面反射

鏡は入射光線を特定の方向にしか反射しません。光沢のある表面もそれに似た性質を持ち、入射光線を特定の方向およびその周辺へ特に強く反射します。これが被写体表面にメタリックな光沢感

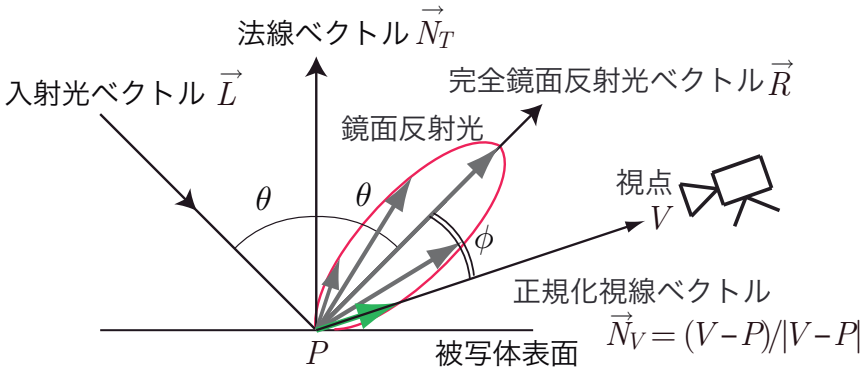


図 6.20: 鏡面反射モデル

を与えます。鏡による光線の反射を文字通り **鏡面反射** （specular reflection）と呼びますが、3D CG では光沢のある面による反射も広い味で鏡面反射と呼んでいます。

既に学んだ拡散反射光は陰影による立体感を与えますが、その質感はツヤ消しであって、光沢のある輝きではありません。しかし多くの物質の反射は拡散反射と鏡面反射の両方の特性を兼ね備えており[15]、両者による光強度の比率を調整することで様々な質感を表現可能です。

フォンの鏡面反射モデルを図 6.20 で解説します。まず、被写体表面が鏡（完全な鏡面）である場合、図の入射光ベクトル $\vec{L}$ は表面の入射点 P において完全鏡面反射光ベクトル $\vec{R}$ として反射します。ここに $\vec{R}$ は以下の式で計算できます。

$$\vec{R} = \vec{L} - 2\left(\vec{L} \cdot \vec{N_T}\right)\vec{N_T} \tag{6.7}$$

ベクトル $\vec{L}$ と $\vec{N_T}$ のなす角度と $\vec{R}$ と $\vec{N_T}$ のなす角度は同じであることに注意してください。

考察　式 (6.7) を証明しなさい。

鏡の場合、光は厳密に $\vec{R}$ の方向にのみ反射しますが、鏡ではない物質の場合には、$\vec{R}$ からのズレ量に応じて反射光が減少します。そこでフォンの反射モデルでは、入射点 P から視点 V への正規化視線ベクトル（normalized view vector）を

$$\vec{N_V} = \frac{V-P}{|V-P|}$$

と与えるとき、視点への反射光の輝度値 I_s を以下のように、入射光の強さ $|\vec{L}|$ および $\vec{N_V}$ と $\vec{R}$ の余弦値 $\cos\phi$ の累乗に比例すると考えます。

$$I_s \propto |\vec{L}| \cdot \cos^{\gamma}\phi$$

[15] より正確に言えば、両者の特性を兼ね備えていると見なして近似的に CG 描画を行なっています。

```
function setup() {
  createCanvas(500,500,WEBGL);
  background(200);
  camera(200,150,0, 0,0,0, 0,0,-1);
  noStroke();

  ambientMaterial(255,0,0);
  ambientLight(60,60,60);

  //鏡面反射に関するマテリアルと光源の設定
  specularMaterial(255,255,255);
  specularColor(150,150,150);
  shininess(64.0);

  fill(255,0,0);
  directionalLight(200,200,200, 0,-1,-1);

  sphere(100,100,100);
}
```

図 6.21: 鏡面反射光を加味して描画するプログラム

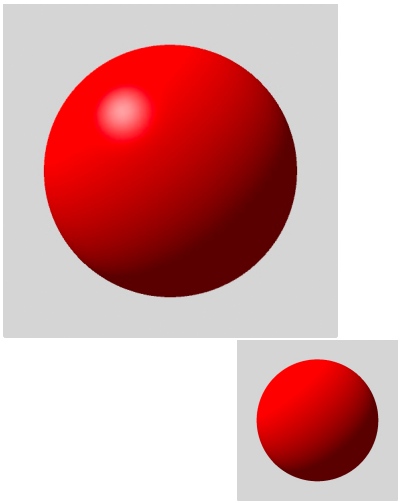


図 6.22: 左のプログラムの実行結果（比較のため、右下に図 6.19 を再掲）

あるいは比例定数 K_s (> 0) を導入して、

$$I_s = K_s \cdot |\vec{L}| \cdot \cos^{\gamma} \phi \tag{6.8}$$

ここに K_s は K_d と同様に画像の見映えを考慮してプログラマ（あるいは CG デザイナー）が調整するのが一般的です。正定数 γ はフォン指数（Phong exponent）と呼ばれる材質の滑らかさ、光沢感を表す定数です。ここに余弦値は以下の式で計算できます。

$$\cos \phi = \frac{\vec{R}}{|\vec{R}|} \cdot \vec{N_V}$$

拡散反射の場合と同様に、この余弦値が負値のときには輝度値は 0 で置き換えます。

図 6.21 は、図 6.18 のプログラムに鏡面反射を付加するプログラムです。図 6.22 はその実行結果です。描画された球の左上に白い反射光が見えます。反射光が入ったことで、球がツルツルした質感に変わっています。図 6.23 には、図 6.22 の拡散反射、環境反射、鏡面反射のそれぞれの描画像を示します。

鏡面反射には次の関数を用います。

`specularMaterial(mr, mg, mb)` ... 鏡面反射に関するマテリアルの色を (`mr`, `mg`, `mb`) で指定します。プログラム中でこの関数を呼び出さない場合にはマテリアルの色は $(128, 128, 128)$ がデフォルト値になります。

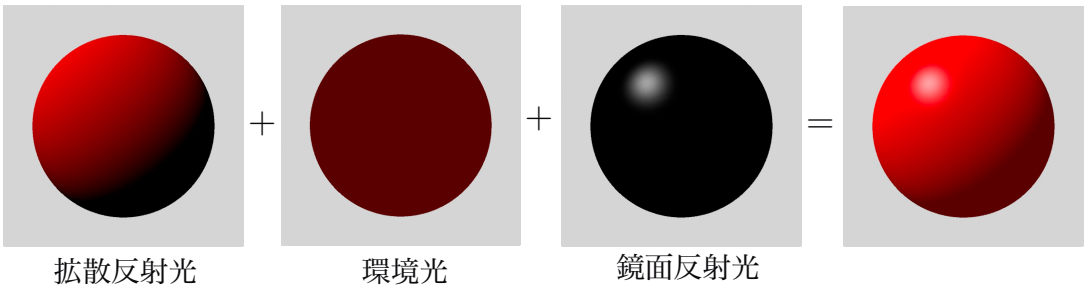


図 6.23: それぞれの反射光の画像と合成像

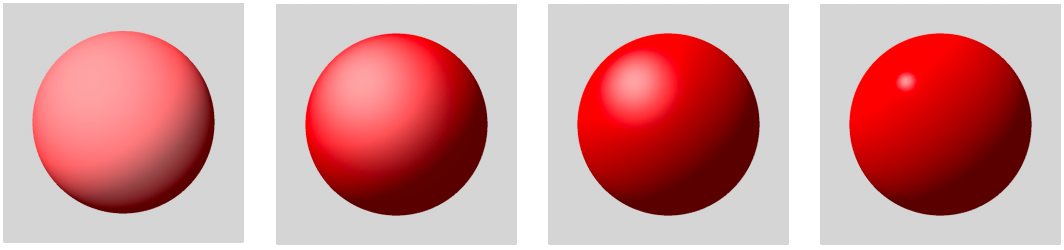


図 6.24: フォン指数 γ による鏡面反射の変化（(d) 64.0 に相当する画像は図 6.21 あるいは図 6.23 最右を参照）

`specularColor(sr, sg, sb)` ... 鏡面反射光の色を $(\mathtt{sr}, \mathtt{sg}, \mathtt{sb})$ で指定します。**鏡面反射は `directionalLight()` または `pointLight()` を呼び出したときに自動的に発動しますが、**この関数を呼び出さない場合には引数として $(0, 0, 0)$ が与えられた場合がデフォルトで仮定されるため、鏡面反射の効果は実際には表れません。光線の方向は、`directionalLight()`、`pointLight()` の指定値が用いられます。

`shininess(shine)` ... 式 (6.8) のフォン指数 γ の値を指定します。材質のメタリック感を決定する定数です。効果が分かるように、図 6.24 に `shine` $= 1,\ 4,\ 16,\ 256$ の場合の画像（`shine` $= 64$ の場合の画像は図 6.21 あるいは図 6.23 最右を参照）を示します。値が上がるほどにメタリック感が増しますが、値を大きくしすぎると効果は限定的になります。

注意： プログラム中では `specularColor()` は `directionalLight()` の前に呼び出してください。つまり以下の呼び出し順序でプログラムを作成してください。

```
...
specularColor(...)     // 正しい順序
...
directionalLight(...) // 正しい順序
...
sphere() // 被写体の描画
```

以下の順序：

```
  ...
directionalLight(...) // 間違った順序
...
specularColor(...)    // 間違った順序
...
sphere() // 被写体の描画
```

では鏡面反射が意図した通りに描画されません[16]。この理由は、p5.js では拡散反射と鏡面反射に関する設定を、拡散反射の光源を設定する関数（`directionalLight()` やすぐ後に紹介する `pointLight()`）が統括するような仕様になっているためと思われます。これを考慮してこの節の全てのプログラムでは反射光の設定を以下の順で統一しました。

1. 環境反射
2. 鏡面反射
3. 拡散反射

鏡面反射で実際に描画される色 (r, g, b) は、拡散反射、環境反射と同様に、`specularMaterial()` で指定するマテリアルの色と `specularColor()` で指定する光源の色の積として以下のように計算されます。

$$(r, g, b) = 255 \left(\frac{\texttt{mr}}{255} \frac{\texttt{sr}}{255}, \ \frac{\texttt{mg}}{255} \frac{\texttt{sg}}{255}, \ \frac{\texttt{mb}}{255} \frac{\texttt{sb}}{255} \right) \tag{6.9}$$

補足： 一般に、拡散反射光、環境反射光と異なり、鏡面反射光の色は入射光の色です。光が成分を変えず、そのまま反射されるからです。その意味では p5.js の式 (6.9) の計算式も自由すぎるのでしょうが、... 以下同文。

6.3.4 自発光

以下の関数を用いれば被写体自らが発光し、光を放射する場合をシミュレートできます。

`emissiveMaterial(r, g, b)` ... 被写体の描画像に RGB 輝度値 $(\texttt{r}, \texttt{g}, \texttt{b})$ を加算します。

しかしこれを用いても単に被写体がベタ塗りの描画像になるため、この講義では解説しないこととします。自発光被写体も取り扱えるとのみ紹介しておきます。

[16] 現在、調べた範囲ではその他の設定関数どうしの呼び出し順序には制約はないようです。

第XIII部

```
function setup() {
  createCanvas(600,500,WEBGL);
  background(200);
  camera(200,0,0, 0,0,0, 0,0,-1);
  noStroke();

  fill(200,200,100);
  pointLight(255,255,255,80,0,0);
  //directionalLight(255,255,255, 0,-1,-1);

  translate(0,60,0);
  sphere(50,100,100);
  translate(0,-120,0);
  sphere(50,100,100);
}
```

図 6.25: 点光源または平行光線で球を描画するプログラム

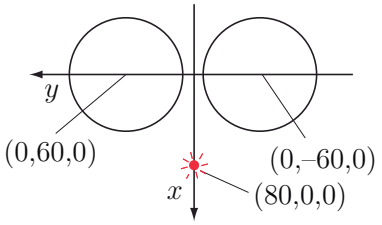


図 6.26: 球と点光源の位置関係

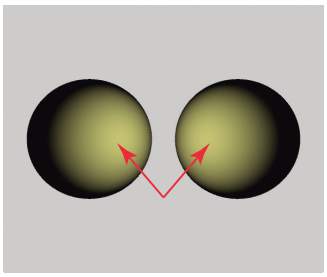


図 6.27: 左のプログラムの実行結果：点光源の場合

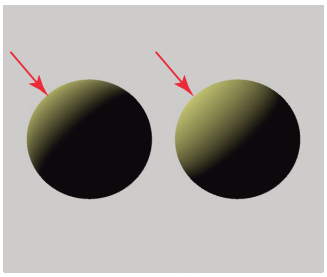


図 6.28: 左上のプログラムの実行結果：平行光線の場合

6.3.5 様々な光源

p5.js では点光源も扱うことができます。また、複数の光源を設定し、様々な方向から被写体を照らすこともできます。これらは上に紹介した拡散/環境/鏡面反射の応用にすぎませんが、より印象的な映像を作ることができます。

点光源

p5.js では平行光線の代わりに点光源も設定可能です。設定関数は以下です。これを平行光線を設定する `directionalLight()` の代わりに使用できます。

`pointLight(r, g, b, x, y, z)` ... (r, g, b) で点光源の色、輝度を RGB 値で指定します。(x, y, z) に光源の位置を指定します。これによって、その位置から 360 度全方向へ光が放射されます。反射光の計算は、光線ベクトルの計算（光源の位置と被写体表現の入射点の位置から計算）

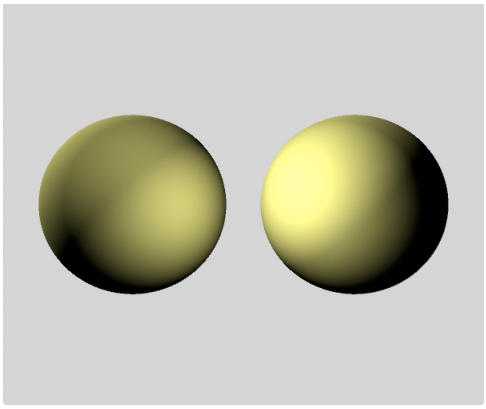


図 6.29: 図 6.25 のプログラムで点光源と平行光線の両方を用いた場合の実行結果

が追加されることを除けば、平行光線と同じです。

図 6.25 は点光源を用いたプログラムです。簡単のため、環境反射、鏡面反射は省略しました。ここでの被写体の拡散反射のマテリアルは少し黄色を帯びた明るい色（$(200, 200, 100)$）です。点光源は白色（$(255, 255, 255)$）であり、視点（位置は $(200, 0, 0)$）と二つの球（位置はそれぞれ $(0, 60, 0)$、$(0, -60, 0)$）の間（$(80, 0, 0)$）に置きました（図 6.26 参照）。このプログラムの実行結果は図 6.27 です。赤い矢印は、点光源から球への光線を示すために書き加えました。比較のため、図 6.28 は点光源を平行光線に置き換えた場合の画像です。プログラム中の点光源をコメントアウトし、平行光線をアンコメントしてください。この場合、光線（赤い矢印）は左上から二つの球へ平行に入射しており、反射光は左上に偏っています。

複数の光源

p5.js では複数の光源を設定可能です。

たとえば図 6.25 のプログラムの二つの光源を同時に有効にすると図 6.29 を得ます。これは図 6.27 と図 6.28 を合成したものに他なりません。

単に複数の光源を用いるだけではなく、それぞれに個別の色、輝度を設定することも可能です。その場合、光源毎に設定します。

図 6.30 は、2 方向 $(0, -1, -1)$ と $(0, -1, 1)$ の平行光線を設定したプログラムです。簡単のため、環境反射光は省略しました。一つ目の設定では赤い鏡面反射を設定しています。二つ目の設定では青い鏡面反射を設定しています。二つの鏡面反射を設定しない場合、一つ目の設定がそのまま引き継がれます。なお、フォン指数は共通の設定になっているため、１回だけ指定します。図 6.30 ではそれを球の描画の直前に設定しています。

```
function setup() {
  createCanvas(500,500,WEBGL);
  background(200);
  camera(200,150,0, 0,0,0, 0,0,-1);
  noStroke();

  specularMaterial(255);

  // 一つ目の平行光線に関する設定
  specularColor(180,0,0); // 鏡面反射光は赤
  // 拡散反射光は灰色、前方左上方向から
  directionalLight(180,180,180, 0,-1,-1);

  //二つ目の平行光線に関する設定
  specularColor(0,0,180); //鏡面反射光は青
  // 拡散反射光は灰色、前方左下方向から
  directionalLight(180,180,180, 0,-1,1);

  shininess(32.0);
          // フォン指数は全ての鏡面反射で共通

  sphere(100,100,100);
}
```

図 6.30: 2 方向からの平行光線で球を描画するプログラム

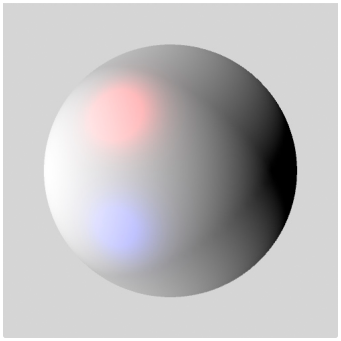


図 6.31: 左のプログラムの実行結果

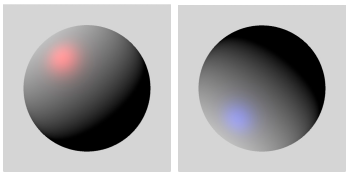


図 6.32: 単独の照明での実行結果

6.4 3D CG アニメーション

3D CG 画像をアニメーションする手順について新規の事項はありませんが、以下を再確認しておきます。

1. 視点、視野角、クリッピング空間の設定などの投影変換、視野変換の設定は `setup()` の中で 1 回だけ行うことで問題ありません。一度設定された投影変換、視野変換はプログラムの実行中、永続して保持されます。視点が動くような場合には `draw()` の中で設定を行っても構いません[17]。

2. 通常、 `draw()` の冒頭で `background()` を呼び出します。

3. 光源の設定は `draw()` の中で毎回設定する必要があります。

4. モデリング変換も `draw()` の中で毎回設定する必要があります。

図 6.33 とその実行例の図 6.34 は 3D CG アニメーションを行う例です。

[17] 図 5.36 のプログラムは `draw()` の中で視点を設定した例です。

```
function setup() {
  createCanvas(600, 500, WEBGL);

  // 視点の設定は 1 回だけでよい
  camera(200, 0, 0, 0, 0, 0, 0, 0, -1);
  noStroke();
 }

function draw() {
  background(200); // キャンバスの初期化

  // 光源の設定は draw() の中で毎回行う。
  fill(200, 200, 100);
  pointLight(255, 255, 255, 80, 0, 0);
  directionalLight(255, 255, 255, 0, -1, -1);

  // 被写体の座標変換は draw() の中で毎回行う。
  rotateZ(3 * radians(frameCount)); //座標の回転
  translate(0, 60, 0);
  sphere(50, 100, 100);
  translate(0, -120, 0);
  sphere(50, 100, 100);
}
```

図 6.33: 図 6.29 の球を回転させるプログラム

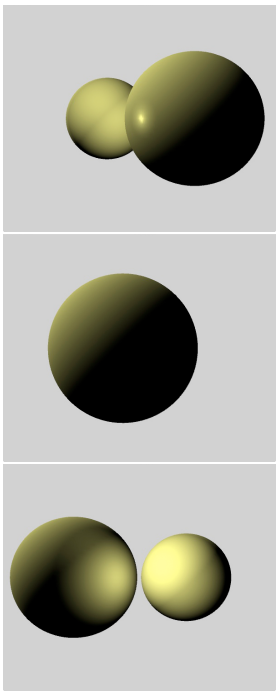


図 6.34: 左のプログラムの実行結果

6.5　陰影づけの線形補間

この節は、曲面における陰影付けの技法に関する解説です。

陰影付けの例題プログラムでは、球の描画を `sphere(100, 100, 100)` などとパラメータ指定し、球の表面を緯度/経度に沿って 100 分割する設定を行ってきました。これは球を十分滑らかに描画することを意図したためです。この場合、球の表面を 2 万枚近くの三角形で近似するため、実際、十分滑らかな描画が可能です。

ここで逆に、分割数を極端に小さくした場合、つまり、少数の大きな三角形で球の表面を覆った場合、球はどう描画されるのでしょうか。たとえば `sphere(100, 16, 8)` を設定した場合、つまり、経度方向に 16 分割、緯度方向に 8 分割した場合を考えてみます。

6.5.1　フラットシェーディング

一枚の三角形は平らな面ですから、三角形のどの内部点でも法線ベクトルは同一です。平行光線を用いる場合には内部点に入射する平行光線もどこでも同一です。よって、式 (6.4) の拡散反射の値は三角形の内部点で全て同じ値となります。そうすると、隣り合う三角形の境界では陰影付けが不連続に変化します。一般に拡散反射光は陰影付けに最も大きく寄与しますから、この不連続性は描画像にはっきり見えるはずです。図 6.35 (a) の左図はその説明図です。T_1、T_2、T_3 は 3 角形の表面を表します。それぞれの三角形ではそれぞれの法線ベクトルを求め、陰影付けを行います。そうすると、T_1 と T_2 の境界 P_1、T_2 と T_3 の境界 P_2 では陰影計算が不連続になる訳です。図 6.35 (a) の右図はこれを実際に 3D CG 画像化したものです。緯度/経度に沿って陰影が不連続に変化していることが分かります。このように三角形毎に陰影付けを行うこの方法を**フラットシェーディング**（flat shading、平坦な陰影付け）と呼んでいます。これは最も単純な陰影付けの方法です。

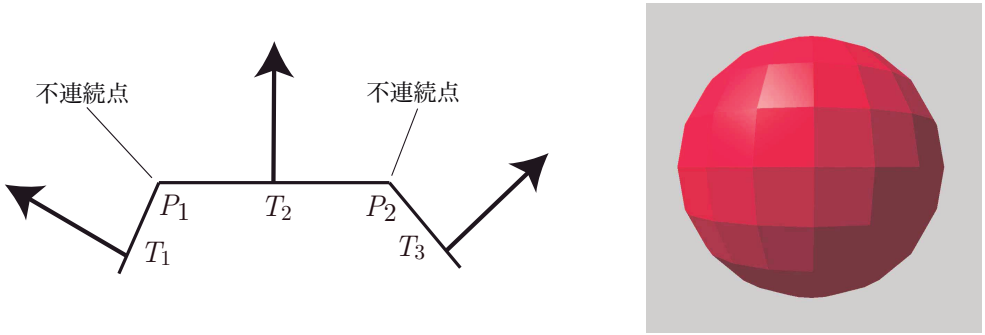
フラットシェーディングは曲面の描画に適しません。この方法で曲面を滑らかに描画するには曲面を多数の三角形で覆う必要がありますが、しかしこれは根本的な解決策にはなりません。視点が曲面に十分に近づけば、その粗（陰影の不連続性）が拡大されて描画されるからです。

6.5.2　スムーズシェーディング

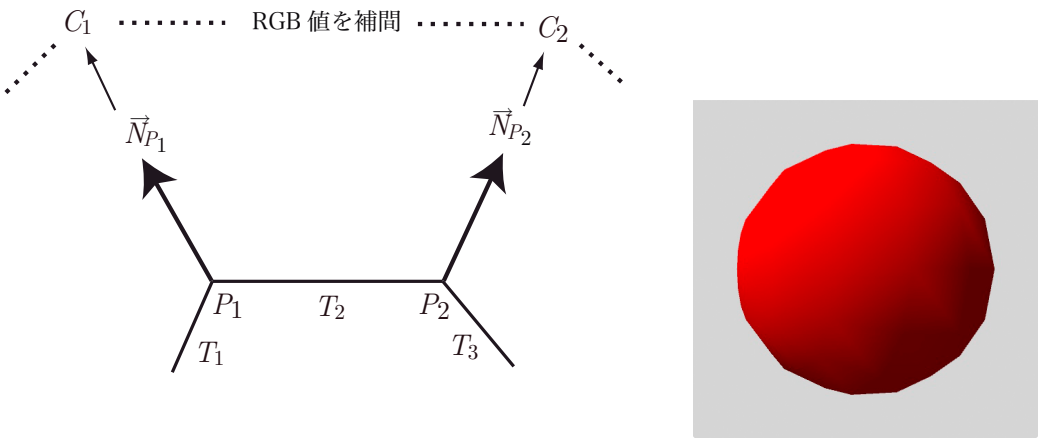
上の問題を解決する方法として、線形補間を用いて平面を曲面に見せかける技法が利用されています。これは**スムーズシェーディング**（smooth shading）[18] と呼ばれ、p5.js ではこの技法が用いています。

実はフラットシェーディングの描画例の図 6.35(a) 右図は p5.js で作成した画像ではありません。授業担当者が Visual Studio でフラットシェーディングを含む 3D CG プログラムを作成し、描画

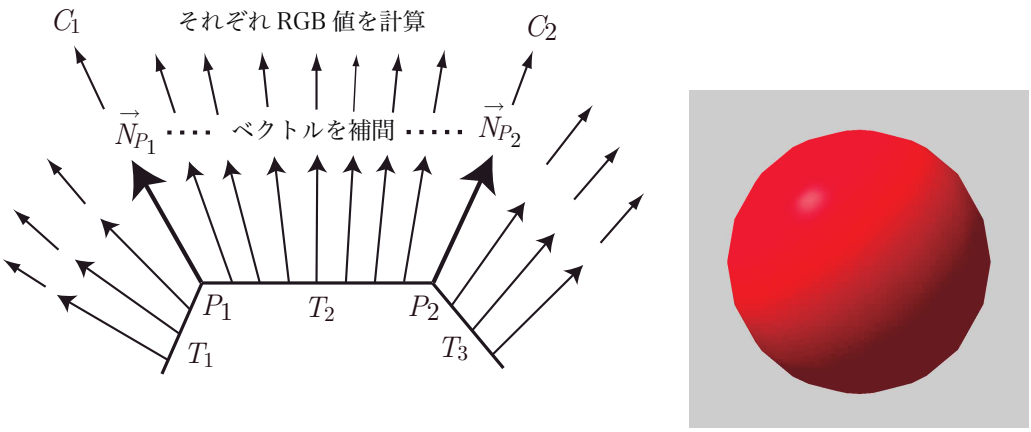
[18] あえて邦訳すれば「滑らかな陰影付け」ですが、そのような言い方は普及していません。



(a) フラットシェーディング



(b) グーローシェーディング



(c) フォンシェーディング

図 6.35: 3 種類の補間方法の違い

したものです[19]。p5.js のプログラム（図 6.21）で `sphere(100, 16, 8)` を実行した結果は、図 6.35(c) の右の画像です。スムーズシェーディングを用いているため、ほぼ滑らかな描画になっています。

スムーズシェーディングには二つの代表的な技法があります。ひとつは**グーローシェーディング**（Gouraud shading）[20]、もうひとつは**フォンシェーディング**（Phong shading）です。p5.js ではフォンシェーディングを採用しています。以下、これらについて概説します。

グーローシェーディング

この方法は、三角形の 3 頂点で陰影計算したRGB 値を用いて三角形の内部点の RGB 値を線形補間する手法です。図 6.35(b) 左図がその説明図です。三角形 T_1 と T_2 に共有される境界点 P_1 の法線ベクトル $\vec{N}_{P_1}$ を T_1 と T_2 の法線ベクトルの平均値として定義します。そしてそれら法線ベクトルを用いて P_1 の点の反射光 C_1 を計算します。同様に点 P_2 の法線ベクトル $\vec{N}_{P_2}$ を T_2 と T_3 の法線ベクトルの平均値として定義し、その法線ベクトルを用いて P_2 の点の反射光 C_2 を計算します。三角形の内部点の反射光は C_1、C_2 の線形補間で計算します。図 6.35(b) 左図では三角形が線分に単純化されていますが、実際には 6.2.3 節で紹介した重心座標系を用いた線形補間で計算します。つまり、三角形の 3 頂点 A、B、C のそれぞれで計算された反射光を C_A、C_B、C_C とするときに任意の内部点 P の反射光 C_P を

$$C_P = \lambda_A \cdot C_A + \lambda_B \cdot C_B + \lambda_C \cdot C_C$$

と計算します。

グーローシェーディングでは三角形 T_1、T_2、T_3 の反射光が連続的に変化していくため、境界での不連続性は現れません。比較的少ない計算で擬似的に滑らかな面の陰影付けができる点で優れています。

なお、図 6.35(b) 左図では点 P_1 が二つの三角形 T_1 と T_2 で共有されているように図示されていますが、ひとつの点が多数の三角形で共有されている場合もあります。たとえば図 6.36 は四つの三角形で共有されている場合です。この場合には以下の方法で P_1 上の法線ベクトルを計算します。

$$\vec{N}_{P_1} = \text{normalize}\left(\vec{N}_1 + \vec{N}_2 + \vec{N}_3 + \vec{N}_4\right)$$

ここに normalize() は単位ベクトルを求める正規化関数です。

[19] 自作プログラムで p5.js と同じ条件（視点の位置、マテリアルの色などシェーディング技法を除く全て）を設定していますが、微妙に色合いが異なっています。動作環境の違いが影響していると思われます。

[20] 提案者の名前：アンリ・グーロー (Henri Gouraud) に由来します。邦訳ではグローシェーディングと書かれる場合もあります。

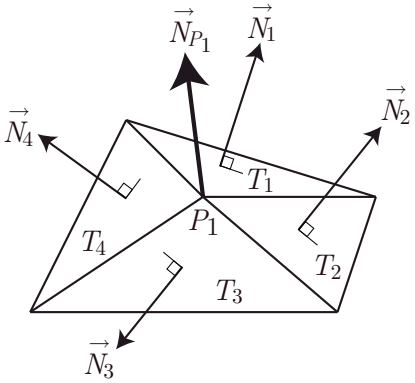


図 6.36: 頂点の法線ベクトルの計算

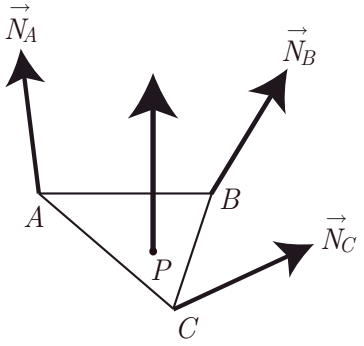


図 6.37: 法線ベクトルの補間計算

フォンシェーディング

フォンシェーディングはグーローシェーディングよりも精密な線形補間を行います。ただしその分だけ計算コストが上がります。

図 6.35(c) 左図がその説明図です。この方法ではまず、グーローシェーディングと同様に、三角形の各頂点でその頂点を共有する三角形の法線ベクトルの平均値 $\vec{N}_{P_1}$、$\vec{N}_{P_2}$ を計算します。次に、三角形の各内部点の法線ベクトル $\vec{N}$ を $\vec{N}_{P_1}$ と $\vec{N}_{P_2}$ の線形補間で計算します。図 6.35(c) 左図では三角形が線分に単純化されていますが、実際には 6.2.3 節で紹介した重心座標系を用いた線形補間で計算します。つまり、三角形の 3 頂点 A、B、C のそれぞれで計算された法線ベクトルを $\vec{N}_A$、$\vec{N}_B$、$\vec{N}_C$ とするときに任意の内部点 P の法線ベクトル $\vec{N}_P$ を

$$\vec{N}_P = \lambda_A \cdot \vec{N}_A + \lambda_B \cdot \vec{N}_B + \lambda_C \cdot \vec{N}_C$$

と計算します[21]（図 6.37 参照）。そして、$\vec{N}_P$ を元にその点の陰影計算を行います。

グーローシェーディングが RGB 値を補間するのに対して、フォンシェーディングでは陰影計算の基となる法線ベクトルを補間する訳です。そのため、フォンシェーディングの方が計算量が増えます[22]（各内部点毎に陰影計算が必要であるため）が、より精度の高い、自然な陰影計算が可能です。特に、三角形の 3 頂点では鏡面反射がほとんど生じないものの内部点で鋭い鏡面反射が起きるような場合、グーローシェーディングでは頂点の計算結果を補間するため、鏡面反射が内部点に反映されませんが、フォンシェーディングでは法線ベクトルを補間するため、その鏡面反射をかなり正確に描画できます。少し見えにくいのですが、図 6.35(c) 右図の球の左上部分には白い鏡面反射が見られます。しかし、図 6.35(b) 右図の同じ場所には鏡面反射は描画されていません。鏡面反射は被写体の材質感に大きく影響するため、この違いは無視できません。

[21] 一般には $\vec{N}_P$ は単位ベクトルでないため、さらに正規化を行います。
[22] 現在の GPU は高速であり、この計算量はほとんど問題にならなくなってきています。

第XIV部

6.6　ファイルを用いた描画

この講義テキストでは様々な CG 技術を可能な限り小さいプログラムを Web エディタで実行しながら解説してきました。しかし、より印象的な、より複雑な CG 作成には、データを別途、ファイルで用意し、それをプログラム中に読み込む技術が避けて通れません。ここではその方法を例題と共に紹介します。

6.6.1　環境設定

p5.js でのファイル読み込みは以下の 2 通りの方法で可能です。

1. p5.js のサイトにアカウントを作成します。そうすると、サイトが提供するサーバーのアーカイブ（ファイル保存エリア）が利用可能になります。そこでサーバーの自分のアカウントのエリアに事前にファイルを保存しておきます。Web エディタのプログラムはそのファイルを読むことが可能です。

2. PC にローカルに p5.js の実行環境を構築します。そうすれば PC 上のファイルを読むことが可能です。

ここでは 1. の方法を紹介します。2. については各自で調べてください。

6.6.2　画像の描画

上記 1. の方法で jpeg 画像を p5.js のキャンバスに表示する方法を解説します。

画像ファイルのアップロード

図 6.38 は画像ファイル “`flowers.jpg`”[23] をキャンバスに描画するプログラムです。図 6.39 はその実行結果です。

このプログラムでは `loadImage("flowers.jpg")` によって画像ファイル “`flowers.jpg`” をプログラム内に読み込んでいます。その読む込みの準備として、画像ファイルは事前にプログラム環境にアップロードしておく必要があります。そのアップロードの手順は次の通りです。

1. まず、Web エディタの右上のメニュー（図 6.40 参照）の “Log in” からアカウントにログインします。アカウントは事前に “Sign up” で作っておきます

2. 次に、図 6.41 の左の赤い矢印の箇所をクリックし、このプログラムを構成している全ファイルを表示します。そうすると、現在の構成ファイルが編集エリアの左側に表示されます。

[23] この花の画像は「フリー写真素材ぱくたそ」（`https://www.pakutaso.com`）からダウンロードしました。

```
let img;

function preload() {
  img = loadImage("flowers.jpg");
}

function setup() {
  createCanvas(640,480);
  background(200);
  image(img, 10, 10);
}
```

図 6.38: 画像の描画プログラム（その 1）

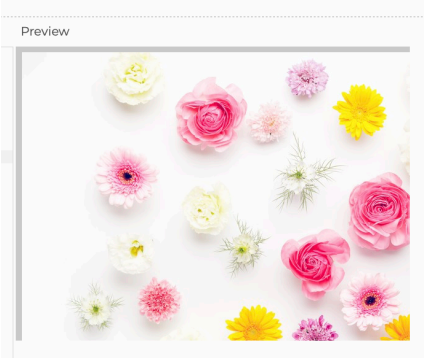


図 6.39: 左のプログラムの実行結果

3. 次に、図 6.42 の左の赤い矢印の指している箇所（“Sketch F”の右隣の下向き三角形）をクリックします。そうすると、図 6.43 のようなプルダウンメニューが表示されます。

4. 次に、図 6.43 のプルダウンメニューから “Upload file”を選びます。そうすると、図 6.44 のように、ファイルアップロード用のウインドウが開きます。

5. このウインドウにファイルをドラッグすることでファイルのアップロードが可能です。あるいはウインドウをクリックすると、アップロード用のファイルブラウザが開きますから、それを用いてファイルを選択できます。そうすると、図 6.45 のように構成ファイルにアップロードしたファイルが追加されているのが分かります（この場合には “`fl...s.jpg`”）。最後にアップロード用ウインドウの右上の × 印をクリックし、アップロード用ウインドウを閉じましょう。

以上でアップロードは終了です。プログラムでアップロードしたファイルを読み込むことができます。

ところで、アカウントを作ると、サーバーにプログラムやファイルを保存できるようになります。それらの保存状況を見るには、図 6.46 のように、Web エディタのメニューから “File-Open” を選択します。

そうすると、図 6.47 のように、このアカウントで保存されているファイル（p5.js では Sketch と呼びます）の一覧が表示されます。プログラムを選択すれば、そのプログラムは Web エディタでいつでも編集、実行可能です。メニューから “Assets”を選ぶと、アップロードされたファイルの一覧が表示されます。

図 6.40: アカウントへのログイン：Web エディタの右上のメニューから事前に登録したカウントでログインする

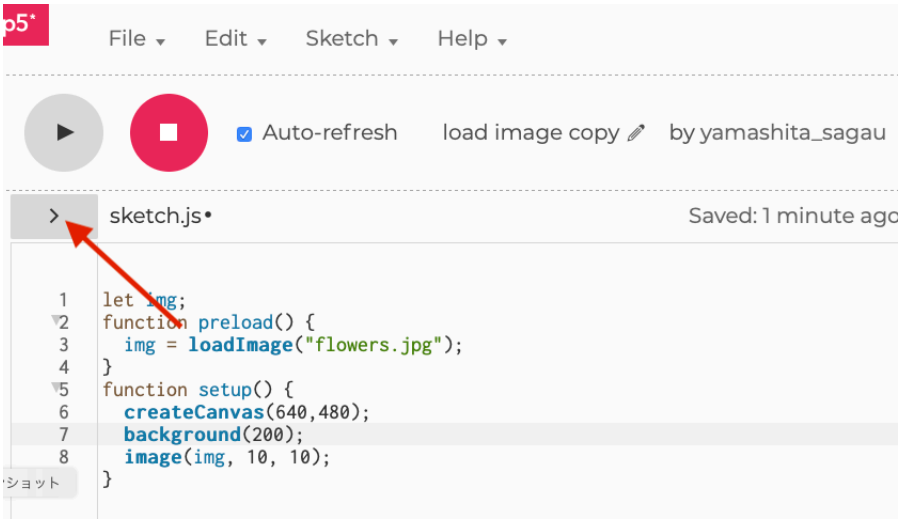


図 6.41: 構成ファイルの表示

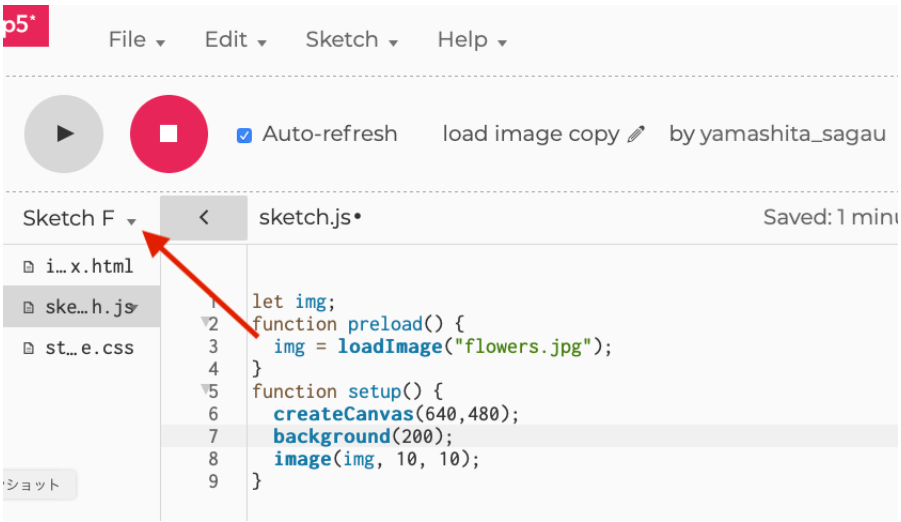


図 6.42: プルダウンメニューの表示

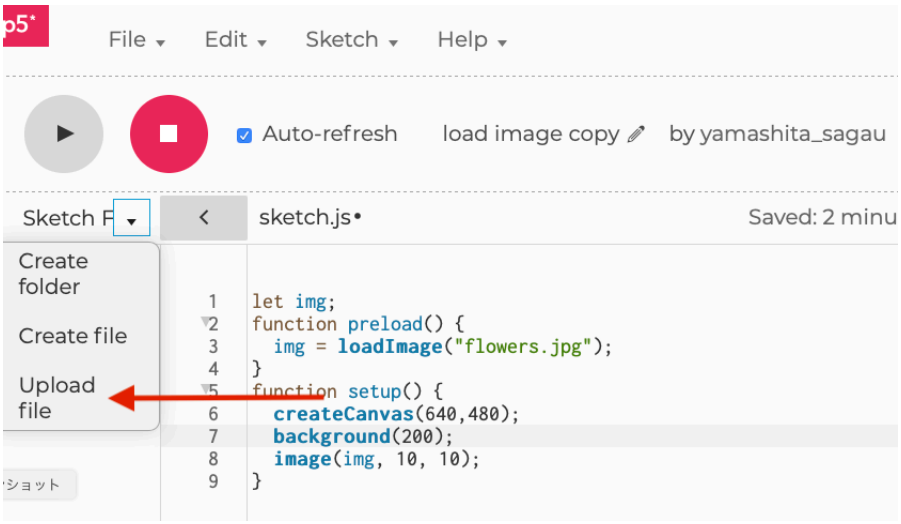


図 6.43: アップロードコマンドの選択

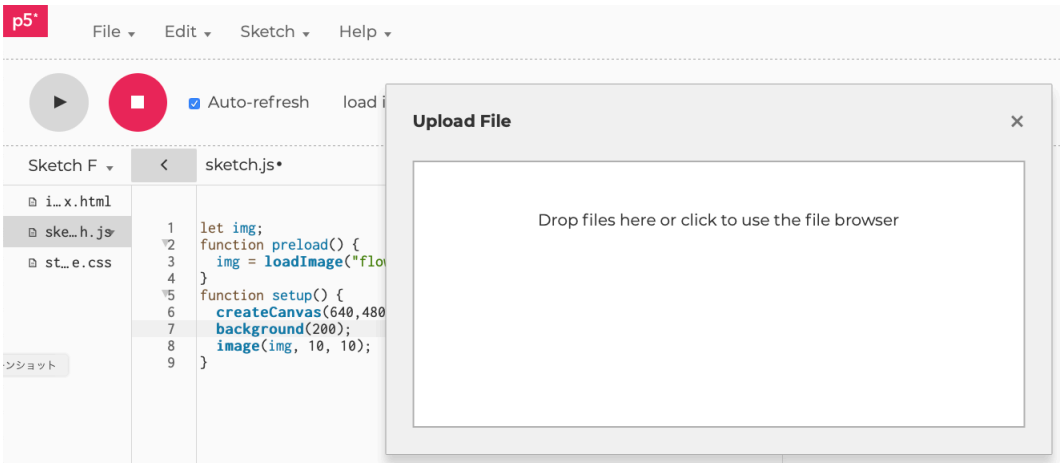


図 6.44: アップロード用ウインドウの表示

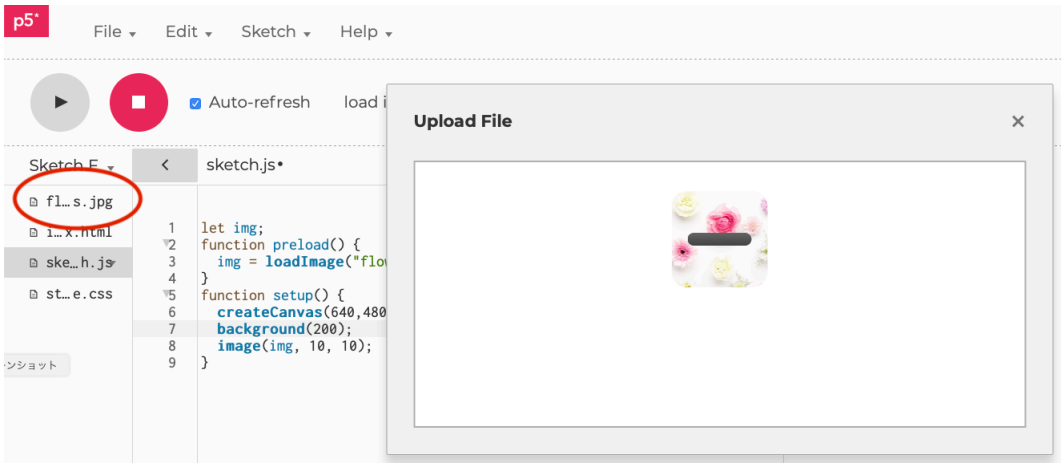


図 6.45: アップロードの完了

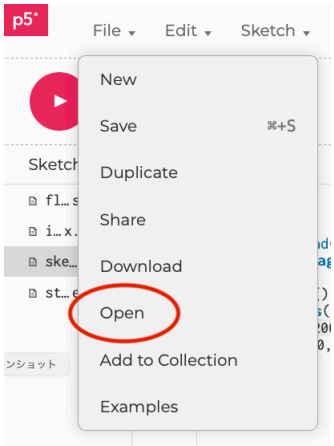


図 6.46: アカウントの保存ファイルを調べるには “File-Open”

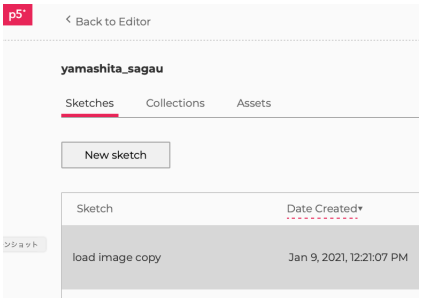


図 6.47: プログラム（スケッチ）のリスト

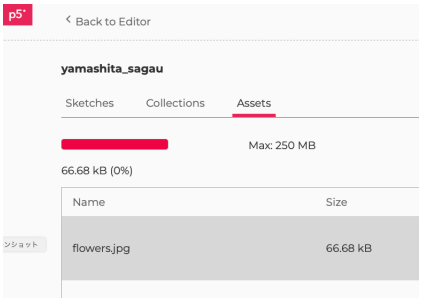


図 6.48: アップロードファイルのリスト

矩形への画像描画

p5.js では、各種ファイルの読み込みは関数 `preload()` で行う必要があります。この関数の内容は以下の通りです。

`preload()` ... プログラム中にこの関数が定義されている場合、`setup()` の実行前にこの関数が呼び出されます。ファイルの内容をプログラム中に読み込む作業はほとんどの場合[24]、この関数で事前に行うことが推奨されています。

画像ファイルの読み込みには以下を用います。

`p5.Image loadImage(filename)` ... 画像ファイル名を引数 `filename` に指定します。`p5.Image` 型の画像データが作成され、関数の戻り値となります。読み込むことのできる画像形式は、`.jpg`、`.png`、`.gif` などです。

読み込まれたデータはグローバル変数 `img` に一旦保持し、`setup()` 内で描画に使用します。画像データの描画には以下を用います。

`image(img, x, y)` ... 引数 `img` に `p5.Image` 型画像データを指定します。引数 `x`、`y` にはキャンバス上に描画する画像の左上の座標点 (x, y) を指定します。キャンバスに描画される画像の水平/垂直方向のサイズは元の画像データの水平/垂直方向のサイズです。

`image(img, x, y, w, h)` ... 引数 `img`、`x`、`y` は上と同じです。キャンバスに描画される画像の水平方向/垂直方向のサイズは引数 `w` / `h` で指定します。指定されたサイズに合うように画像が縦横に拡大/圧縮されて描画されます。

図 6.38 のプログラムでは、3 引数の `image()` を用いており、キャンバスに描画された画像の左上がキャンバスの左上から縦横それぞれ 10 画素分だけ右下へずれています。なお、画像ファイルは 800×533 画素の大きさのため、キャンバスに入りきれていません。

これに対して、図 6.49 のプログラムは 5 引数の `image()` を使用し、画像サイズを明示的に与えています。この場合には、画像ファイルの元のサイズは失われ、縦横に拡大/縮小された画像が描画されます。

背景としての描画

画像を描画するもうひとつの方法は、背景（background) 画像として描画する方法です。図 6.51 のプログラムでは関数 `background()` の引数として画像データを与えています。

`background(PImage img)` ... キャンバスの背景を画像データ `img` で描画します。画像はキャンバスのサイズに合わせて縦横が拡大縮小されます。

[24] 非常に小さなサイズのファイルを読み込むなどの例外的な場合を除きます。

```
let img;

function preload() {
  img = loadImage("flowers.jpg");
}

function setup() {
  createCanvas(640, 480);
  background(200);
  image(img, 10, 10, 620, 460);
}
```

図 6.49: 画像の描画プログラム（その 2）

図 6.50: 左のプログラムの実行結果

```
let img;

function preload() {
  img = loadImage("flowers.jpg");
}

function setup() {
  createCanvas(640, 480);
  background(img);
}
```

図 6.51: 画像の描画プログラム（その 3）

図 6.52: 左のプログラムの実行結果

考察　関連事項として、余裕のある人は p5.js の関数 `tint()` について調べなさい。

6.6.3　形状定義ファイルの読み込み

図 6.53 は形状定義ファイル “`utahteapot.obj`”[25]をキャンバスに描画するプログラムです。図 6.54 はその実行結果です。

p5.js では以下の関数を用いてモデラーなどで別途、作成し、obj ファイル、stl ファイルに格納された形状を読み込むことができます。形状定義ファイルの読み込みには以下を用います。

`p5.Geometry loadModel(filename, norm)` ... ファイル名を引数 `filename` に指定します。`p5.Geometry` 型の形状データが作成され、関数の戻り値となります。読み込むことのできる画像形式は、`.obj`、`.stl` です。引数 `norm` には、形状のサイズを正規化するか否かを指定します。正規化する場合、`true` を指定します。形状のサイズが不明の場合に便利です。

[25] この形状定義ファイルはインターネットから取得できますが、著作権の詳細が確認できないため、このテキストで URL を明示することは控えます。

```
let teapot;

function preload() {
  teapot = loadModel("utahteapot.obj", false);
}

function setup() {
  createCanvas(700, 500, WEBGL);
  camera(200, 100, 150, 0, 50, -40, 0, -1, 0);
  noStroke();
  background(200);

  specularColor(80, 80, 80);
  specularMaterial(60, 0, 0);
  shininess(32.0);
  fill(0, 255, 0);
  directionalLight(230, 230, 230, 0, -1, -1);

  scale(50, 50, 50);
  model(teapot);
}
```

図 6.53: obj 形式の形状定義ファイルを読み込み、描画するプログラム

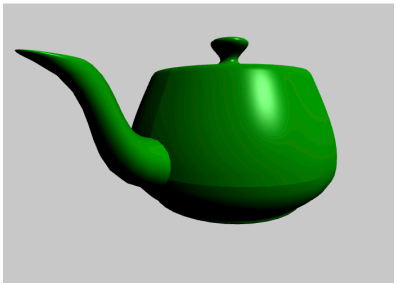


図 6.54: 左のプログラムの実行結果

図 6.53 では、読み込まれたデータをグローバル変数 teapot に一旦保持し、setup() 内で描画に使用します。描画には以下を用います。

model(geo) ... 引数 geo に p5.Geometry 型形状データを指定します。

この例で使用されている形状は Utah 大学で CG 研究用に作成されたもので、Utah teapot と呼ばれ、3D CG の分野では非常に有名です[26]。この形状には、平面、曲面、凹凸、トーラスなどの 3D CG 作成で注意すべき様々な形状が含まれているため、3D CG の研究でよく用いられています。

6.6.4 様々な形状の定義

p5.js では被写体の形状を自由に定義することも可能です。図 6.55 のプログラムでは beginShape()、endShape() で形状を構成する頂点を順番に指定し、簡単なリボンの形状を定義しています。以下はこれに関連する関数です。

beginShape() ... ひとつの被写体の形状定義を開始します。引数として POINTS、LINES、TRIANGLES、TRIANGLE_FAN、TRIANGLE_STRIP、QUADS、QUAD_STRIP、TESS が指定できます。詳細は省略

[26] ハリウッド映画にもしばしば登場することが知られています。

```
function setup() {
  createCanvas(500, 500, WEBGL);
  camera(150, 150, 100, 0, 0, 0, 0, 0, -1);
  noStroke();
}

let s = 0;

function draw() {
  background(128);
  directionalLight(50, 250, 250, 0, 0, -1);
  rotateY(0.03 * s);

  beginShape(TRIANGLES); // 形状定義の開始
  vertex(0, 0, 0);       // 頂点位置の設定
  vertex(0, 100, -100);  // 頂点位置の設定
  vertex(0, -100, -100); // 頂点位置の設定
  vertex(0, 0, 0);       // 頂点位置の設定
  vertex(0, 100, 100);   // 頂点位置の設定
  vertex(0, -100, 100);  // 頂点位置の設定
  endShape(CLOSE);       // 形状定義の終了
  s += 1;
}
```

図 6.55: 回転する青いリボンの描画プログラム

図 6.56: 左のプログラムの実行結果（上：上からの光線を浴びて拡散反射が強い、中：光線が斜めに当たって拡散反射が弱い、下：被写体の裏側は黒に描画される）

します。例題では、TRIANGLES を指定しています。この場合、3 頂点（3 回の vertex() の呼び出し）毎にひとつの三角形を定義することを意味します。

vertex(x, y, z) ... 被写体を構成するひとつの頂点の位置 (x, y, z) を指定します。この関数で指定された頂点間を順に結ぶように被写体の境界が定義されます。

endShape() ... 形状定義を終了します。この関数呼び出しで一連の形状が直ちに描画されます。

6.6.5 テクスチャマッピング

被写体表面に色を付けるだけではなく、2 次元画像として別途用意しておいた模様を張り付ければ被写体により豊かな質感を与えることができます。3D CG ではこの模様のことを特に**テクスチャ**（texture）と呼んでいます。テクスチャはもともと「織物」のことを指しますが、広い意味で手ざわり、質感を意味します。織物以外にも大理石の模様、動物の皮膚、雲などが典型的なテキスチャです。これらテクスチャを写真に撮るなどして 2 次元画像データ化しておき、それを CG プログ

```
let img;

function preload() {
  img = loadImage("earth.jpg");
}

function setup() {
  createCanvas(600, 600, WEBGL);
  camera(100, -100, 0, 0, 0, 0, 0, 1, 0);
  noStroke();
}

let a = 0;

function draw() {
  background(220);
  rotateY(a);

  texture(img);

  sphere(50, 24, 24);
  a += 0.01;
}
```

図 6.57: obj 形式の形状定義ファイルを読み込み、描画するプログラム

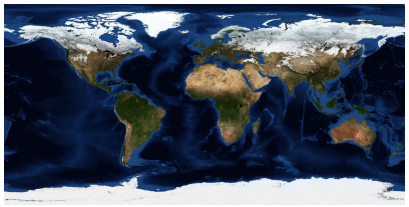


図 6.58: 地球のテクスチャ画像

図 6.59: 左のプログラムの実行結果

・・・

ラムで被写体に張り付ければ実物に近い質感をまねることができます。この処理を**テクスチャマッピング**（texture mapping）と呼んでいます。

p5.js では p5.Image 型の画像を被写体に貼り付けることができます。

たとえば図 6.57 は、球に地球のメルカトル座標の画像[27]（図 6.58）を球に貼り付けて描画するプログラムです。このプログラムでは以下の関数を使用しています。

texture(img) ... 引数の画像 img をこの関数呼び出しの後に描画される直方体（box()）、球（sphere()）、トーラス（torus()）に、それぞれの形状であらかじめ定められた方法で貼り付けます。貼り付け方法をプログラマが詳細に指定することも可能なようですが、調査していません。

図 6.60、図 6.61 は、図 6.57 のプログラムを少し改造した描画例です。プログラムは省略します。

[27] NASA のフリー画像のページ https://visibleearth.nasa.gov から取得

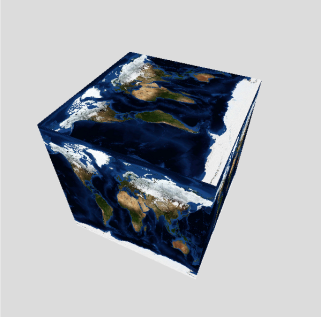


図 6.60: 立方体に地球のテクスチャを貼った場合

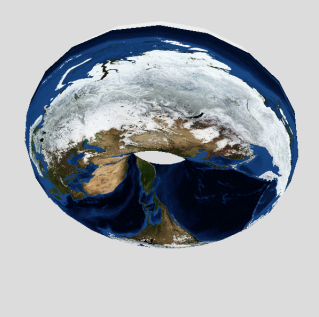


図 6.61: トーラスに地球のテクスチャを貼った場合

第XV部

第7章　その他の話題

7.1　グラフィックスプロセッシングユニット

最近ではほとんどのコンピュータ（ノート PC を含む）に**グラフィックスプロセッシングユニット**（graphics processing unit、以下 GPU）が搭載されており、CG 画像を GPU のハードウェアで高速に描画できます。p5.js も GPU を用いた描画を行なっています。

ここでは GPU による高速描画の概要を紹介します（図 7.1 参照）。

GPU は複数の計算ユニットがパイプライン的に動作するように作られています。また、各計算ユニットは多数の（数百個程度の）小規模な CPU で構成されており、大量の入力データを同時並列的に処理します。各計算ユニットの処理内容は以下の通りです。

バーテックス・シェーダ　（vertex shader）[1] ... 被写体の各頂点についてモデリング変換（5.7 節）、視野変換（5.5 節）、投影変換（5.3 節）を行ないます。この計算ユニットの計算内容はプログラマが自由にプログラム可能（プログラマブル）です。

ラスタライザ　（rasterizer）　... バーテックス・シェーダで頂点情報が確定したならば、

1. 各三角形をラスタライズ（6.1 節）し、三角形の全ての内部画素点を求めます。
2. また、各画素点について、重心座標値 $(\lambda_1, \lambda_2, \lambda_3)$ を計算し、正規化された深さ値（式 (eqz)）、光線の入射点 P の座標値（6.3.3 節）、P における法線ベクトルなどを加重平均による線形補間で求めます。この計算ユニットの計算内容は固定（非プログラマブル）です。

フラグメント・シェーダ　（fragment shader）[2]　...　各画素点について拡散反射光（式 (6.4)）、環境反射光（6.3.2 節）、鏡面反射光（式 (6.7)、式 (6.8)）の計算、テクスチャ画像からの RGB カラー値の取得などを行ない、最終的にその画素点の RGB カラー値、正規化された深さ値を求めます。この計算ユニットの計算内容はプログラマが自由にプログラム可能（プログラマブル）です。

[1] 歴史的に「シェーダ」（shader）は、陰影計算（shading）を行なうプログラム、装置のことを意味します。バーテックス・シェーダに限って言えば、本文にあるように、陰影計算というよりも座標変換が主な仕事であるため、シェーダと呼ぶにはやや違和感がありますが、陰影計算の前処理部と理解すればよいでしょう。

[2] フラグメント・シェーダは Open GL の呼称です。Windows DirectX ではこれをピクセル・シェーダ（pixel shader）と呼んでいます。

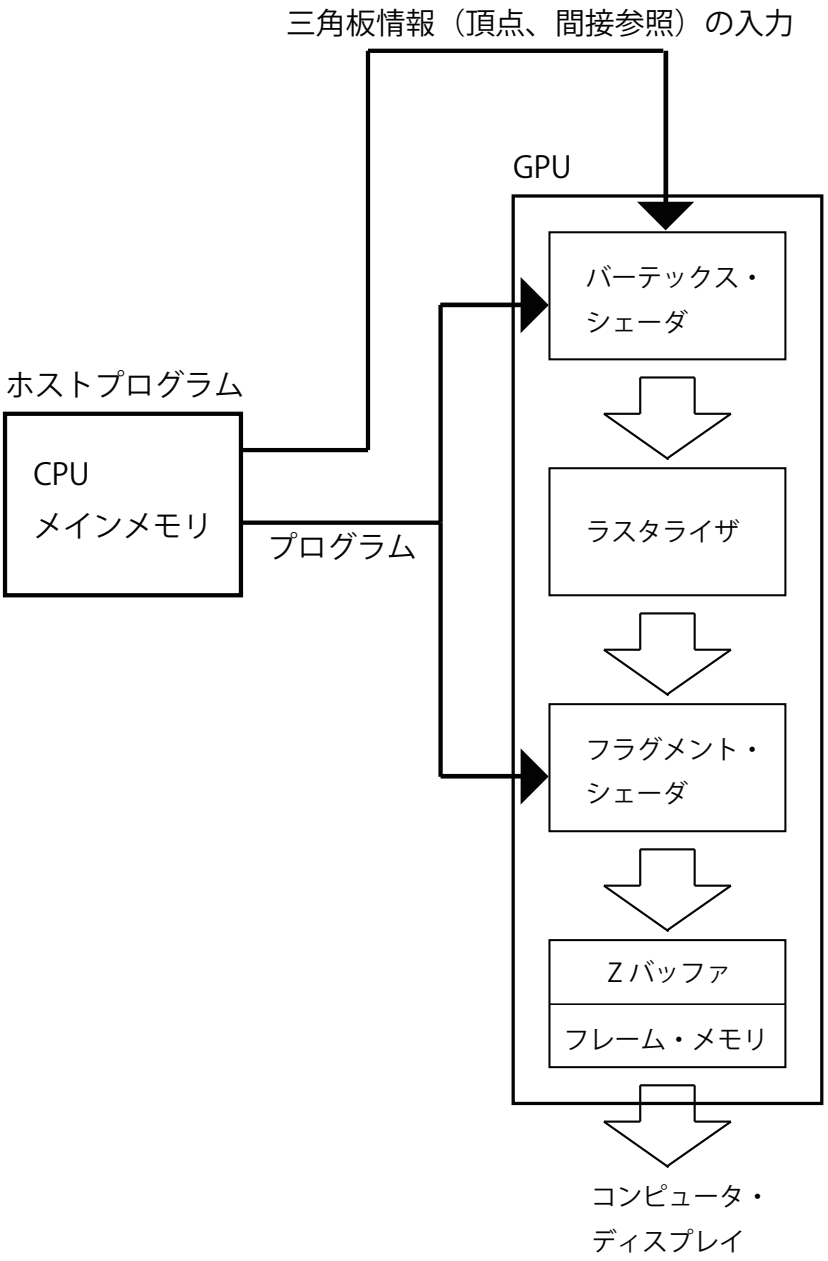


図 7.1: GPU の内部構造

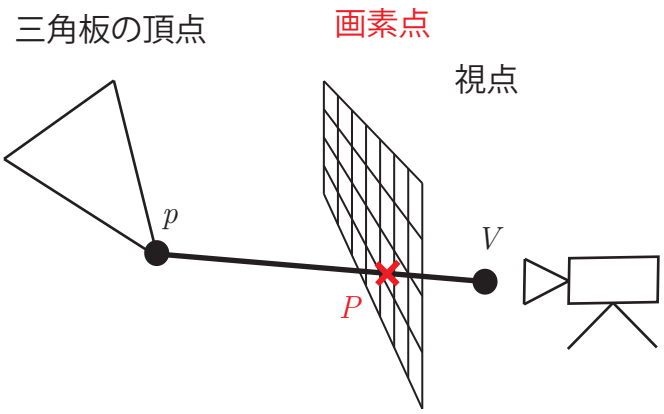


図 7.2: ラスタライズ法の枠組み

Z バッファ/フレームバッファ ... 各画素点の正規化された深さ値に基づいて、その画素点の RGB カラー値をフレームバッファに色付けするか否かを判断します。

このように、5 章、6 章で学んだ内容がほぼそのまま GPU に実装されており、その意味で 5 章、6 章の内容は 3D CG の標準的な描画手法と言ってよいでしょう。

上にバーテックス・シェーダ、フラグメント・シェーダはプログラマブルと述べました。ならば、これら計算ユニットで CG 描画に関係もない大規模数値計算を高速実行させようというアイディアが生まれ、2010 年頃からその試みが広く認知されるようになってきています。これを **GPGPU**（General-Purpose computing on Graphics Processing Units）と呼びます。最近では GPGPU 専用のプログラミング環境 CUDA、OpenCL なども登場し[3]、利用が拡大しています。

7.2　レイトレーシング

実は 3D CG（面画）の描画方法には大きく分けて 2 種類あります。ひとつは、前章まで述べてきた方法です。これを**ラスタライズ法**（rasterizing method、rasterization method）と呼びます。あるいはこの方法を構成するアルゴリズム群の中の最も重要な処理のひとつに由来し、Z バッファ法、深さバッファ法と呼ぶこともあります。この方法の特徴は、

入力情報として、 視点の位置[4] V と各三角形の頂点 p を与えて

出力情報として、 V と p を結ぶ直線が投影面と交差する画素位置を求める

ことです（図 7.2 参照）。

[3] 厳密に言えば、OpenCL は GPU 専用ではなく、より広く並列計算をサポートするプログラミング環境ですが、GPU を主なターゲットにしています。

[4] 正確に言えば、視点（視線）の方向も重要ですが、方法の大枠を論じる立場では除外して考えます。

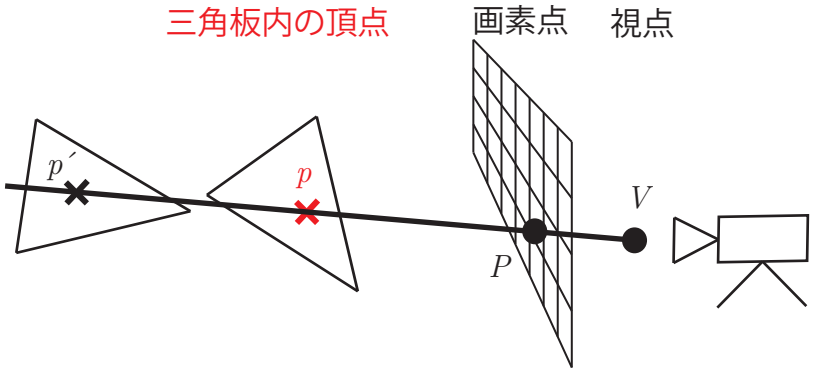


図 7.3: レイトレーシング法の枠組み

これに対して、**レイトレーシング法**（ray tracing method、または光線追跡法）と呼ばれる、ラスタライズ法とは全く別のアプローチで 3D CG 画像を作成する方法が提案されています。この方法の特徴は

入力情報として 、視点の位置 V と投影面上の各画素点 P を与えて

出力情報として 、V と P を結ぶ直線が交差する三角形上の点 p を求め、その点で陰影計算を行なう

ことです。図 7.3 に概念図を示しました。一般に、V と P を結ぶ直線に交差する三角形は複数ありえますが、交差点の中で最も視点に近い点が選ばれます。図 7.3 の場合、p と p' の 2 カ所で交差していますが、視点に近い p が選ばれることになります。

被写体は、被写体から発射された光線が投影面上のある画素点を通って視点に入射する過程で映像化されます。レイトレーシング法ではこの過程を逆に辿り、視点から画素点を通って発射された光線が被写体に入射する過程をシミュレートします。逆の過程とは言え、物理的な自然なシミュレーションになっていますから、様々な物理現象を 3D CG に素直に実現することができ、よりリアルな CG 画像が得やすいという利点があります。そのひとつの例証として、レイトレーシング法では被写体による光線の反射、屈折を物理現象に則ってかなり忠実にシミュレートできることが知られています。

図 7.4 で説明します。視点 V から発射された光線が画素点 P と通り、点 p_1 で被写体と交差したと仮定しましょう。もし被写体がガラスのような透明、半透明の材質の場合、入射光線の一部は点 p_1 で反射し、別の一部は屈折するでしょう。そこで p_1 で反射する光線が次に交差する三角形を求めます。たとえばそれが点 p_2 で被写体と交差したとしましょう。点 p_2 でも反射と屈折が起きますから、さらにそこから再帰的に光線を追跡していくことができます。p_1 で屈折した光線についても同様です。これを繰り返して様々な反射光、屈折光の強度を計算することで、逆に画素点 P に集積する光の強度を計算できます。図 7.5 は、上のような方法で作成した CG 画像の例です。

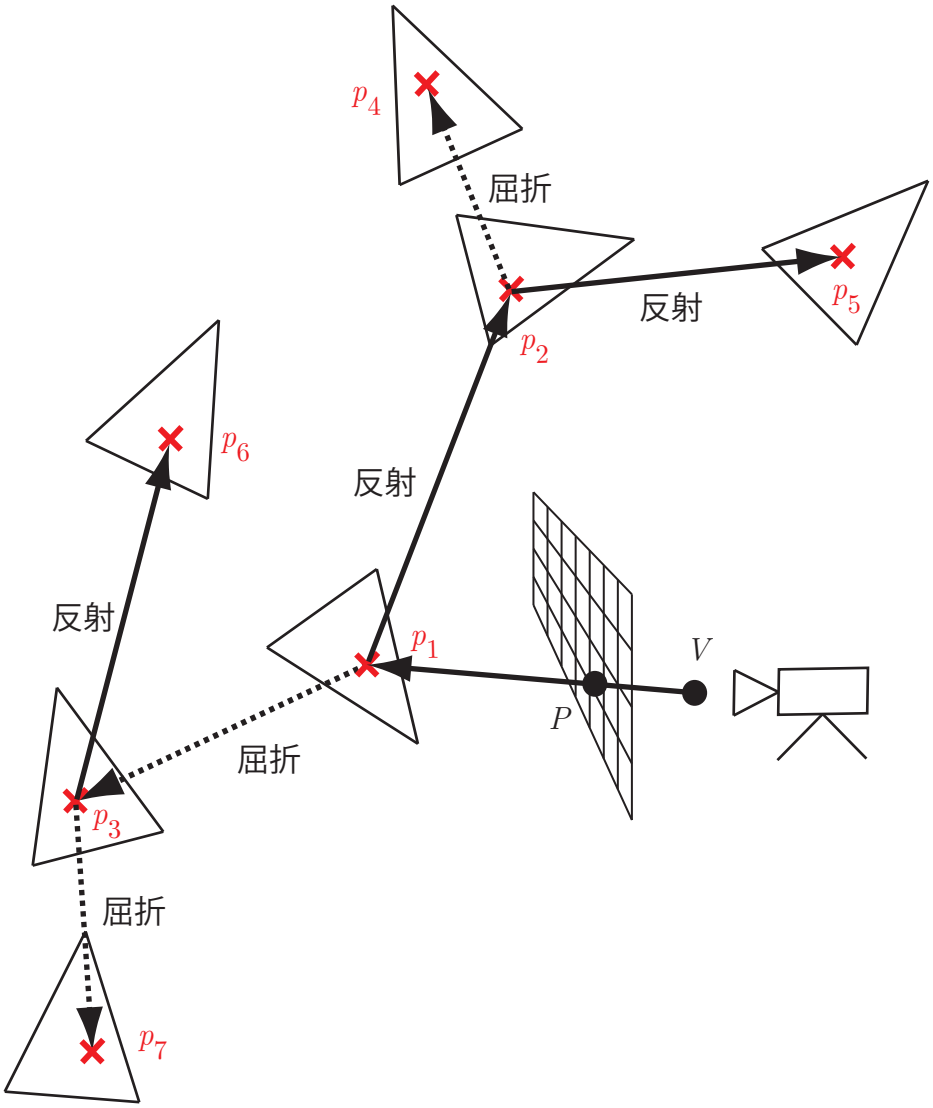


図 7.4: レイトレーシング法による光線の反射・屈折の再帰的計算

表 7.1: 二つの描画方法の比較

描画手法	ラスタライズ法 （このテキストの方法）	レイトレーシング法
与えられる情報	視点の位置、頂点の位置	視点の位置、投影面上の位置
求めるべき情報	投影面上の位置	光線と交差するポリゴン上の位置
特徴	高速描画	リアルな描画

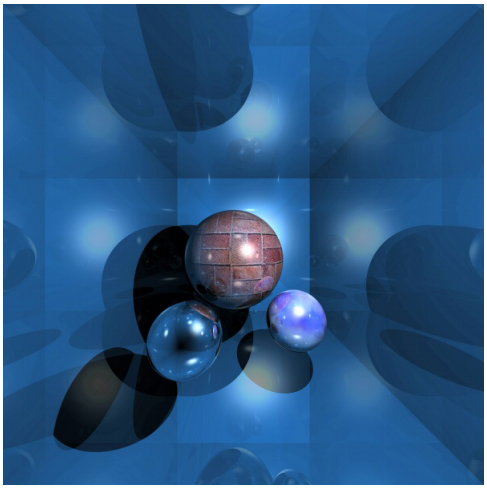


図 7.5: レイトレーシング法による 3D CG 画像例（ウィキペディア「レイトレーシング」（日本語版）の図から転載））

このような CG 画像はラスタライズ法では作ることができません。

　レイトレーシング法の欠点は計算量が膨大になることです。しかしコンピュータの計算能力は年々増大しており、レイトレーシング法による画像作成時間は年々短縮されています。GPU を使ったレイトレーシングも可能になっています。結果として、より高品位でリアルな CG 画像が作成できるレイトレーシング法の需要が高まっています。

— 以上