第1章 はじめに

1.1 講義概要

1.1.1 何を学ぶのか

三次元コンピュータグラフィックス (3DCG) の基礎理論を習得することを目的に、3D ゲームの映像生成に用いられるリアルタイムレンダリングの各種の手法について、OpenGL および GLSL (OpenGL Shading Language) による実装をもとに学習します。

1.1.2 何の役に立つのか

3D ゲーム開発に必要となる、グラフィックスプログラミングの基本的なスキルを身に付けることを目標とします。しかし、その下地である 3DCG の理論や、そこで用いられている数学や物理の基礎知識、各種のアルゴリズムやプログラミングテクニック、数値計算法などは、一般的なコンピュータのプログラミングにおいても有用です。これらの学習を通じて、コンピュータのプログラミングが少しだけ「うまく」できるようになることも狙います。

1.1.3 単位認定

課題と期末試験で評価します。課題は毎回の講義内容をもとにプログラミングを行うものを課します。個々の課題の評価は、未提出: 0 点、期限遅れ: 2 点、期限内に提出: 3 点、期限内に課題を達成: 4 点、高評価 (拡張課題の達成等): 5 点とします。成績は課題の評価の合計を 40%、期末試験の点数を 60%の割合で加算して求めます。

プログラミングのスキルは、与えられた問題に対して解決方法を考案し、実際にコードを書くことを繰り返すことによって向上します。そのために、課題は自分でコードを書いたかどうかに注目して採点します。また、期末試験では自分でプログラムが書けないと解答できない問題を用意します。

1.2 ゲームグラフィックス

1.2.1 ゲームと 3DCG

ゲームグラフィックスとは、言うまでもなく、ゲームに使われるグラフィックスのことをいいます。これには写真やイラスト、図版、テクスチャ、およびアニメーションなどの映像素材があります。ゲームグラフィックスが一般的な映像素材と異なる点は、これらの中に、コンピュータによって動的に生成されるものが含まれるところにあります。

コンピュータによって作成されたグラフィックスや、そのための技術を、コンピュータグラフィックス (CG) といいます。その意味では、ゲームグラフィックスは CG にほかなりません。

ただ、CG には映画の映像のように、あらかじめ時間をかけて生成されるものも含まれます。これに対してゲームグラフィックスでは、プレイヤーの操作に応じて映像を「その場」で生成する必要があります。このような CG をリアルタイム (実時間) CG といいます。特に、最近のゲームはフィールドを三次元で表現するものが主流になっていますから、ゲームグラフィックスの技術は、リアルタイム 3DCG の技術そのものであると言えます。

1.2.2 リアルタイム 3DCG

リアルタイム 3DCG と言っても、コンピュータによる映像生成に全く時間がかからないわけではありません。確かに、プレイヤーがゲームに没入するには、プレイヤーの操作に画面表示が遅れなく追従している必要があります。しかし、人間の視覚は一定より短い時間間隔で発生する情景の変化を知覚することができません。したがって、人間が変化を知覚できない時間間隔の上限が、リアルタイム 3DCG を生成する際の制限時間になります。

映像生成にかかる時間がこの制限時間を超えてしまうと、ゲームのユーザ体験が大きく損なわれてしまいます。そのためゲームグラフィックスでは、映像の品質よりも処理時間が優先されます。とは言っても、映像の品質自体もゲームの価値を構成します。したがって、この制限時間内で最大の映像品質を目指すことが、ゲームグラフィックスにおける技術開発の目標になります。

コンピュータによる映像生成は、以前は主としてソフトウェアで行われていました。これは映像生成の手順が複雑で、それを実行できるグラフィックスハードウェアの開発が困難であるか、開発できたとしてもコストが非常に高かったことが原因でした。しかしソフトウェアによる映像生成は、一般に専用のグラフィックスハードウェアを用いるより時間がかかります。そのため、制限時間内に映像生成を完了することと引き換えに、映像の品質を犠牲にしなければならないこともありました。また、映像生成に時間を取られることによって、物理シミュレーションなどの映像生成以外の処理に制限が加わることもありました。

コンピュータのハードウェア技術が進み、映像生成専用のグラフィックスハードウェアが低価格で実現できるようになって、リアルタイム 3DCG は日常生活でも一般的なものになりました。現在のリアルタイム 3DCG 技術は、このようなグラフィックスハードウェアの支援を用いることを前提にしています。このため現在のリアルタイム CG の技術開発では、いかにグラフィックスハードウェアの機能を活用して映像生成の速度や品質を向上するかが、目標のひとつになっています。また、そのような技術開発の蓄積をもとに、新たなグラフィックスハードウェアの機能を要求し、開発することも継続的に行われています。ソフトウェアの技術開発とハードウェアの技術開発は、リアルタイム 3DCG 技術の進歩の両輪なのです。

1.2.3 なぜゲームグラフィックスなのか

コンピュータサイエンスの中で、3DCG は人気のある分野のひとつです。しかし現実には、3DCG の応用分野は、これまでそれほど広いとは考えられていませんでした。特に、生成される映像のリアルさと引き換えに時間を潤沢に使えるような応用は、映画などの限られた分野にしか存在しないと思われてきました。また、これは 3DCG が応用されている分野として一般的に知られている映像制作やゲームがエンターテイメントの領域にあり、「ものづくり」第一主義である日本の産業の中では傍流にあると見なされていたということもあったのかもしれません。

しかし、「ものづくり」の中でも、CAD (コンピュータ支援設計) や CAE (コンピュータ支援工学) などの分野では、3DCG は非常に重要な技術です。それに、現在のパソコンのユーザインタフェースの部分でも、内部的に 3DCG の技術が使われています。ユーザの操作に対して即座に応答を返すというリアルタイム 3DCG の特性は、手元での対話的処理が重視されるであろう今後のコンピュータのユーザインタフェース技術の核となるものです。

そもそも、CG 技術の原点である Sutherland の Sketchpad は、コンピュータと人間との図形による対話 (interaction) を目指したシステムでした。このような CG はインタラクティブコンピュータグラフィックスと呼ばれ、CG 本来の目的でもあります。この点でもゲームグラフィックスは、これからのコンピュータの応用技術の基盤となるものだと考えています。

1.3 リアルタイム 3DCG の技術開発

1.3.1 グラフィックスハードウェアと CG 技術

近年のグラフィックスハードウェアの性能向上と価格低下には目を見張るものがあります。現在ではグラフィックスハードウェアは、パソコンに限らずスマートフォンやタブレット端末などのモバイルデバイスまで、ありとあらゆる情報機器に搭載されています。さまざまな局面でリアルタイム 3DCG が利用可能になった結果、AR (拡張現実感) を応用したものなど、これまでの枠にとらわれないグラフィックスのアプリケーションが提案されるようになりました。

また、グラフィックスハードウェアの機能自体も近年大きく向上し、できることも増えました。特に実数演算ハードウェアを搭載した GPU (Graphics Processing Unit) の登場と、そのプログラマブル化は、新技術をハードウェア処理の高速性を損なうことなくソフトウェア的に実装することを可能にしました。このことはグラフィックスハードウェアを使いこなすための技術開発の重要性を高めたと同時に、グラフィックスハードウェアをグラフィックス以外の汎用的な処理に活用する、いわゆる GPGPU (General Purpose GPU) への道を開きました。

1.3.2 技術開発のステップ

リアルタイム 3DCG といえども、用いられるソフトウェア開発の技法は、他の一般的なソフトウェアと変わりはありません。強いて重視すべき点を挙げるとすれば、それがユーザに何らかの体験を与える、ひとつの表現につながるものである、というところでしょうか。そして、その表現のためには、表現を生み出すもととなる対象や現象のモデル化が不可欠になります (図 1)。

ところがこのモデル化は、仮に自然現象や物理現象の再現を目的としたものであっても、ゲームグラフィックスは一般的な科学技術計算とは異なり、必ずしも「真の解」を求めようとするものではありません。求められているのは表現のための手法であるため、デザイナが現象の制御を

行える仕組みを用意しなければならない場合もあります。

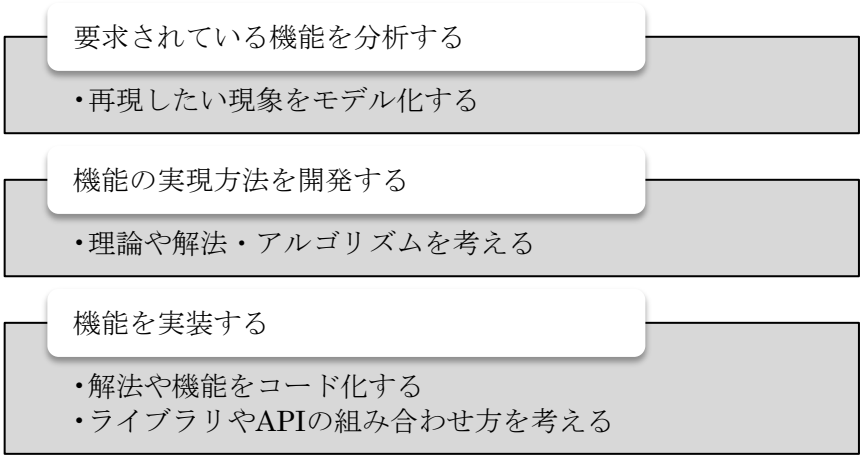

図 1 CG の技術開発のステップ

1.3.3 ゲーム開発に必要な知識

ゲームや CG の技術開発は、その目的が表現であるために、非常に多くの要素を含みます。空間や立体形状を取り扱うという点では、幾何学や線形代数などの数学の領域の知識が必要になります。現象のモデル化には、微分方程式が必須です。照明や運動を取り扱うには、物理の知識が必要になります。さらに、ゲームの対戦相手は人間に限りませんから、人工知能の技術も利用されます。もちろん、これらをソフトウェアとして実装するには、アルゴリズムの知識も必要になります。そして、このように多様な要素を持つ大規模なソフトウェアを開発する場合には、ソフトウェアエンジニアリングの成果も利用すべきでしょう。

また、ゲームの中には、ゲームのシナリオをデータとして内包するものもありますが、そのために専用のスクリプト言語 (プログラミング言語) を用意する場合もあります。グラフィックスハードウェアを効果的に制御するには、そのためのグラフィックスライブラリやミドルウェアを活用する知識が必要になるだけでなく、その特性に合わせた特殊なプログラミング技術が必要になる場合もあります。また、最近のゲームはネットワークと組み合わせてサービスを展開していますから、ネットワーク技術やそのためのシステムに関する知識も要求されます。

- 数学
- 物理学
- 人工知能
- アルゴリズム
- プログラム開発技法
- 言語処理系の実装の知識
- グラフィックスライブラリ/ ミドルウェア
- システムプログラミング / ネットワークプログラミング
- そして、もちろん英語 (最新の技術情報は英語で入ってくる)

このように考えると、ゲームプログラマは何でもできるスーパープログラマのように思えますが、実際、活躍している人達はそんな人達です。もちろん、大規模なソフトェアを一人で開発す

るようなことはありませんから、これらのすべてを完璧に習得しておかなければならないとは言いません。でも、新しいもの生み出すためには、基本的な知識は持っておくべきでしょう。

1.4 OpenGL

ハードウェア技術の向上によって、グラフィックスハードウェアは、パソコンの CPU に匹敵する、あるいはそれ以上の複雑さを持つようになりました。そのため、これを有効に活用するには、グラフィックス表示に要求される機能を整理し、グラフィックスハードウェアの機能を抽象化する、高機能なグラフィックスライブラリが必要になります。

グラフィックスライブラリの中には、コンピュータのハードウェア全体を制御するオペレーティングシステム (OS) の機能の一部として提供されるものがあります。アプリケーションプログラムはこれを介してグラフィックスハードウェアを制御するので、このようなグラフィックスライブラリは**アプリケーションプログラムインタフェース** (API) と呼ばれます。パソコンの OS として最も普及している Microsoft 社の Windows には、一般的な二次元のグラフィックス表示を行う Graphics Device Interface (GDI) や、リアルタイム 3DCG 表示機能を包含した DirectX (リアルタイム 3DCG 部分は Direct3D) というグラフィックス API が用意されています。

ところが Windows には、さらにもうひとつ別の OpenGL と呼ばれるグラフィックス API が用意されています。OpenGL は Microsoft 社が DirectX を用意する前から Windows に組み込まれていた、リアルタイム 3DCG に対応したグラフィックス API です。

OpenGL はもともと Silicon Graphics 社 (現在は SGI 社) が中心になって開発したグラフィックス API です。同社は IRIX と呼ばれる UNIX 系 OS を搭載した、エンジニアリングワークステーション (EWS) のメーカーです。OpenGL は同社の EWS のグラフィックス表示に用いられていたグラフィックスライブラリ (当時は GL と呼ばれていましたが、現在は OpenGL と区別するために IRIS GL と呼ばれることがあります) を、**プラットホーム** (ハードウェアや OS などのコンピュータの基盤) に依存する部分を分離して再実装したものです。

EWS は主に CAD などの技術的用途に用いられるコンピュータですが、パソコンが性能向上によりこの目的にも使用されるようなりました。その結果、EWS で動作していたアプリケーションをパソコンに移行するために、Windows 上にも OpenGL が移植されました。

当時のグラフィックスライブラリにはこれら以外のものもありましたが、パソコン用の OS の Windows による寡占化が進んだ結果、グラフィックスライブラリも DirectX と OpenGL の二つ以外のものは、実質的に淘汰されてしまいました。このうち DirectX は Microsoft 社の専有物のため、OS として Windows を採用しているパソコン以外で共通に使用できるグラフィックスライブラリは、現在では事実上 OpenGL しかありません。しかし、これは言い換えれば、Windows パソコン以外のコンピュータ関連機器のほとんどが、3DCG 用のグラフィックス API として OpenGL (または、組み込み機器向けの OpenGL ES) を採用していることになります。

OpenGL がこのように広く使われるようになった背景には、もちろん OpenGL しか選択肢がなかったということもありますが、その仕様がプラットホームとは独立していることも大きな理由だと考えられます。そのため、OpenGL はさまざまな機器に導入されました。Apple 社のパソコンの OS である Mac OS X や、UNIX あるいは Linux の画面表示に用いられている X Window System でも OpenGL が使われています。またパソコンに限らず、スマートフォンの OS である Apple 社の iOS や Google 社の Android も、OpenGL ES を採用しています。

1.5 GLFW

1.5.1 OpenGL とツールキット

OpenGL はプラットホームに依存しないグラフィックス API ですが、アプリケーションプログラムからこの機能を使用するためには、やはりプラットホームごとに異なる手順で「お膳立て」をする必要があります。しかし、その部分の実装はそれなりに面倒なものになるため、それをうまく包み隠して簡単に使えるようにしたツールキットがいくつも提案されています。

中でも GLUT (OpenGL Utility Toolkit) は、OpenGL を開発した Silicon Graphics のエンジニア (当時) が作った、使いやすいツールキットです。また GLUT はマルチプラットホームに対応しているため、これを使ったソースプログラムは Unix / Linux、Windows、Mac OS X の間で共通にすることができます。GLUT は OpenGL の初期の頃に作られたものですが、OpenGL の学習や OpenGL を使った簡単なプログラムの作成を手軽に始めることができるため、今でも有用なツールキットとして利用されています。

しかし、オリジナルの GLUT は既に長い間メンテナンスされていません。代わりに GLUT 互換の freeglut や OpenGLUT というツールキットが開発されていますが、これらは本稿の執筆時点では Mac OS X には対応していません。また、Mac OS X には以前から標準で GLUT が搭載されていましたが、これは OpenGL 2.1 にしか対応しておらず、Mac OS X のバージョン 10.7 (Lion) 以降で使用可能になった OpenGL 3.2 Core Profile や、10.9 (Mavericks) 以降で使用可能になった OpenGL 4.1 を (公式には) 使用することができません 。加えて 10.9 では、ついに GLUT の使用自体が非推奨になりました。

1.5.2 GLFW というツールキット

OpenGL に対応していて GLUT のようにマルチプラットホームで使用できるツールキットには、FLTK や Qt、SDL などがあります。中でも Qt は非常に高機能なツールキットであり、3DCG 関連のいくつかの主要なアプリケーションがこれを使って開発されています。しかし、OpenGL の学習のための短いプログラムを作成するには、Qt はちょっと規模が大きすぎる気がします。そこで GLUT の代わりに使える簡単で小さなツールキットとして GLFW があります。

GLFW のプロジェクトのホームページ (http://www.glfw.org/) には、「GLFW はウィンドウを作成し、OpenGL のコンテキストを作って、入力 (デバイス) を管理する、無料の、オープンソースの、マルチプラットホームのライブラリである」と書いてあります。ライセンスは zlib/libpng license を採用しています[1]。

1.5.3 GLFW の特徴

GLFW は、次のような特徴を持っています。

- **マルチプラットホームである**

  GLUT と同様に Windows / Mac OS X / Linux でソースを共通化することができます。

[1] GLFW の開発者である Marcus Geelnard 氏、Camilla Berglund 氏、及びコミュニティの皆様に感謝致します。

● OpenGL のバージョンやプロファイルが指定できる

Mac OS X バージョン 10.7 (Lion) 以降では、OpenGL のバージョン 3.2 の Core Profile を指定することができます。

● コールバックベースではない

イベントループを自分で書くことができます。GLUT と同様なコールバック形式で書くこともできます。

● 最初からダブルバッファリングになっている

明示的にイベントを待つようにプログラムを書かなければイベントループが回り続けるので、アニメーション表示が基本になります。またイベントの取得はダブルバッファリングのバッファの入れ替えの際に行われるため、コールバック形式でプログラミングする場合もイベントループは自分で用意しておく必要があります。

● 入力デバイスの取り扱い方法が異なる

キーボードからの入力は GLUT のように文字として得られるほか、押されているキーそのものを知ることもできます。また、マウスホイールやジョイスティックのデータを取得することもできます。

● その他の機能

このほか、GLUT ではちょっとネックになっていたテクスチャ用の画像を読み込む機能が用意されています。また、スレッド (並行処理) やミューテックス (同期機構) が用意されているため、他のデバイスからのデータ入力やサウンドとの組み合わせが容易になります。それともうひとつ、Mac OS X の GLUT では無効になっていた、ウィンドウのクローズボタンが有効になります。

● GLFW にない機能

一方、GLUT にあって GLFW にない機能もいくつかあります。例えば、Cube や Sphere、Teapot のような図形を表示する機能は省かれています。またビットマップフォントをレンダリングする機能もありません。ポップアップメニューを表示する機能も用意されていません。

最初からダブルバッファリングになっていることや、マウスやキーボード、ジョイスティックの扱い方を見ると、GLFW は GLUT に比べて、かなりゲーム向きに作られているように思われます。また、GLUT にあって GLFW に無い機能の多くは、OpenGL の Core Profile では使えない機能を使っています。Mac OS X の GLUT が OpenGL の Legacy Profile、すなわち OpenGL 2.1 にしか対応していないのは、このような問題があるからかも知れません。

● ドキュメント

GLFW に添付されている Reference や User Guide は非常にわかりやすいので、ぜひ目を通してください。

第2章 レンダリングパイプライン

2.1 グラフィックスライブラリ

2.1.1 セマンティックギャップとアルゴリズム

パソコンはいろんな仕事ができる便利な道具ですが、今のところユーザが期待していることを自動的にやってくれるほど賢い道具にはなっていません。ユーザは目的の仕事を達成するために、パソコンに用意されている機能を使った仕事の手順を考える必要があります。このように、ユーザが「やりたいこと」と、コンピュータが「できること」の間には、意味的なギャップが存在します。このようなギャップはセマンティックギャップと呼ばれます。

コンピュータのアプリケーションプログラムとコンピュータのハードウェアとの間にも、同じようなセマンティックギャップが存在します (図 2)。例えば、グラフィックス系のアプリケーションプログラムにおいて、画面上に線を引くニーズがあったとします。これに対して、コンピュータのハードウェアが画面上に点を打つ機能しか持っていなければ、そのままでは線を引くことはできません。そこで、点を打つ機能を組み合わせて線分を描く必要があります。この「点を組み合わせて線分を描く方法」がアルゴリズムです (Bresenham のアルゴリズムなど)。

当初、このアルゴリズムは、すべてアプリケーションプログラム内に組み込まれていました。しかし、ひとつのコンピュータ上で複数のアプリケーションプログラムを使用すると、この方法では、同じ目的のプログラムコードが、アプリケーションごとに重複して存在してしまいます。これはメモリなどのコンピュータのリソース上も無駄ですし、ソフトウェア開発の面でも、同じ目的の異なるプログラムコードを作成することになって、コストの上昇を招きます。

そこで、アプリケーションプログラムがグラフィックスハードウェアに要求する機能を整理して、それをアプリケーションプログラムから分離するということが行われました。このアプリケーションプログラムが要求する機能を実現するグラフィックスアルゴリズムを集めたソフトウェアの層が、グラフィックスライブラリです (図 3)。

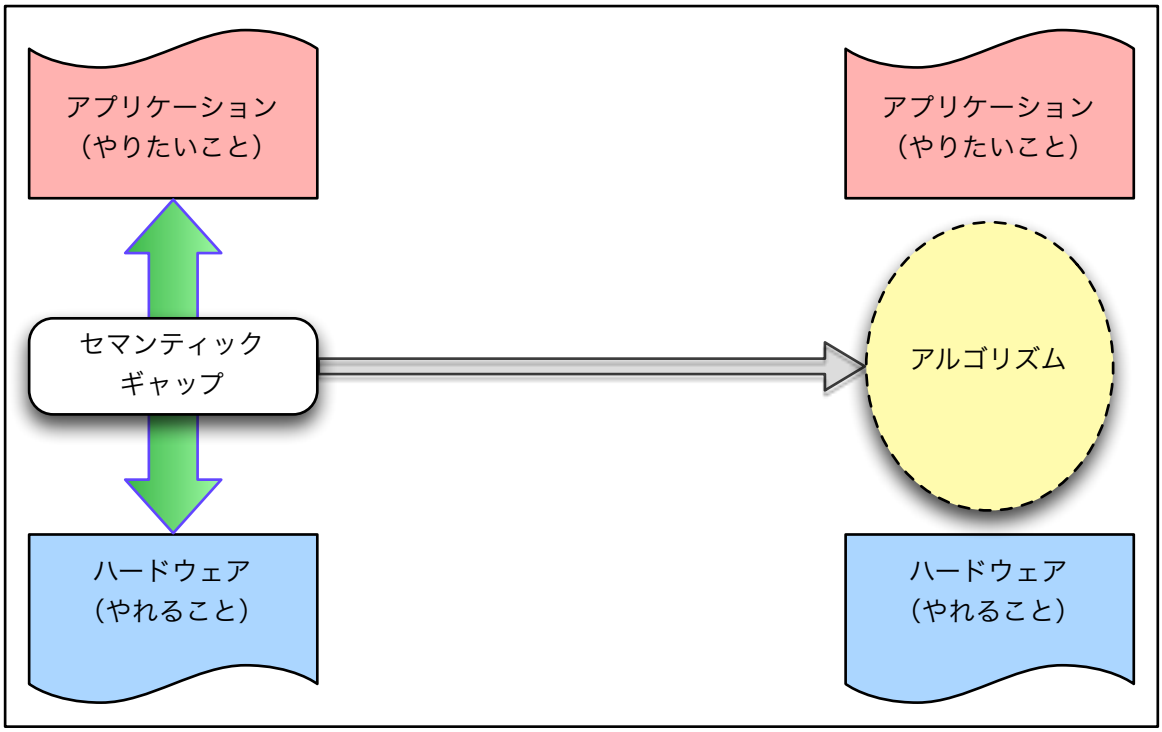

図 2 セマンティックギャップとアルゴリズム

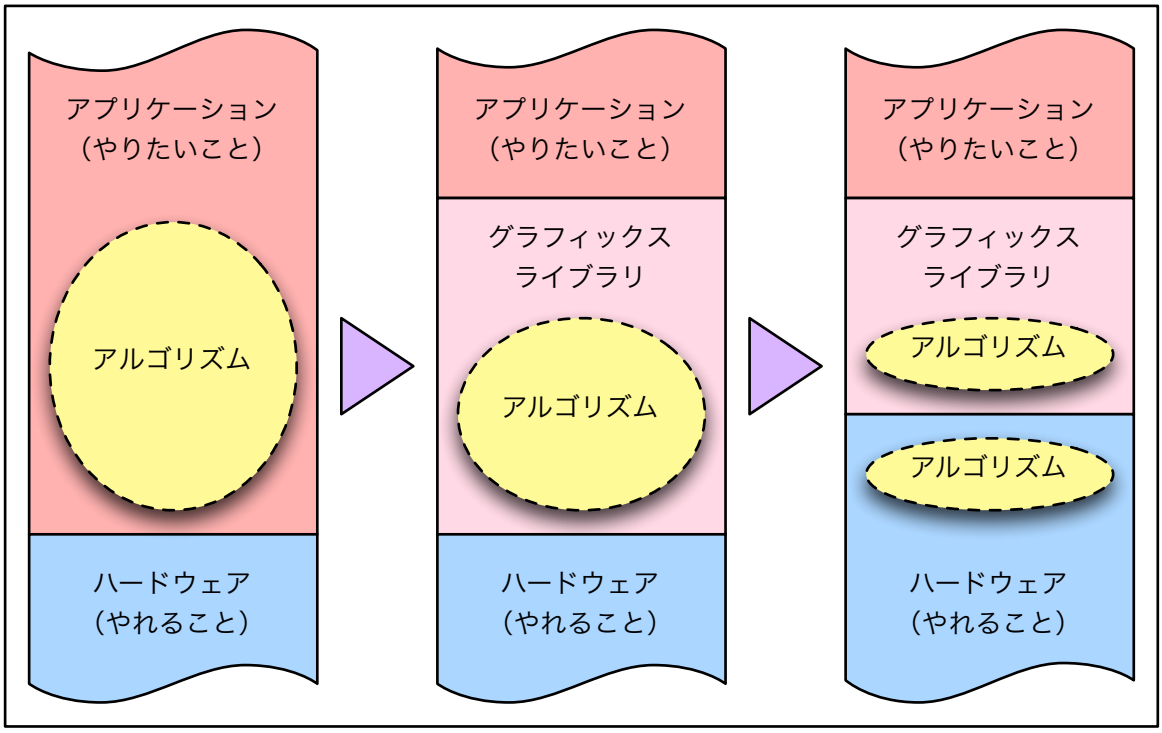

図 3 アルゴリズムの実装場所

なお、前述の「線分を描く方法」のような簡単なアルゴリズムは、実際にはほとんどのグラフィックスハードウェアに**固定機能**として実装されています。またハードウェア技術の向上により、近年はより高度な機能がグラフィックスハードウェアに実装されています。すなわち、グラフィックスハードウェア自体にもアルゴリズムが組み込まれているのです。

2.1.2 グラフィックスライブラリの二つの役割

このようにグラフィックスライブラリは、アプリケーションプログラムとグラフィックスハードウェアの仲立ちとして、アプリケーションプログラムにグラフィックスの機能を提供します。この役割には、次の二つがあります。

● 標準的なグラフィックスアルゴリズムを提供する

グラフィックスの機能をアプリケーションプログラムから分離することによって、次のようなメリットが生まれます。

- ソフトウェア開発の手間を減じる
- ソフトウェアのポータビリティを向上する

このようなグラフィックスライブラリは、ACM CORE、ISO GKS、GKS-3D、PHIGS、PHIGS+など数多くのものが提案されてきており、また標準化も試みられました。しかし、現在ではこのような形式のグラフィックスライブラリは、あまり使われなくなりました。これは技術の進歩により要求されるグラフィックスの機能が多様化し、単一のグラフィックスライブラリの層ではそれに対応しきれなかったことも、理由のひとつだと思います。

● グラフィックスハードウェアの機能を呼び出す方法を提供する

グラフィックスハードウェアの高機能化にともなって、グラフィックスライブラリはその複雑な機能を抽象化してアプリケーションプログラムに提供する、インタフェースの役割 (API) を担うようになりました。

- グラフィックスハードウェアの機能を抽象化する
- アプリケーションとのインタフェース (API) を提供する

現在の代表的なグラフィックスライブラリである OpenGL や DirectX は、そのような性格を持つものです。これらも複雑なアルゴリズムをソフトウェアで実装した、高度な機能を提供していますが、今後そのような機能は別の層 (ミドルウェア等) に移管され、グラフィックスライブラリ自体はハードウェアの機能の抽象化に特化する方向にあると考えられます (図 4)。

2.1.3 GPU におけるグラフィックスライブラリ

プログラマブルなグラフィックスハードウェアとなった現在の GPU では、アプリケーションプログラムからグラフィックスハードウェアの**プログラマブルシェーダ**上にプログラムを導入して、必要な機能を実現します。このプログラムを**シェーダプログラム**といいます。この場合、グラフィックスの機能は、CPU 側のプログラムとシェーダプログラム、そしてグラフィックスハードウェアの固定機能の組み合わせで実現されます。

このため、このグライックスライブラリは、アプリケーションプログラムに対してグラフィックスの機能を提供するだけでなく、シェーダプログラムや GPU 上のメモリ、演算機能などのリソースを管理する、OS のような役割を果たします。

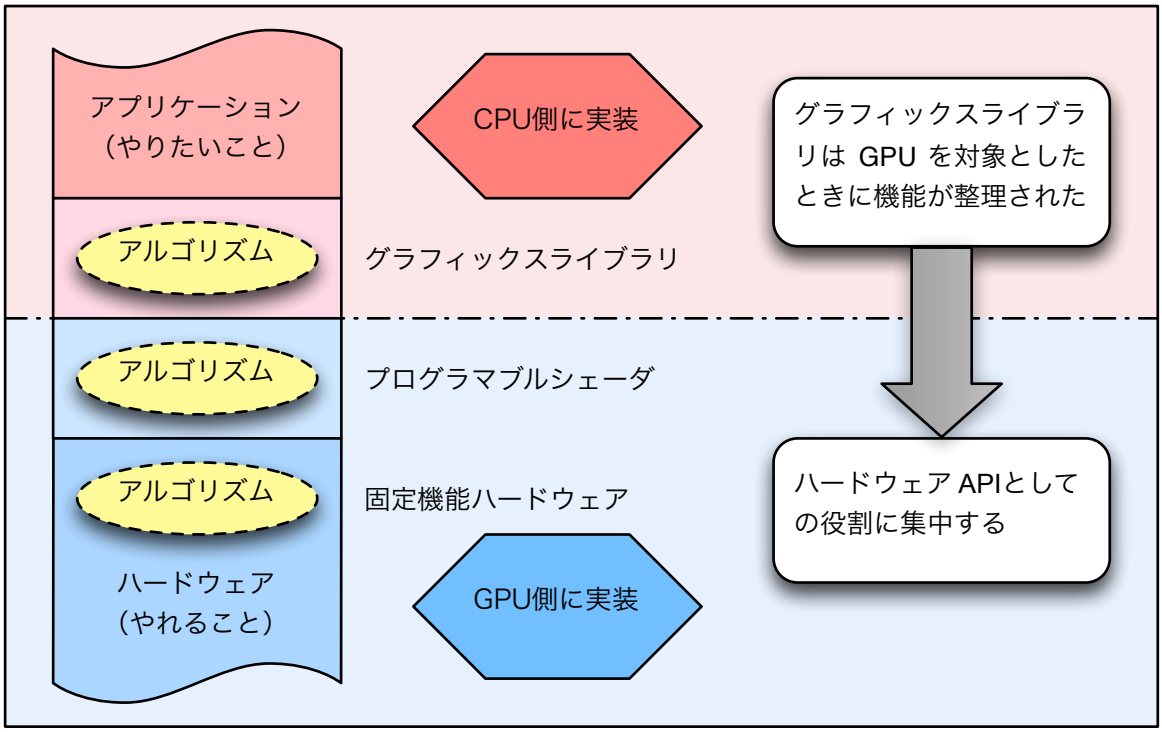

図 4 GPU におけるグラフィックスライブラリ

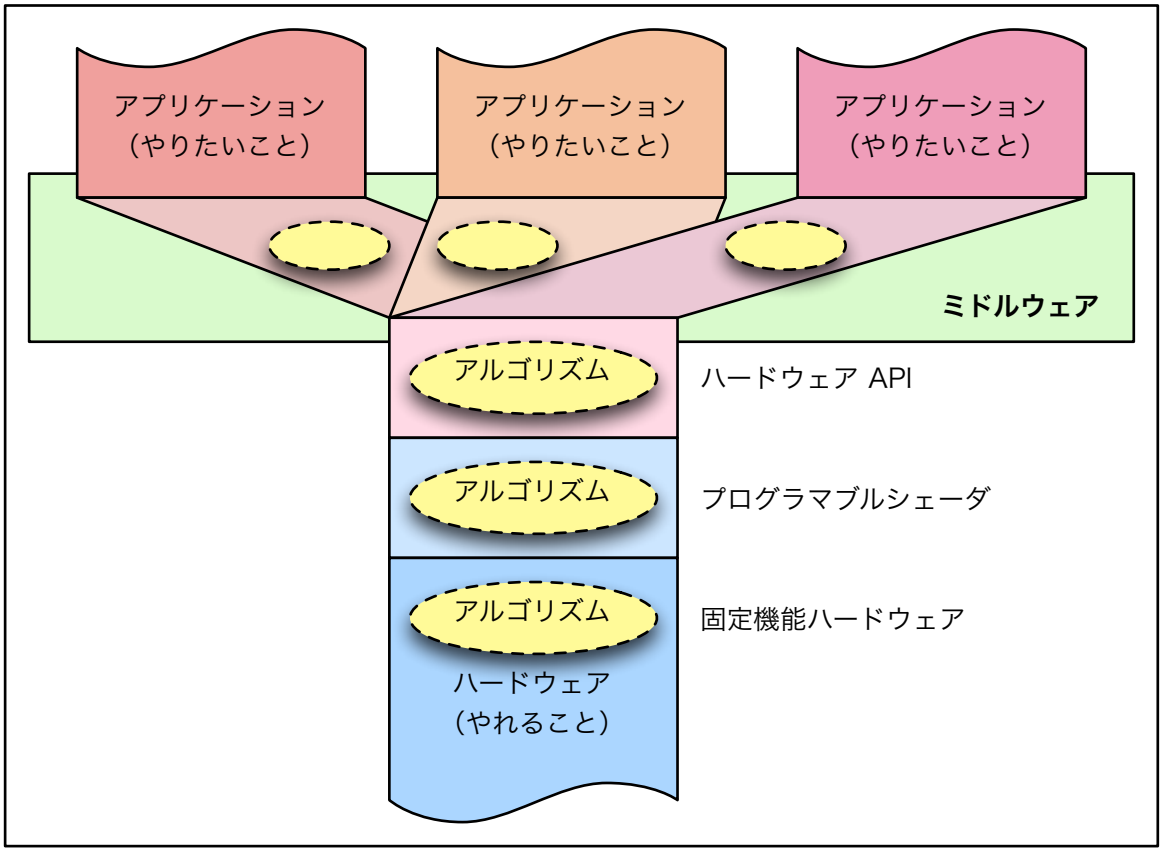

図 5 多様なアプリケーションとミドルウェア

2.1.4　ミドルウェア

グラフィックスライブラリがグラフィックスハードウェアの機能の呼び出しに特化する一方で、より高度な機能は、**ミドルウェア**と呼ばれる上位の層に移管されるようなりました。これを含めてグラフィックスライブラリとしてとらえることもできます。この層はアプリケーションプログラムの要求に合わせて、ミドルウェアごとに異なる機能を提供します (図 5)。

たとえば、CG のシーンを構成するデータ (オブジェクト) の階層的な管理に用いられるシーングラフの操作とその映像生成を行う**シーングラフ API** もそのひとつで、初期の OpenInventor をはじめ、OpenSceneGraph や SceniX などがあります。また、ゲームに使われる機能を集めたものは**ゲームエンジン**と呼ばれ、CryENGINE、Unreal Engine、Unity、OROCHI、MascotCapsule、Irrlicht などの総合的なもの、chidori、DAIKOKU、OGRE などの映像生成主体のもの、BISHAMON、YEBIS などのエフェクト生成用のもの、CRI-ADX2 のようなサウンド用のもの、それに衝突などの物理シミュレーションを行う PhysX や Havok、ODE、Bullet などの**物理エンジン**など、目的に応じて様々なものがあります。

現在のゲーム開発に使われている各種の技術は非常に高度になってきているため、個々のゲームメーカーが独自に対応することが困難になりつつあります。このようなミドルウェアの特徴や特性を知り、うまく組み合わせて使うことも、現在のゲーム開発では重要になっています。

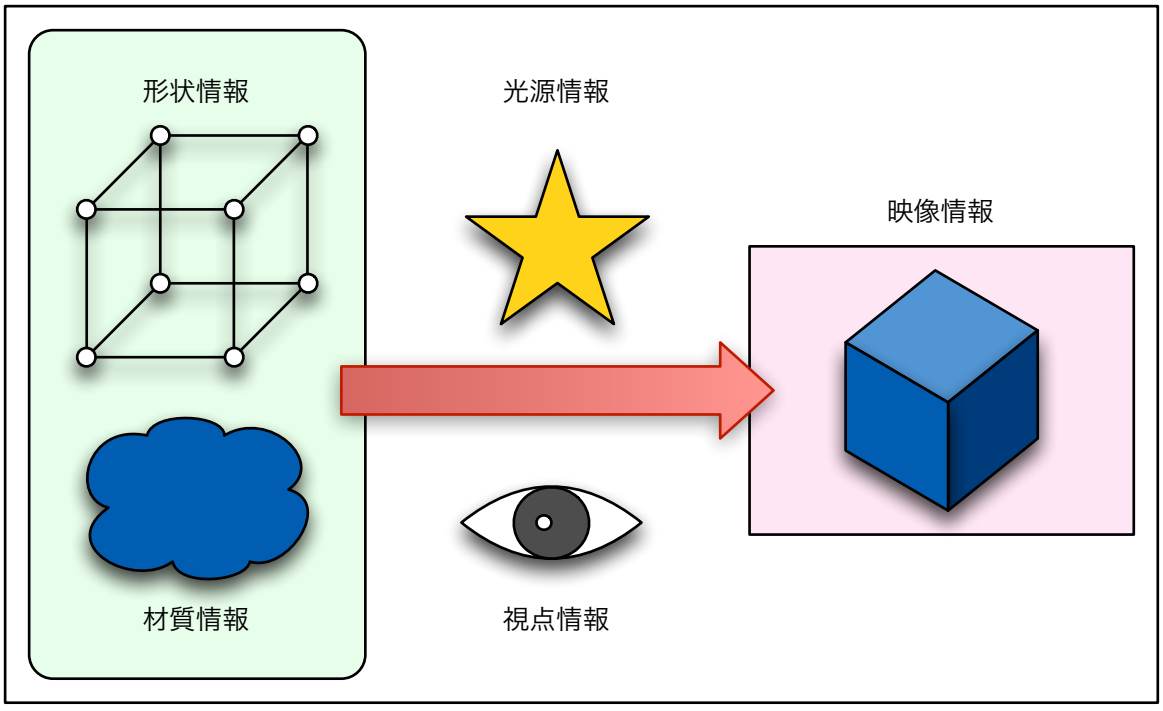

図 6 レンダリングのイメージ

2.2　レンダリング

2.2.1　3DCG におけるレンダリング

デザイン用語としての**レンダリング**はデザイナが自分のイメージを可視化することをいいますが、3DCG ではコンピュータによってシーンのデータから映像を生成することを示します。シ

ーンのデータは、配置する物体の形状情報や、その表面の材質情報、それにシーン中の光源に関する情報、視点あるいはカメラの位置や方向などの視点情報などから構成されます。これらをもとに、コンピュータは視点位置から見たシーンの映像を生成します (図 6)。

2.2.2 レンダリングの二つの方向

現在用いられているレンダリング手法は、**サンプリング**による方法と**ラスタライズ**による方法の二つに大別できます。

● **サンプリングによる方法**

これは視点を出発してスクリーン上の一点を通る半直線、すなわち視線と物体との交点を求め、交点における陰影を求めてスクリーン上の点の色とする方法です (図 7)。レイキャスティング法やレイトレーシング法がこれに分類されます。この方法は計算のモデルが単純で複雑な光学現象の再現が行いやすいため、高品質な映像生成を行う場合によく用いられます。しかし、一般に計算量が多く、高速な映像生成には向かないと考えられています。

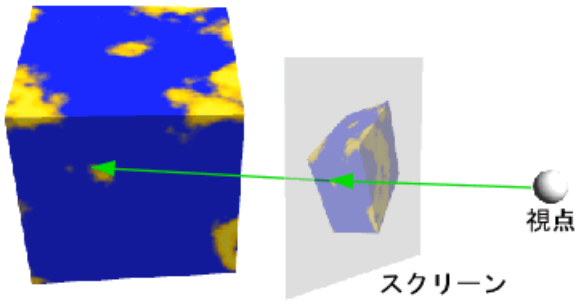

図 7 サンプリングによるレンダリング

● **ラスタライズによる方法**

これは物体のスクリーン上への投影像を求め、その投影像の領域をラスタライズにより塗り潰す方法です。スキャンライン法やデプスバッファ法がこれに分類されます。この方法により描画される形状は、主にスクリーンへの投影や塗りつぶしが簡単な多面体になります。また、複雑な光学現象の再現には向かないため、生成される映像はリアルさに欠ける場合があります。しかしハードウェアによる実装が行われており、高速な映像生成が目的の場合には、一般的にこの方法が用いられます (図 8)。

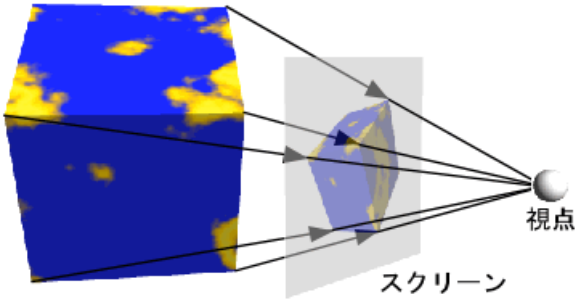

図 8 ラスタライズによるレンダリング

映画のように高品質な映像の生成には主としてサンプリングによる方法が用いられており、これは現在でも時間がかかります。処理時間を短縮するために計算のモデルなどを単純化すれば、得られる映像の品質が損なわれます。処理時間と品質は**トレードオフ**の関係にあります。

しかし現在では、これら二つの方法は互いに近づきつつあります。例えば、既にプログラマブルな GPU を用いてレイトレーシングをリアルタイムに実現する手法が提案されています。その一方で、デプスバッファ法により高品質な映像を生成する手法も数多く提案されています。

2.2.3 リアルタイムレンダリング

リアルタイムレンダリングは、レンダリングしようとするシーンの変化に対して、遅れることなく映像を生成することをいいます。ゲームや CAD システムなどの場合、現在の画面表示に対するユーザの反応 (コントローラやマウスの操作など) によりシーンのデータが変更され、それが次の瞬間の画面表示に反映されます。このとき、この「反応」に対する「表示」のサイクルが十分高速であれば、ユーザはゲームに対して没入感を得ることができ、CAD システムに対して快適に対話的操作を行うことができるようになります。

コンピュータはディスプレイの画面への表示を連続的に行っているわけではなく、静止画を一定の時間間隔で書き換えています。この一枚の静止画を**フレーム**といい、その一秒間あたりの描き替え回数を**リフレッシュレート**といいます。リフレッシュレートの単位には、周波数と同じ Hz (Hertz) が用いられます。この値はグラフィックスハードウェアとディスプレイとの関係によって決定される定数値です。これに対して、レンダリングによって一秒間に生成されるフレームの数を**フレームレート**といい、これには fps (frame per second) という単位が用いられます。リアルタイムレンダリングでは、一定以上のフレームレートを確保することが必要になります。

一般に動画として知覚するには、6 〜 8 fps 程度のフレームレートが必要だと言われます。しかし、これはかなりぎこちない表示になります。15 fps 以上あれば、表示はほぼ滑らかに変化するように感じられます。逆に、液晶などの一般的なフラットパネルディスプレイのリフレッシュレートは 60 Hz 程度のため、これを超えるフレームレートで映像を生成しても、すべてのフレームが表示されることはありません。したがって、リアルタイムレンダリングでは、通常は 60 fps のフレームレートを確保すれば十分です。

表 1 フレームレートと対話性

フレームレート	動きの知覚
1 fps	一枚一枚の画像が順に現れるように見えて対話的操作はほぼ不可能
8 fps	ぎこちないが動画として知覚されて対話的操作が可能になる
15 fps	動きはほぼ滑らかに感じられて対話的操作に違和感がない
60 fps	一般的なフラットパネルディスプレイのリフレッシュレート

しかし、リフレッシュレートが 60 Hz のフラットパネルディスプレイでは、速い動きのときに残像が見えることがあります。これを避けるために、リフレッシュレートを 120 Hz (倍速) や 240 Hz (四倍速) に高めたディスプレイも存在します。ゲームではコントローラの操作に対して画面の次の描き換えタイミングまで表示が遅れることを避けられないため、このようなディスプレイを用い、高いフレームレートで生成した映像の表示を行う場合もあります。

2.2.4 グラフィックスハードウェアの重要性

リアルタイムレンダリングでは、通常、三次元のシーンを対象に映像を生成します。そのためには、単に画面上の領域を塗りつぶすだけでなく、三次元の図形をスクリーンに投影して画面上の領域を求め、その領域の色を照明計算により求める必要があります。リアルな陰影を得るには精密な物理現象の再現が必要になるため、長い計算時間を要します。また、コントローラの操作の受け付けや動き (アニメーション) の生成なども、これらの処理と並行して行わなければなりません。衝突の処理や変形、物理シミュレーションにもとづく破壊や粒体の表現なども、時間のかかる処理になります。コンピュータの CPU の性能が向上した現在でも、これらのすべてを CPU 単独で処理して十分な対話性を達成するのは困難です。

したがって現在のリアルタイムレンダリングでは、グラフィックスの処理を専門に行うグラフィックスハードウェア、すなわち GPU の支援が不可欠だと考えられています。既に GPU は現在の PC に必須のものとなっています。また、ほとんどの 3DCG アプリケーションは、GPU の利用を前提としています。

また、近年はプログラマブルな GPU の登場により、より多様なグラフィックスの処理が可能になりました。また、物理計算などのグラフィックス以外の処理にも GPU が用いられるようになり、映像だけでなく、動きのリアリズムの向上にも大きく貢献するようになりました。

2.3 パイプライン

2.3.1 高速化手法としてのパイプライン処理

パイプラインは、もともと油田などから石油を運ぶ長い管のことをいいます。でもコンピュータでは、図 9 のようにひとつの処理をいくつかの段階 (ステージ) に分割して、処理を順送りすることをいいます。これにより同じ時間がかかる処理でも、分割した各段階を同時に動作させることによって、全体的な処理量 (throughput) を増加させることができます。

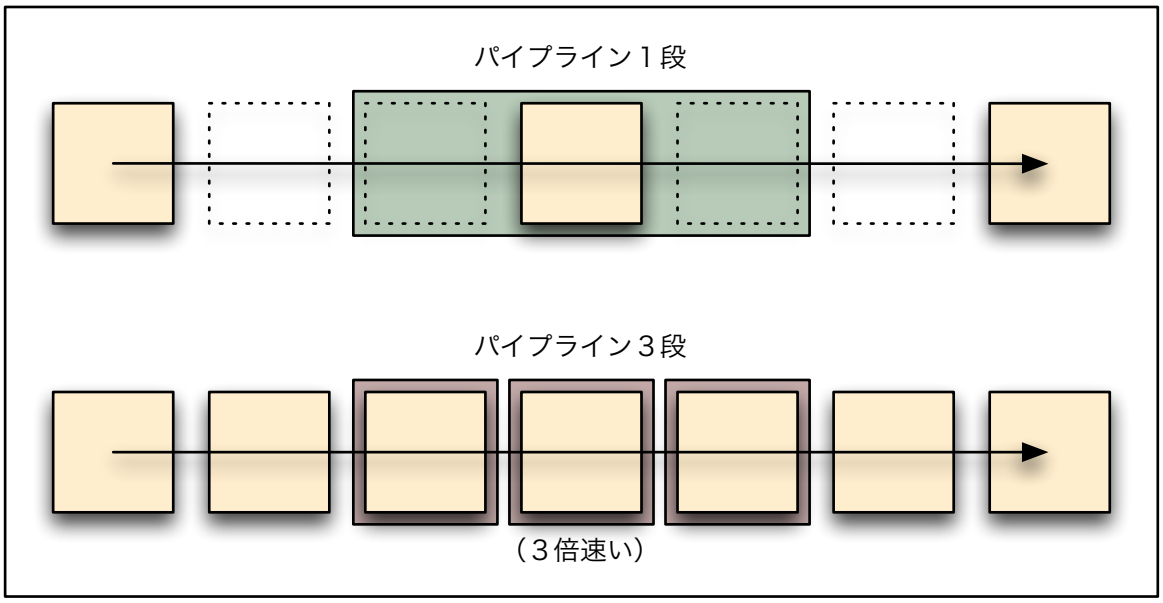

図 9 パイプラインの考え方

パイプライン処理は、各ステージの処理時間がすべて同じときに、最も高い性能が得られます。例えば、三段に分割すれば、三倍速くなります。しかし、もし一部のステージの処理時間が他のステージより長いと、そこで次の処理が待たされてしまい、処理を割り当てられないステージが発生してしまいます。これは全体の処理量を大きく低下させてしまいます。

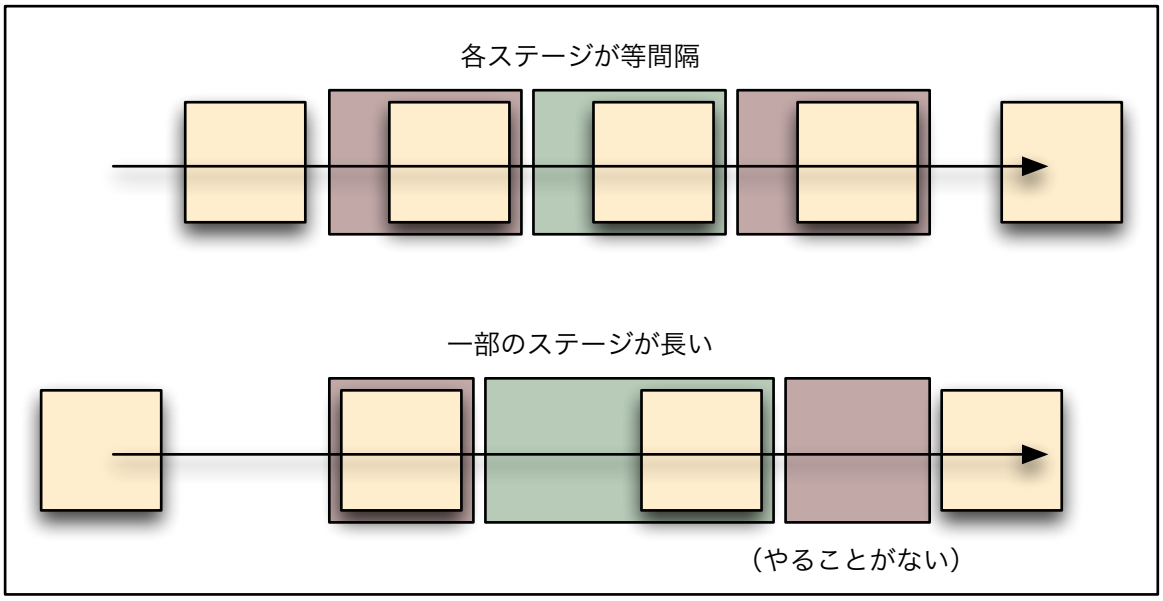

図 10 パイプラインの考え方

2.4 グラフィックス処理のパイプライン

2.4.1 グラフィックス処理の概念上のパイプライン

グラフィック処理の概念上のパイプラインは、**アプリケーション**、**ジオメトリ処理**、**フラグメント処理**の三つのステージで捉えることができます。

図 11 グラフィックス処理の概念上のパイプライン

2.4.2 アプリケーションステージ

ユーザが実際に使用するアプリケーションソフトウェアのステージで、CPU 側で実行され、グラフィックスハードウェアに**ジオメトリデータ** (形状情報) を送って画面表示の処理を依頼します。このステージはソフトウェアだけで構成されるため、開発者はすべてを制御することができます。また、ソフトウェアの実装を変更することで、動作を変更することができます。

ソフトウェアはこのステージで画面表示に必要なジオメトリデータを生成します。ジオメトリデータは点や線分、三角形といった**基本図形** (Rendering Primitive) の種類と、その頂点の位置や色、法線ベクトルなどの**頂点属性** (Attribute) で構成されます。これを次のステージであるジオメトリ処理に送ります。

アプリケーションステージでは、ユーザとの対話に用いるマウス、キーボード、ジョイスティック、位置センサ、イメージセンサなどのインタフェース機器からのデータの入力を管理します。表示している図形の衝突の検出などもこのステージで行われ、結果を画面表示に反映したり、フォース (力覚) フィードバック機能を持つインタフェース機器に返したりします。

また時間に関する処理、例えばアニメーションの生成におけるタイミングの制御なども、アプリケーションのステージでの仕事です。以前は時間に伴う変形などの処理もアプリケーションステージの仕事でしたが、現在の GPU は単純な座標変換によるアニメーションやスキニング (骨格に基づく変形)、ジオメトリモーフィング (個々の頂点の移動による変形) などは、ジオメトリ処理のステージで実行することが可能になっています。テクスチャアニメーションに関しても、あらかじめグラフィックスハードウェア側のメモリに転送しておいた複数のテクスチャを順次切り替える手法などを用いて実現することが可能になっています。

このほか、他の (グラフィックスハードウェア上の) ステージでは実現できないすべての種類の処理が、アプリケーションステージで実行されます。たとえば、グラフィクスハードウェアに送るジオメトリデータの量を減らす最適化処理などは、当然グラフィックスハードウェアでは実行できません。この処理には階層的な視錐台カリング (視野に含まれないものをあらかじめ除去する) やオクルージョンカリング (他の物体に隠されて見えないものをあらかじめ除去する) などがあります。なお、最も単純なカリングであるバックフェースカリング (視点に対して裏側を向いている背面ポリゴンの除去) はグラフィックスハードウェアに組み込まれています。

アプリケーションのステージにおける最適化処理の今後の課題は、現在主流となっているマルチコア CPU による並列処理に対応することでしょう。次のステージ、すなわちグラフィックスハードウェアのデータの入り口はひとつしかないため、並行して動作する複数の処理単位 (スレッド) から同時にデータを流し込むことができません。したがって、アプリケーションのステージの並列化は、現時点ではスレッドごとにカリング、衝突検出、物理シミュレーション、そしてジオメトリデータの送出などの異なる処理を実行することにより行われています。これはスレッドごとに処理内容が異なるために処理時間が揃いにくく、スレッド同士が互いに同期を取ることが困難になります。また、将来の CPU のメニィコア化への対応も今後の課題となっています。

2.4.3 ジオメトリ処理ステージ

アプリケーションから受け取ったジオメトリデータ、すなわち基本図形と頂点属性を処理するステージで、座標変換や頂点の陰影付け、クリッピングなどを行います。

固定機能の (すなわち、プログラマブルではない) グラフィックスハードウェアでは、このステージは複数の固定機能のステージに分割して実装されていました (図 12)。また、さらに性能を向上するために、このパイプラインを複数並列に動作させることも行われました (図 13)。

これは、このステージにおいて座標変換や陰影付けなどの多くの実数演算を伴う負荷の高い処理が実行されるためです。しかし実数演算のハードウェアのコストは高いため、これは初期のパソコン用のグラフィックスハードウェアには搭載されていませんでした。そのために、このステ

ージを CPU 側で実行する実装も行われていました。

その後、実数演算のハードウェアを搭載してジオメトリ処理を担当できるパソコン用のグラフィックスハードウェアが登場しました。これは GPU (Graphics Processing Unit) と名付けられました。また、ハードウェアによる座標変換や陰影付けの機能は、ハードウェア T & L (Transform and Lighting) と呼ばれました。

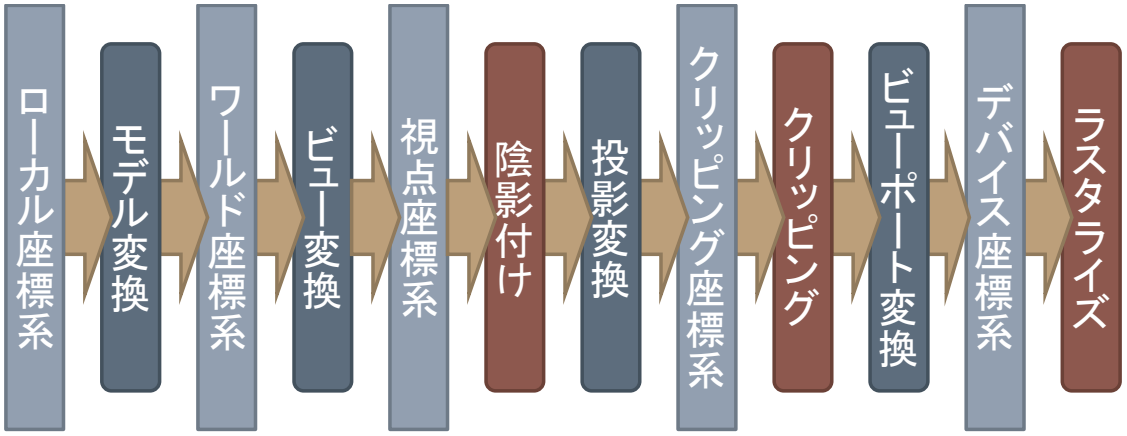

図 12 ジオメトリ処理のパイプライン

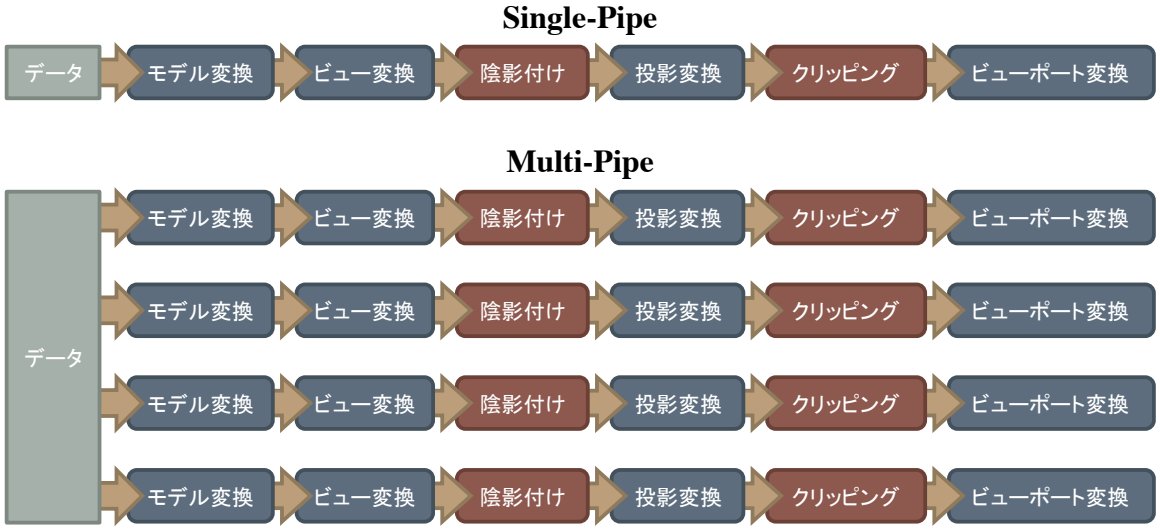

図 13 ジオメトリ処理のパイプラインの並列化

● **モデル変換**

シーンを構成する**オブジェクト** (部品図形) は、個々に独自の座標系で定義されています。これを**ローカル座標系**といいます。シーンを構築するには、このオブジェクトを視点や光源なども配置される、**ワールド座標系**と呼ばれる単一の座標系に配置します。このために行われるローカル座標系からワールド座標系への座標変換を**モデル変換**といいます (図 14)。

モデル変換はオブジェクトごとに設定します。また、オブジェクト間に骨格のような階層構造があれば、この変換も階層的に合成して実行します。モデル変換後は、すべてのオブジェクトがワールド座標系上に存在します。

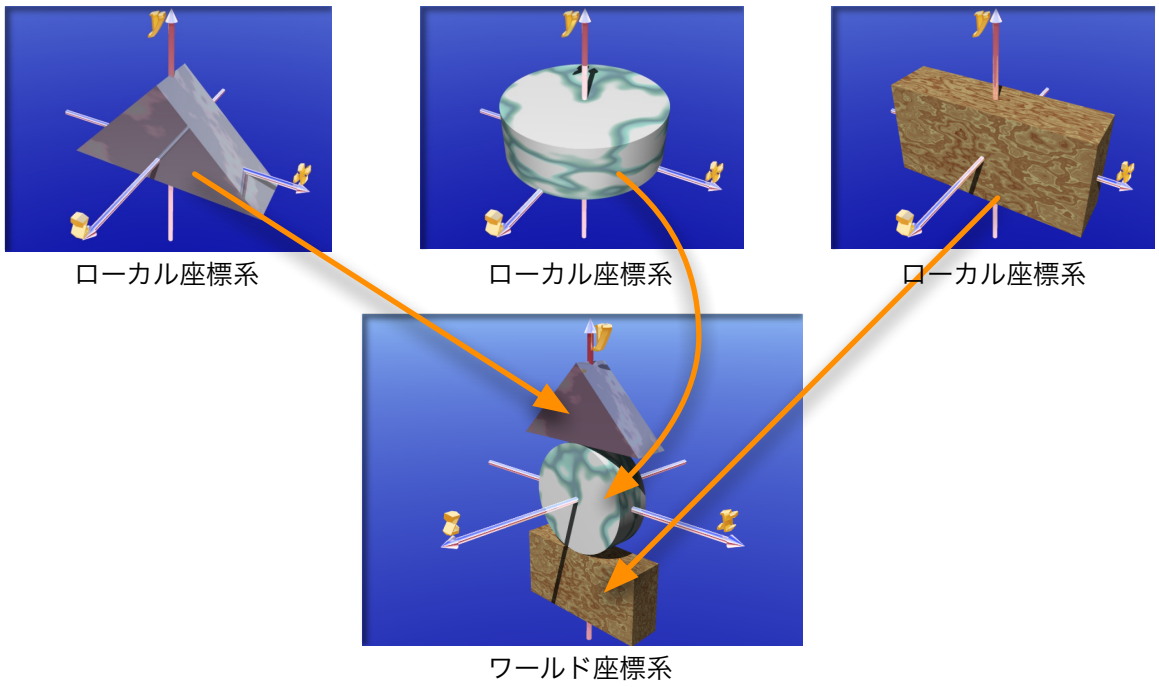

図 14 モデル変換

● **ビュー変換**

ワールド座標系で構築されたシーンの視点位置から見た映像を作成するために、視点を基準とする**視点座標系**にシーン全体を座標変換します。このために行われるワールド座標系から視点座標系への変換を**ビュー変換**といいます (図 15)。なお、モデル変換とビュー変換は通常ひとつの座標変換に合成されます。この合成変換を**モデルビュー変換**といいます。ローカル座標系、ワールド座標系、視点座標系、およびスクリーンの関係を、図 16 に示します。

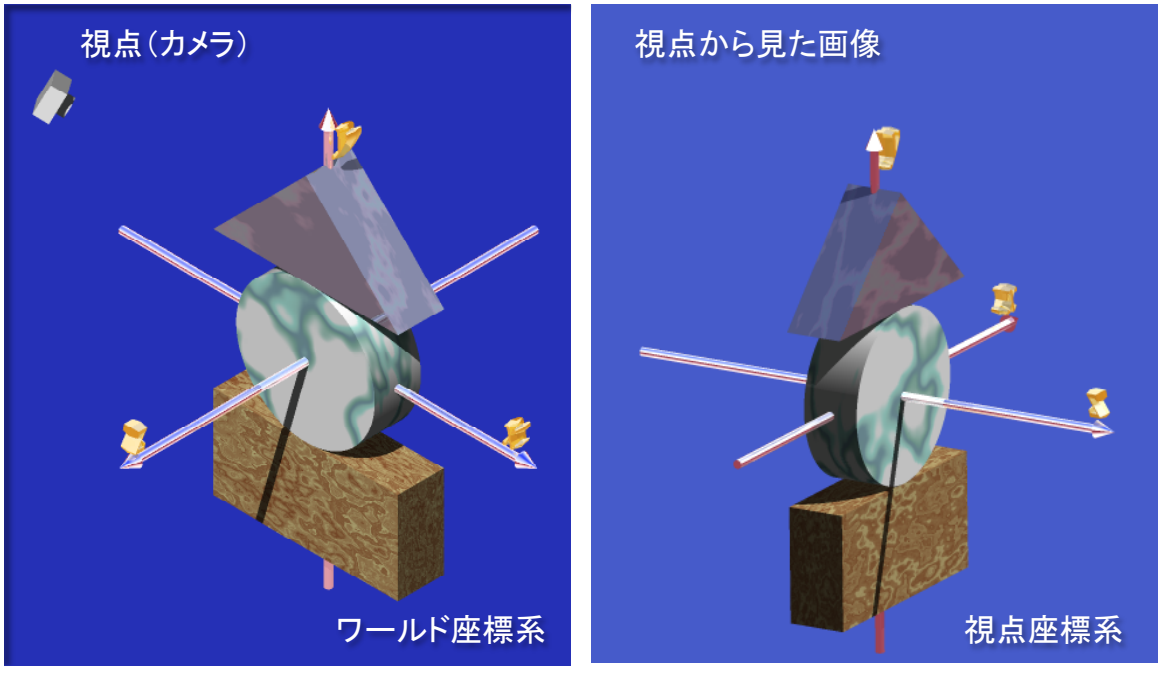

図 15 ビュー変換

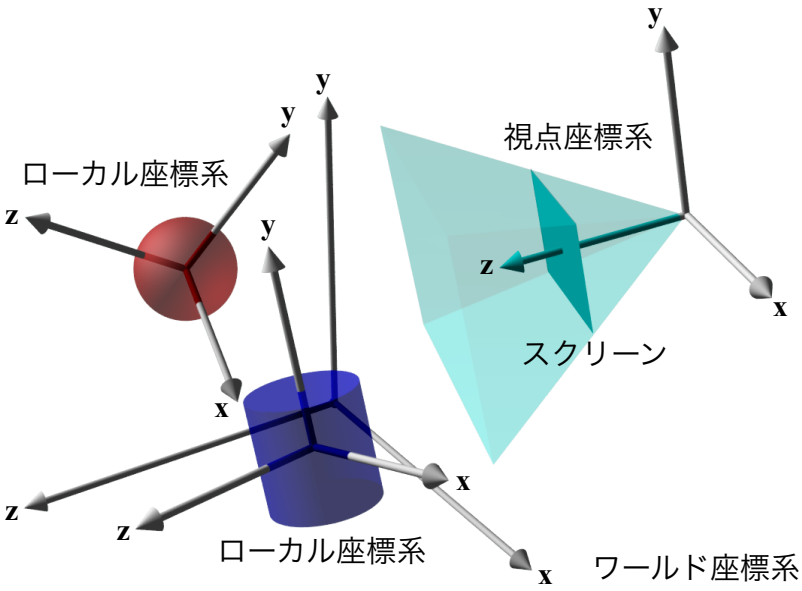

図 16 ローカル座標系、ワールド座標系、視点座標系の関係

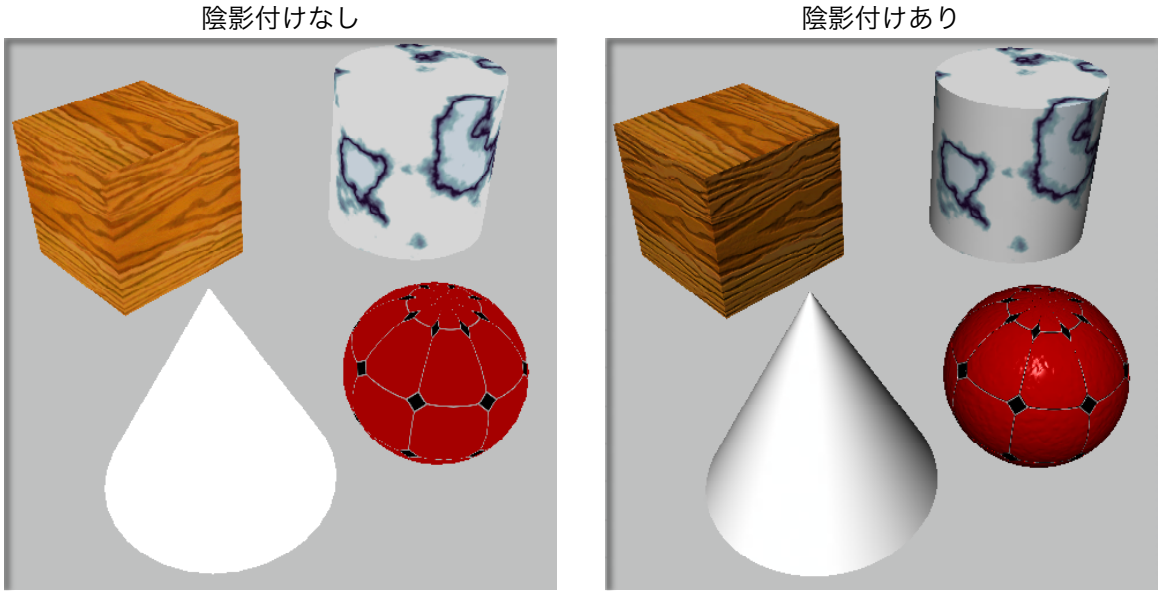

図 17 陰影付けの有無

● 陰影付け

物体の表面に光を当てた時の反射光の強度を計算し、物体にリアルな外観を与えます (図 17)。ジオメトリ処理のステージでは、この処理は頂点単位に行われます。その結果の頂点データがラスタライズの際に補間され、**画素** (**ピクセル**、Pixel) の陰影を求めるために用いられます。

実世界の光の反射は、光と物体表面との相互作用により発生しますが、リアルタイムレンダリングでは、この計算に多くの時間を割くことができません。このため、従来この計算には単純な関係式が用いられ、本物の反射 (映りこみ) や屈折、影 (シャドウ) などの再現は行われませんでした。しかし、近年は GPU によりかなり精密に計算することが可能になりました。

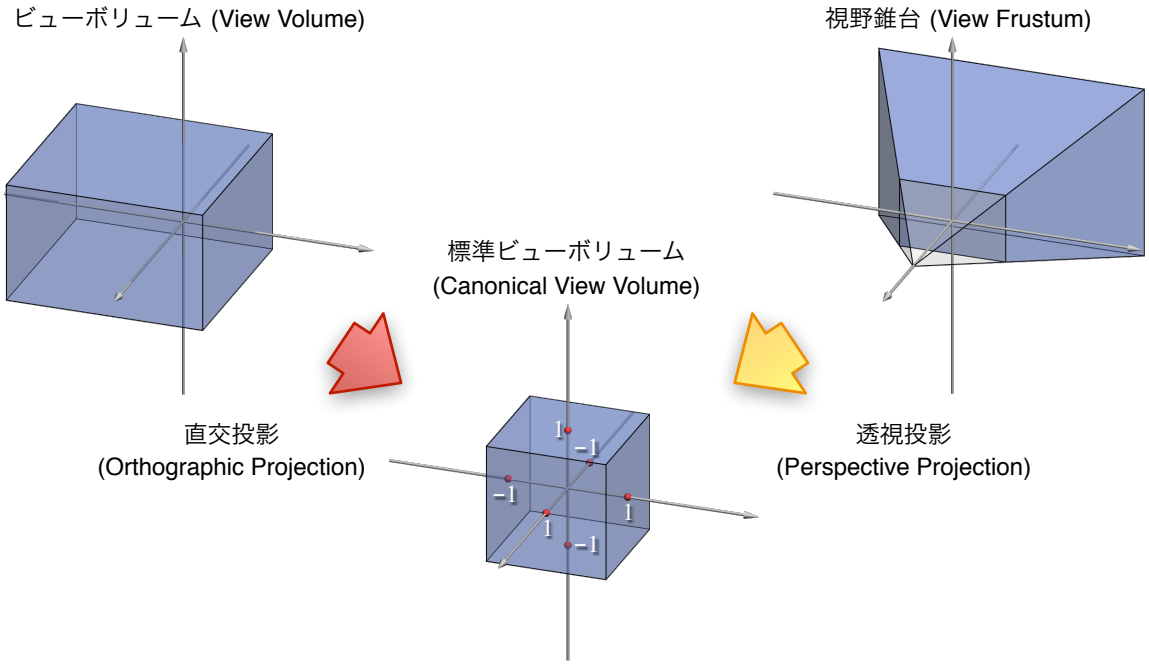

図 18 ビューボリューム

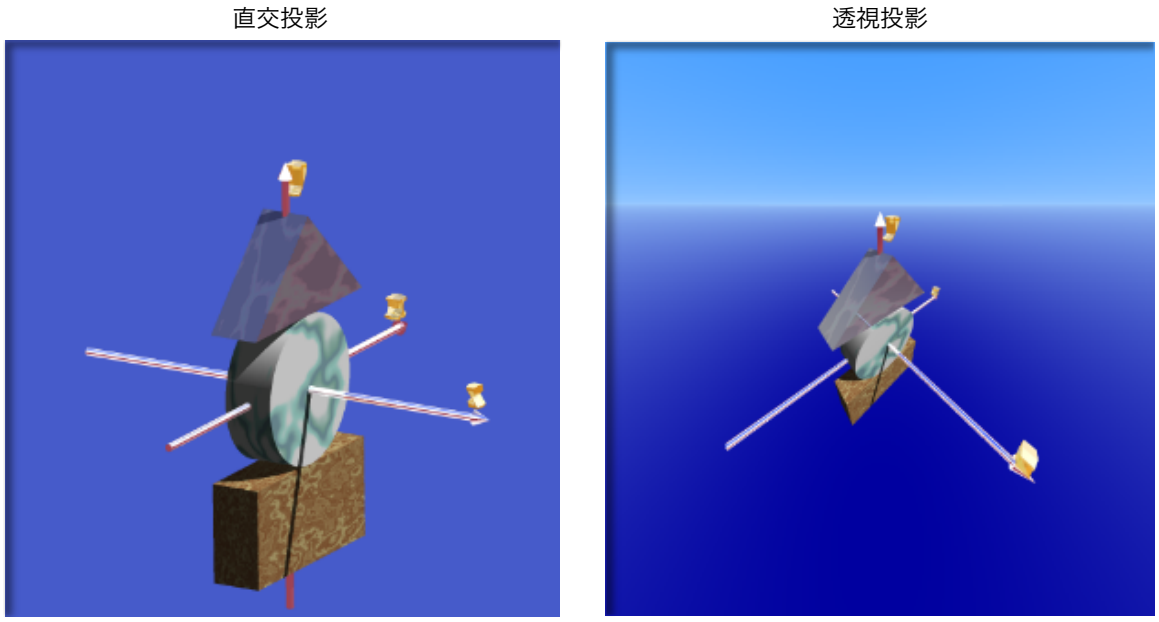

図 19 直交投影と透視投影

● **投影変換**

視点座標系に配置されたシーンをスクリーンに投影する変換を**投影変換**といいます。ディスプレイの表示領域は有限ですから、そこに映るシーンの空間も限定されます。この空間を**ビューボリューム** (View Volume) といいます。投影変換は、この空間を、中心が原点にあり一辺の長さが 2 の立方体の空間である**標準ビューボリューム** (Canonical View Volume) に変形します (図 18)。

このとき、変換前のビューボリュームに直方体を用いる場合と、四角錐台を用いる場合の二通りがあります。直方体を用いれば**直交投影** (Orthogonal Projection) となり、四角錐台を用いれば

透視投影 (Perspective Projection) となります (図 19)。なお、この四角錐台のビューボリュームは特に**ビューフラスタム** (View Frustum, **視錐台**) と呼ばれます。

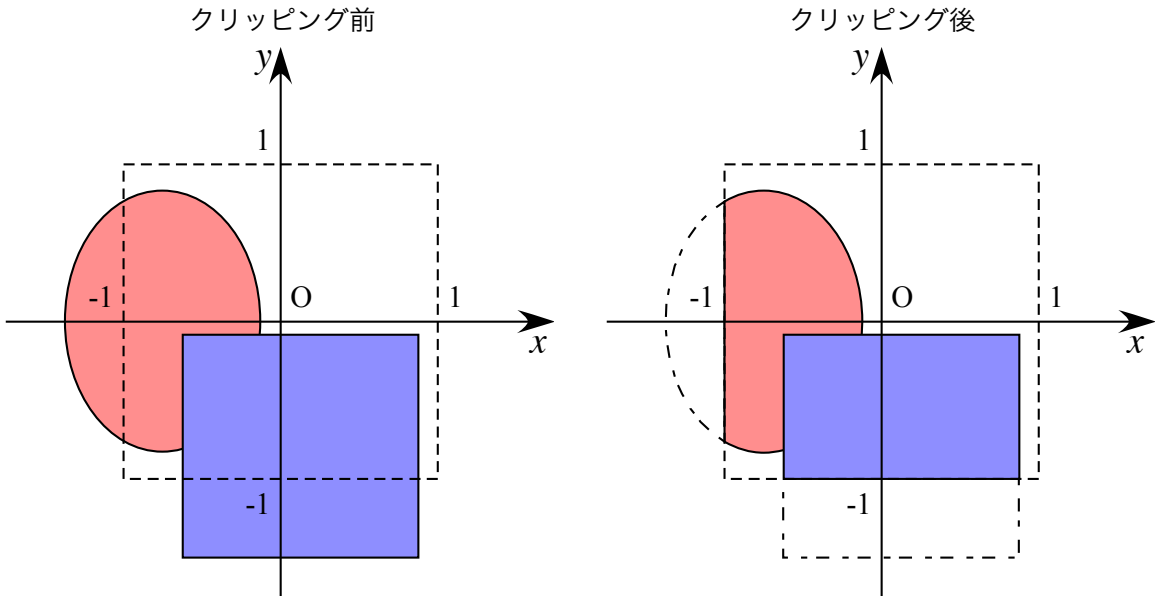

図 20 二次元のクリッピング

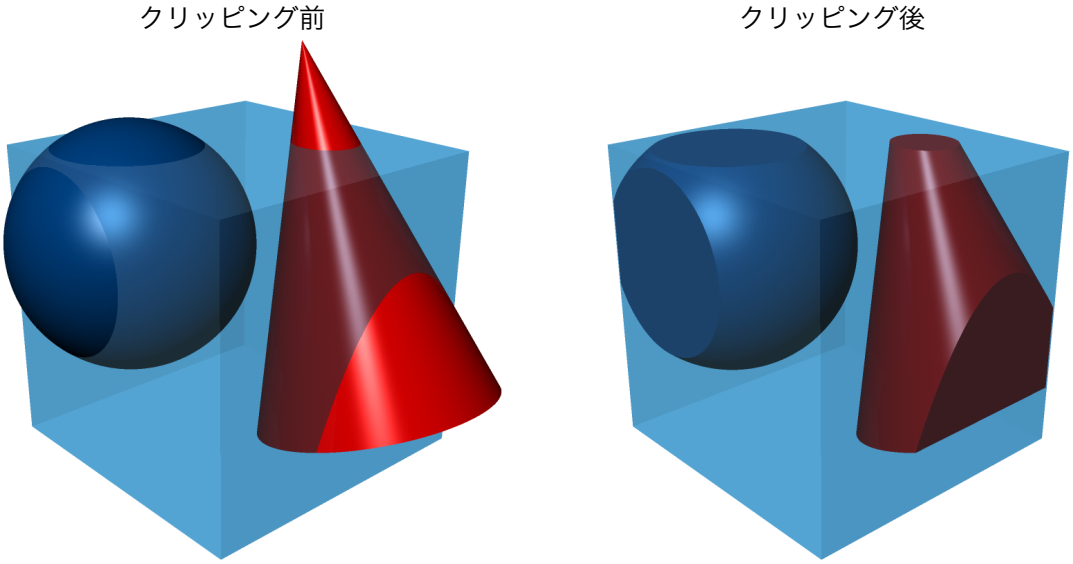

図 21 ビューボリュームによるクリッピングのイメージ

● **クリッピング**

グラフィックスハードウェアは標準ビューボリュームからはみ出た図形を切り取り、この内部にある図形だけを画面に表示します。この処理を**クリッピング** (Clipping) といいます。このため、標準ビューボリュームは**クリッピング空間**とも呼ばれ、クリッピング空間の座標系は**クリッピング座標系**と呼ばれます。また、この座標系は次のビューポート変換によりデバイス座標系に対応づけられるので、**正規化デバイス座標系** (Normalized Dvice Coordnate, NDC) と呼ばれます。

● **ビューポート変換**

クリッピングの結果、ビューボリュームに収められたシーンの、ビューボリュームの xy 平面への直交投影像を、ディスプレイ上の表示領域にはめ込みます。このディスプレイ上の表示領域を**ビューポート** (Viewport) といい、このはめ込みを行う変換を**ビューポート変換** (Viewport Transformation) あるいは**スクリーンマッピング** (Screen Mapping) と呼びます。

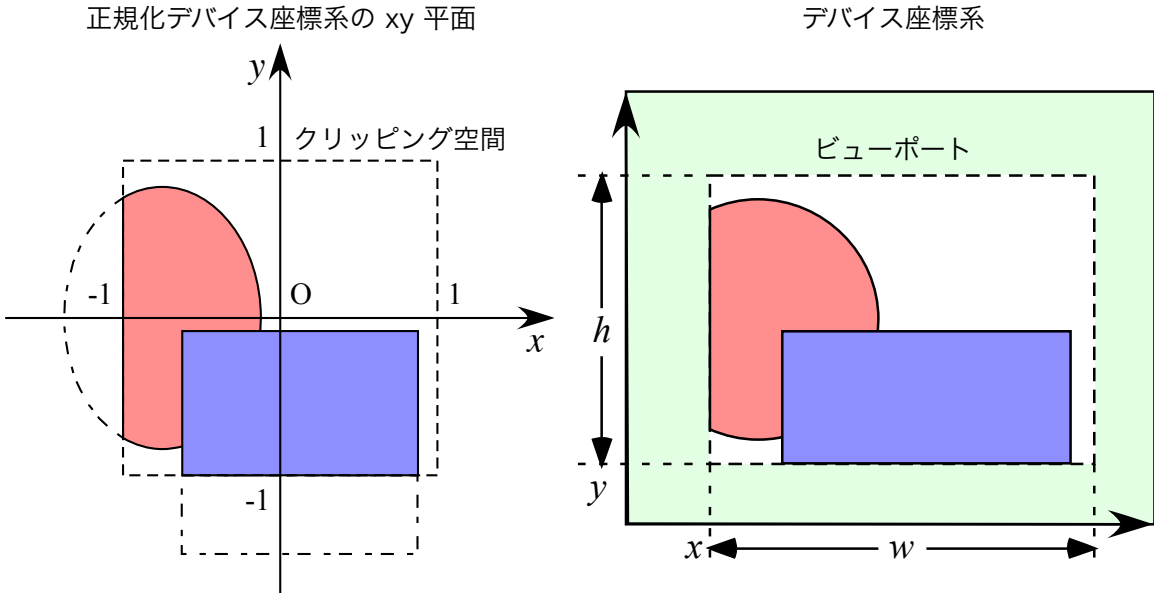

図 22 ビューポート変換

図 23 フラグメント処理のパイプライン

2.4.4 フラグメント処理ステージ

画像を構成する画素に関するデータ、いわゆる**フラグメントデータ**を処理するステージです。まず、ジオメトリ処理の結果を**ラスタライズ**によりデジタル画像化して、処理するフラグメントを選択します。そして、その各フラグメントの色を決定し、フレームバッファに出力して画像を生成します (図 23)。実際のハードウェアには、これらの他にもステージがあります。

● **ラスタライズ**

ラスタライズは**ラスタライザ**によって行われます。これは**三角形セットアップ** (Triangle Setup) と**走査変換** (スキャンコンバージョン, Scan Conversion) の二つのステージで構成されます。走査変換は塗り潰し処理によって三角形内に含まれるフラグメントを選択します。三角形セットアップは三角形のディスプレイ上の頂点位置から走査変換に必要なパラメータの算出を行います。

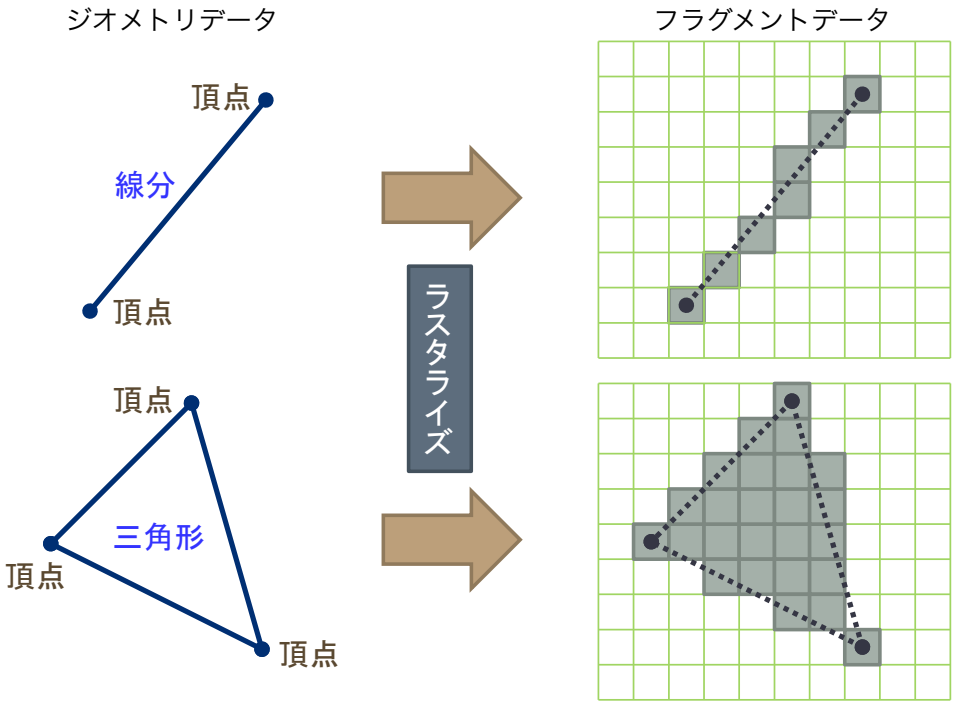

図 24 ラスタライズ

● **フラグメントの陰影付け**

フラグメントの陰影付けはフラグメント処理の中心になるステージで、ラスタライザから受け取った処理対象のフラグメントに関するパラメータ (頂点位置、頂点色、頂点の法線ベクトル、テクスチャ座標などの頂点属性のそのフラグメントにおける補間値) を用いて、フラグメントの色を決定します。テクスチャマッピングもここで行います。

● **フラグメント単位の処理**

フラグメントの陰影付けを行った後、その結果を最終的な画像を保持するフレームバッファに保存するまでの間にも、様々な処理が実行されます。これも以下のようなステージを持つパイプラインとして構成されています。

【フォグ処理】

生成された画像に対して、大気効果 (霧やガスの効果) を与えます。遠景を描かずにごまかすときなどによく用いられます。ただし、現在これはフラグメントの陰影付けの際に行われるため、ハードウェアとして特別なステージは用意されていません。

フォグなし　　フォグあり

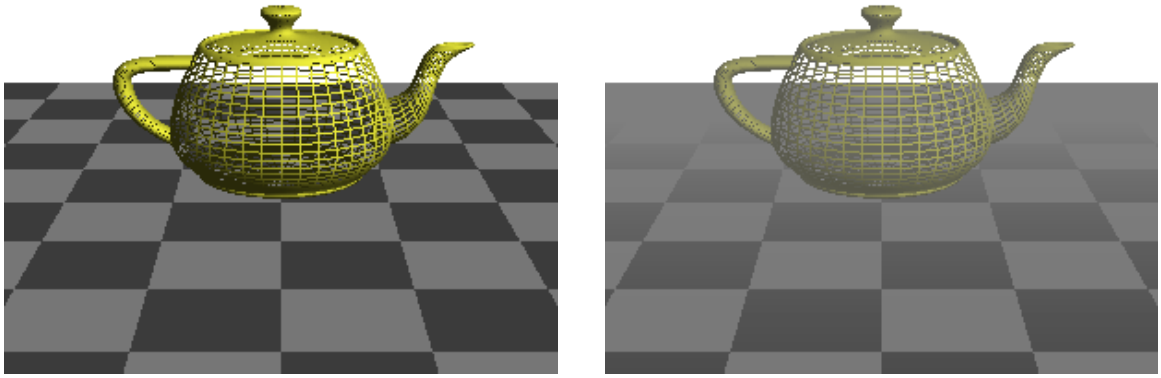

図 25 フォグ処理

【ステンシルテスト】

フレームバッファを構成するバッファの一つであるステンシルバッファに格納されている値を参照して、そのフラグメントの表示の可否を決定します。ステンシルバッファの内容をマスクに用いて表示する形状の型抜きを行うことができます。

【デプステスト】

フレームバッファを構成するバッファの一つであるデプスバッファに格納されている値を参照して、そのフラグメントの表示の可否を決定します。これを用いてデプスバッファ法による隠面消去処理を行うことができます。

【ブレンド処理】

フレームバファの内容と表示しようとするフラグメントの値を合成します。半透明処理や画像の合成などを行うことができます。

【ロジック処理】

フレームバファの内容と表示しようとするフラグメントの値をもとに論理演算を行います。

● **フレームバッファ**

フレームバッファはレンダリングの最終結果を合成し、映像の表示を行うハードウェアです。これはカラーバッファ、デプスバッファ、ステンシルバッファの複数のバッファの集合体です。**フレームバッファオブジェクト** (Frame Buffer Object, FBO) という機能を用いれば、 任意のバッファを組み合わせたフレームバッファをユーザが構成することができます。

【カラーバッファ】

フラグメントの色を保持するバッファで、ここに画像を格納します。ダブルバッファリングを行う場合や立体視表示を行う場合は、複数のカラーバッファが用意されます。

【デプスバッファ】

デプスバッファ法による隠面消去処理に用いるバッファです。これにはフラグメントのデプス値 (深度値) が格納されます。

【ステンシルバッファ】

表示形状の「型抜き」を行うために、マスクとなる画像を保持するバッファです。

【アキュムレーションバッファ】

カラーバッファの内容を累積することができるバファです。これは古い機能であり、近年は使われていません。

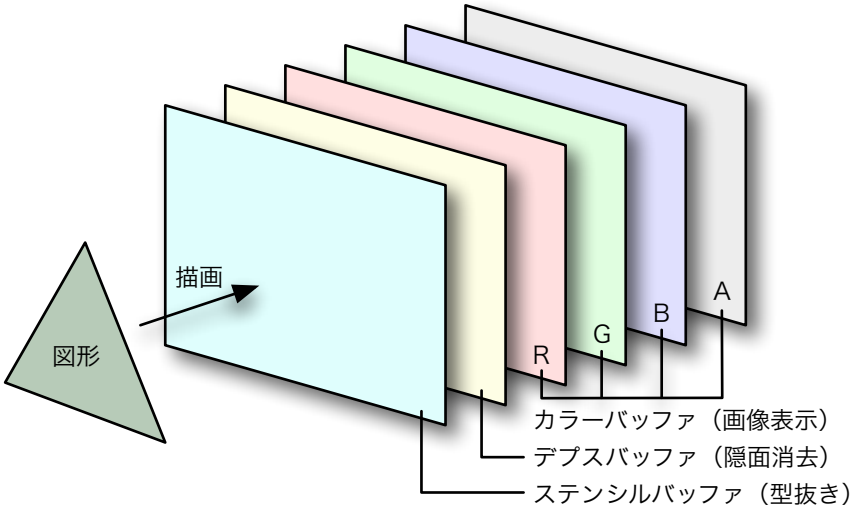

図 26 フレームバッファの構成

第3章 プログラムの作成

3.1 GLFW の導入

OpenGL はプラットホームに組み込まれていますが、1.5.1 節で述べた通り、この呼び出し方法はプラットホームごとに異なります。そこで、この講義では各プラットホームでソースコードを共通にするために、ツールキットの GLFW (http://www.glfw.org) を利用します。GLFW は別途導入する必要があります。

3.1.1 Windows への導入

GLFW のプロジェクトのダウンロードページ (http://www.glfw.org/download.html) の “32-bit Windows binaries” のボタンをクリックしてください。すると 32bit 版のバイナリファイルをまとめた ZIP ファイル glfw-3.X.Y.bin.WIN32.zip (X, Y は数字) がダウンロードされますから、これを右クリックで選択して「すべて展開」を選び、書き込み可能な適当なディレクトリ (マイドキュメント、デスクトップ、C:¥ など) に展開してください。

パソコンの管理者権限を持っている場合は、この中のフォルダやファイルを以下の「システムのディレクトリ」(Visual Studio 2013 の場合) に移動あるいはコピーしてください。これにより、プロジェクトの作成時に「VC++ ディレクトリ」を設定する手間を省くことができます。

- 32bit 版 Windows の場合

  include フォルダにある GLFW フォルダ

  ```
  C:¥Program Files¥Windows Kits¥8.1¥Include¥um
  ```

  lib-msvc120 フォルダにある glfw3.lib ファイル

  ```
  C:¥Program Files¥Windows Kits¥8.1¥Lib¥winv6.3¥um¥x86
  ```

● 64bit 版 Windows の場合

include フォルダにある GLFW フォルダ

```
C:¥Program Files (x86)¥Windows Kits¥8.1¥Include¥um
```

lib-msvc120 フォルダにある glfw3.lib ファイル

```
C:¥Program Files (x86)¥Windows Kits¥8.1¥Lib¥winv6.3¥um¥x86
```

パソコンの管理者権限を持っていない場合は、Visual Studio のプロジェクト作成時に、「VC++ ディレクトリ」に ZIP ファイルを展開したディレクトリ (フォルダ) を指定します。この方法は後述します。

なお、ソースファイルからコンパイルする場合は CMake (http://www.cmake.org) を利用します。CMake (cmake-gui) を起動して “Where is the source code” にソースファイルの ZIP ファイルを展開したディレクトリを指定し、“Where to build the binaries” には書き込み可能なディレクトリを指定します。その後、“Configure” をクリックして使用する Visual Studio のバージョンを指定した後 “Generate” をクリックすれば、Visual Studio のプロジェクトファイルが作成されます。

3.1.2 Mac OS X

Mac OS X では、Xcode と Command Line Tools がインストールされている必要があります。Xcode は AppStore から無料で入手できます。Command Line Tools は、Xcode のバージョン 5 には同梱されていますが、“Xcode” メニューから “Open Developer Tool” の “More Developer Tools” を選ぶとダウンロードページが表示されます (無料の開発者ユーザ登録が必要です)。

本稿の執筆時点では、MacPorts (http://www.macports.org) に GLFW のバージョン 3 のパッケージが用意されていましたが、HomeBrew (http://brew.sh) や Fink (http://www.finkproject.org) にはありませんでした。ソースファイルからは以下の手順でインストールできます。

● インストール

GLFW のプロジェクトのダウンロードページ (http://www.glfw.org/download.html) からソースファイルの ZIP ファイル glfw-3.X.Y.zip (X, Y は数字) をダウンロードしてデスクトップに置き、それをダブルクリックして展開してください。glfw-3.X.Y というディレクトリが作成されます。

次にターミナルを開き、以下のコマンドを順に実行してください (管理者権限が必要です)。‘%’ はシェルのプロンプトを表します。なお、ファイルは /usr/local 以下にインストールされます。

```
% cd ~/Desktop/glfw-3.X.Y
% mkdir build
% cd build
% cmake ..
% make
% sudo make install
```

● アンインストール

インストールした GLFW のファイルを削除する必要が生じた場合は、前述の手順で GLFW をインストールしたときに build ディレクトリの中に作成される Makefile を使ってアンインストールできます (管理者権限が必要です)。

```
% cd ~/Desktop/glfw-3.X.Y/build
% sudo make uninstall
```

3.1.3 Linux

Linux (X11) は NVIDIA GeForce 8 シリーズ以降あるいは ATI RADEON HD シリーズ以降のビデオカードで、プロプライエタリドライバがインストールされていることを想定しています。Intel の CPU 内蔵グラフィックスの場合、ドライバは https://01.org から入手できます。

本稿の執筆時点では、Fedora (http://fedoraproject.org) に GLFW のバージョン 3 のパッケージが用意されていましたが、Ubuntu (http://www.ubuntu.com) や OpenSUSE (http://www.opensuse.org) にはありませんでした。ソースファイルからは以下の手順でインストールできます。

- **インストール**

GLFW のプロジェクトのダウンロードページ (http://www.glfw.org/download.html) からソースファイルの ZIP ファイル glfw-3.X.Y.zip (X, Y は数字) ダウンロードしてください。

次にターミナルを開き、以下のコマンドを順に実行してください (管理者権限が必要です)。'%' はシェルのプロンプトを表します。なお、ファイルは /usr/local 以下にインストールされます。

```
% unzip glfw-3.X.Y.zip
% cd glfw-3.X.Y
% mkdir build
% cd build
% cmake ..
% make
% sudo make install
```

- **アンインストール**

インストールした GLFW のファイルを削除する必要が生じた場合は、前述の手順で GLFW をインストールしたときに build ディレクトリの中に作成される Makefile を使ってアンインストールできます (管理者権限が必要です)。

```
% cd glfw-3.X.Y/build
% sudo make uninstall
```

3.2 ソフトウェア開発環境

プログラムの作成はテキストエディタとコンパイラ・リンカなどの言語処理系さえあれば可能ですが、現在はテキストエディタやコンパイラ・リンカ、デバッガなどをひとまとめにした、統合開発環境 (Integrated Development Environment, IDE) が一般的に用いられます。ここでは各プラットホームにおいてよく使われるソフトウェア開発環境について説明します。

3.2.1 Windows

- **プロジェクトの新規作成**

Windows では Visual Studio 2013 を例にして説明します。Visual Studio 2013 を起動し、新しいプロジェクトを作成してください。新しいプロジェクトを作成するには、「ファイル」のメニューの「新規作成」から「プロジェクト」を選ぶか、図 27 の矢印のところをクリックします。

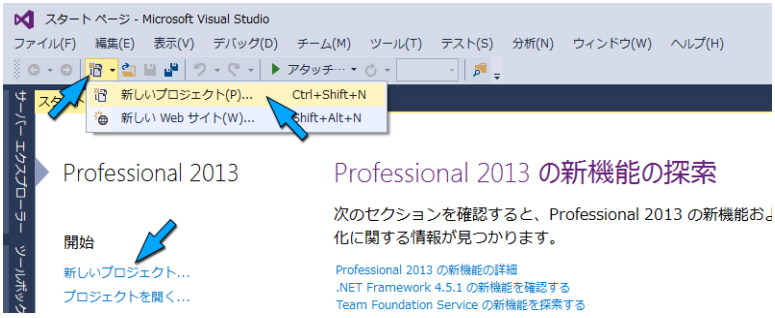

図 27 プロジェクトの作成

インストール済みのテンプレートから「Visual C++」の「Win32 コンソールアプリケーション」を選び、作成するプログラムの「名前」を設定してから「OK」をクリックしてください (図 28)。

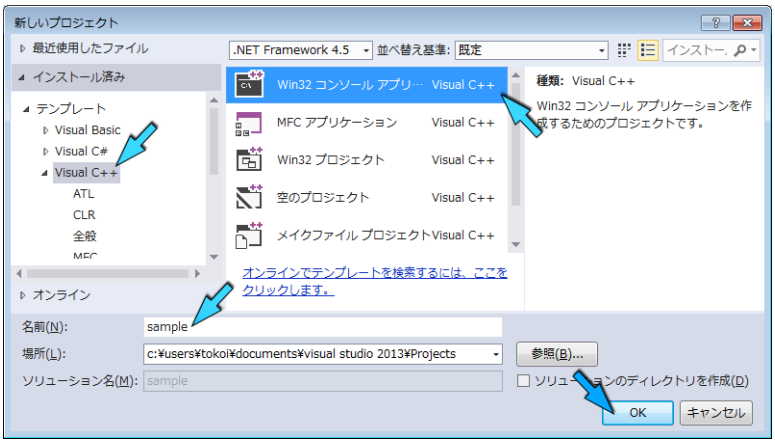

図 28 「Win32 コンソールアプリケーション」の選択

作成するプロジェクトの設定を変更するので、図 29 では「次へ」をクリックします。

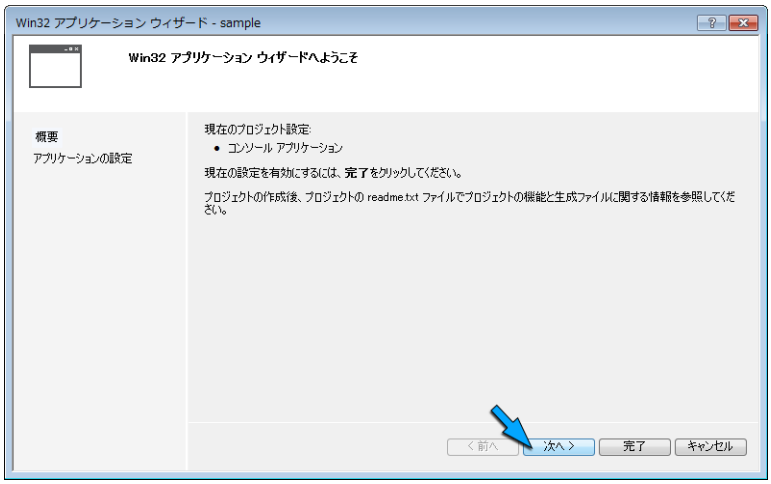

図 29 「次へ」をクリック

「追加のオプション」のウィンドウ (図 30) で「空のプロジェクト」を選択します。これを選択しなければ main() 関数を含むソースファイルが作成されますが、それを使用しても構いません。最後に「完了」をクリックしてください。

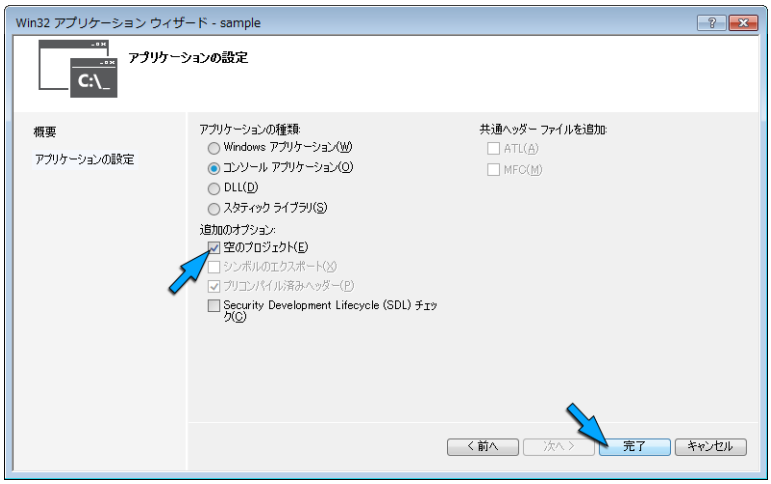

図 30 「空のプロジェクト」の選択

● プロジェクトのプロパティ

このプロジェクトで、GLFW を使用する設定を行います。「プロジェクト」のメニューの「プロパティ」を選択してください (図 31)。

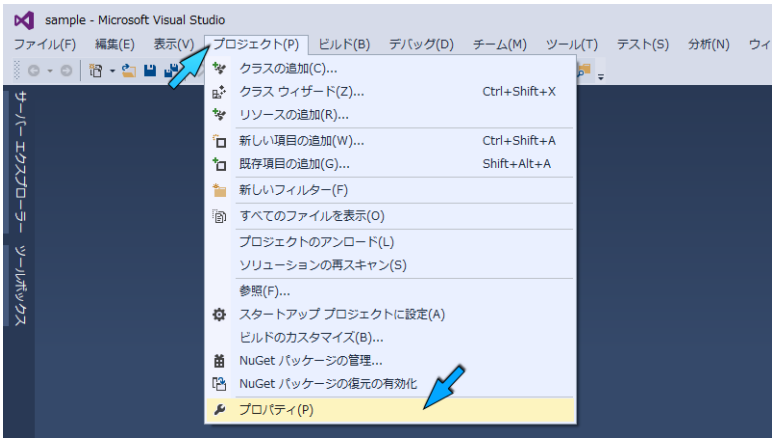

図 31 プロジェクトの「プロパティ」の設定

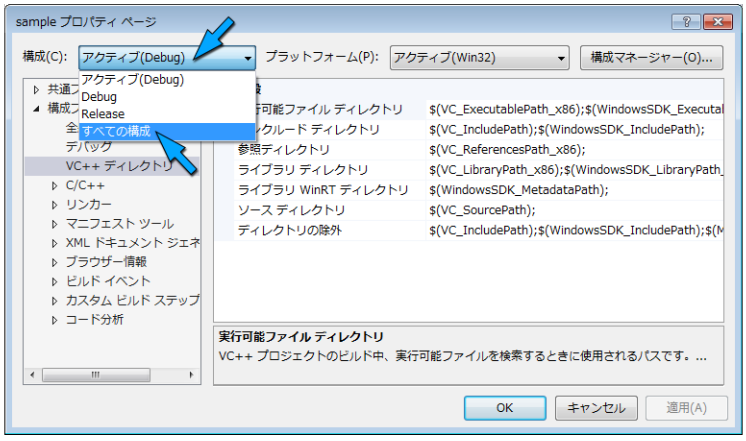

図 32 プロジェクトの「構成」の選択

図 32 のプロジェクトの構成には、デバッグのときに用いる「デバッグ」と、デバッグが完了

して最終的なプログラムを作成するときに用いる「リリース」が用意されています。「リリース」構成でプログラムをビルド (コンパイルやリンク等の一連の処理を経て目的のプログラムを作成する作業) すると最適化により効率の良いプログラムが生成されますが、不要なコードが削除されたりするためにソースプログラムとコンパイルした結果が一致しなくなり、デバッグが難しくなります。ここでは両方の構成に同じ設定を適用するために、「全ての構成」を選びます。

● **「VC++ ディレクトリ」の設定 (ファイルを「システムのディレクトリ」に置いた時は不要)**

GLFW のヘッダファイルやライブラリファイルの場所を設定します。プロパティページのウィンドウ (図 33) の左側のペイン (ウィンドウの領域) で「VC++ ディレクトリ」を選び、「インクルードディレクトリ」の欄の右端の▼をクリックして、<編集...> を選んでください。

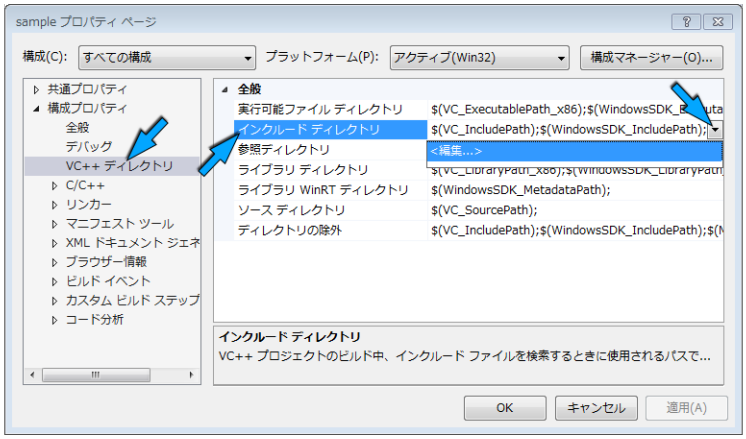

図 33 「インクルードディレクトリ」の項目の編集

「インクルードディレクトリ」の設定ウィンドウ (図 34) の右上のフォルダのアイコンをクリックすると、欄がひとつ作成されます。その欄の右側の「...」をクリックしてください。

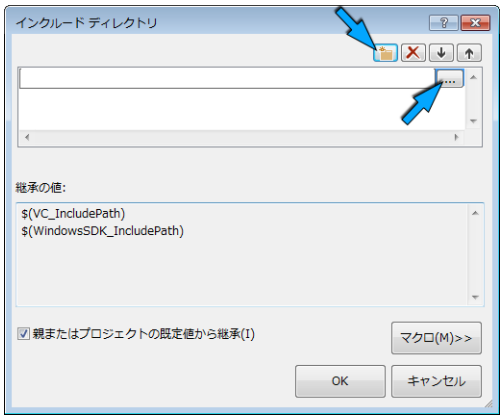

図 34 「インクルードディレクトリ」の設定ウィンドウ

ディレクトリの設定のダイアログウィンドウ (**エラー! 参照元が見つかりません。**) が現れるので、GLFW のバイナリファイルの ZIP ファイルを展開したディレクトリの中の include ディレクトリ (図 35) を選択して、「フォルダーの選択」をクリックしてください。

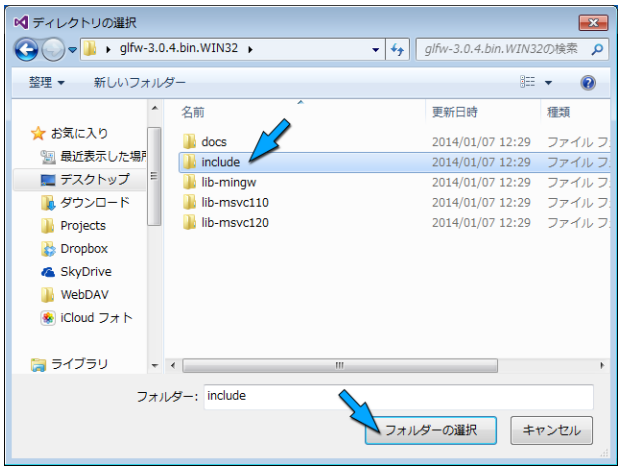

図 35 GLFW のパッケージの include ディレクトリの選択

以上の設定が完了したら、「OK」をクリックしてください (図 36)。

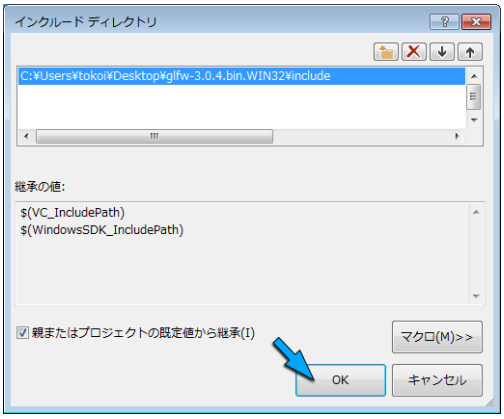

図 36 「インクルードディレクトリ」の設定完了

「ライブラリディレクトリ」の欄の右端の▼をクリックして、<編集...> を選びます (図 37)。

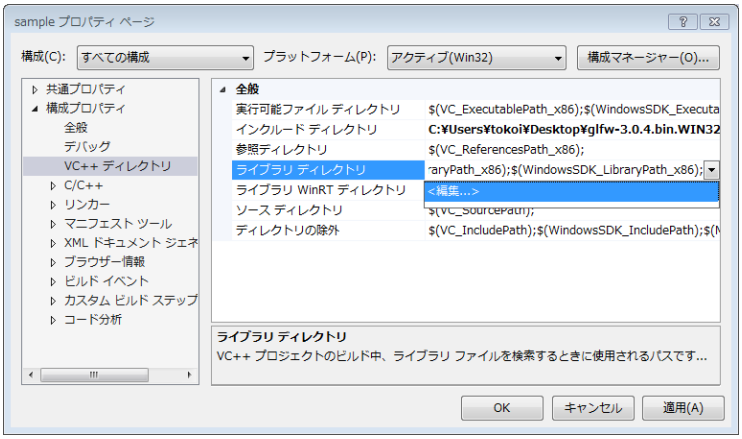

図 37 「ライブラリディレクトリ」の項目の編集

「ライブラリディレクトリ」の設定ウィンドウ (図 38) の右上のフォルダのアイコンをクリックして、欄をひとつ作成します。その後、その欄の右側の「...」をクリックしてください。

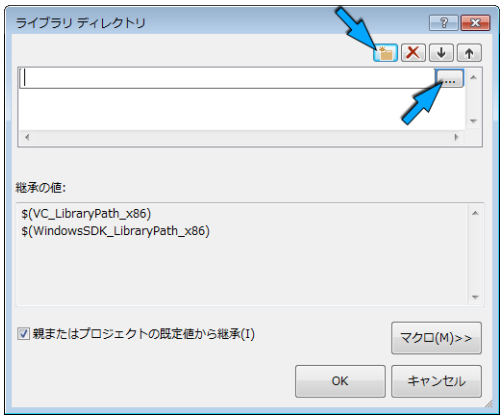

図 38 「ライブラリディレクトリ」の設定ウィンドウ

GLFW のバイナリファイルの ZIP ファイルを展開したディレクトリの中の lib-msvc120 ディレクトリを選択して、「フォルダーの選択」をクリックしてください (図 39)。

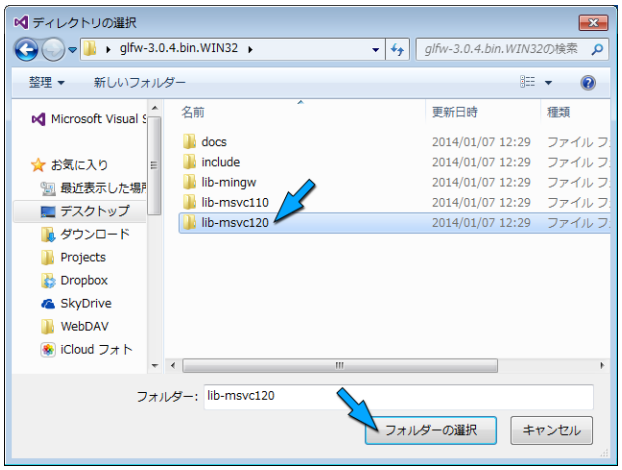

図 39 GLFW パッケージの lib-msvc120 ディレクトリの選択

以上の設定が完了したら、「OK」をクリックしてください (図 40)。

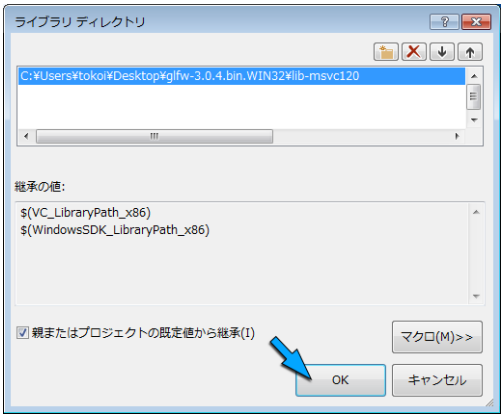

図 40 「ライブラリディレクトリ」の設定完了

- **リンクするライブラリファイルの指定**

プログラムのリンク時にリンクする GLFW と OpenGL のライブラリファイルを指定します。

プロパティページのウィンドウ (図 41) の左側のペインで「リンカー」「入力」の順に選択し、「追加の依存ファイル」の欄の右端の▼をクリックして、<編集...> を選んでください。

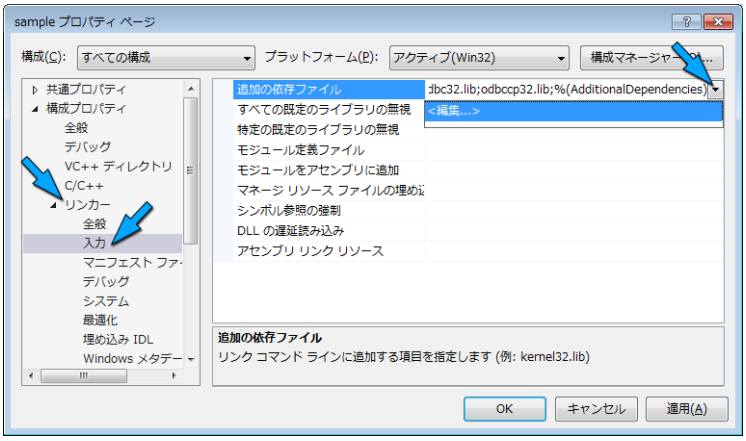

図 41 追加の依存ファイル

リンクするライブラリファイルとして、glfw3.lib, opengl32.lib の二つを「追加の依存ファイル」に追加します (図 42)。その後、「OK」をクリックしてください。

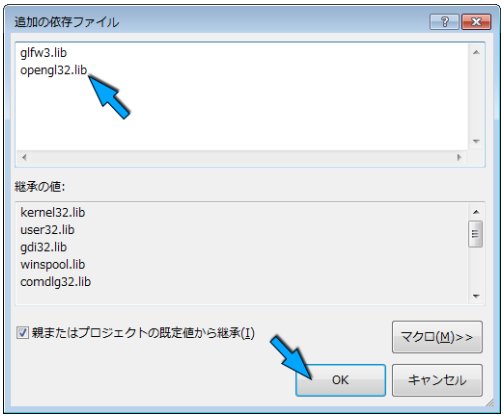

図 42 glfw3.lib と opengl32.lib の追加

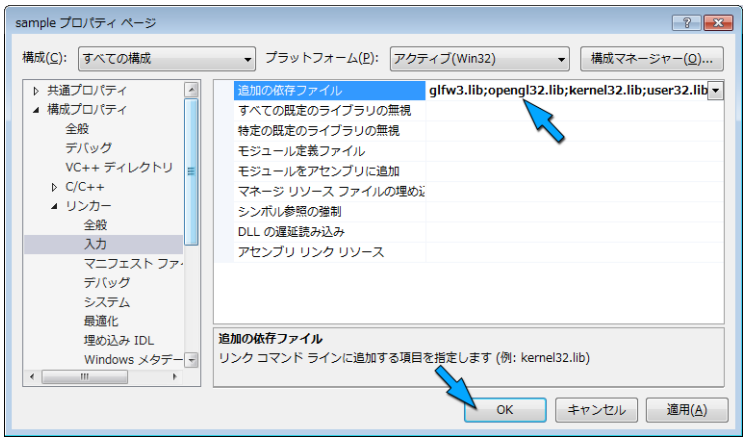

図 43 追加後の「追加の依存ファイル」

追加後の「追加の依存ファイル」は図 43 のようになります。ここに直接 “glfw3.lib; opengl32.lib;”

を入力することもできます。最後に「OK」をクリックしてください。

補足：ソースプログラムへの設定の埋め込み

「リンクするライブラリファイルの指定」はソースプログラム (複数ある場合は、どれかひとつだけで構いません) に次の 2 行を置くことにより、省くことができます。

```
#pragma comment(lib, "glfw3.lib")
#pragma comment(lib, "opengl32.lib")
```

したがって、GLFW 関連のファイルを「システムのディレクトリ」に置き、ソースプログラムの冒頭でこれらの設定を行えば、プロジェクトを新規作成するたびに「プロジェクトのプロパティ」を設定する手間を省くことができます。

また、「Win32 コンソールアプリケーション」のプロジェクトで作成したプログラムは、実行すると OpenGL のウィンドウのほかに「コンソールウィンドウ」が開きます。コンソールウィンドウはデバッグ時にメッセージ等を表示するのに有用ですが、これを開きたくない場合は次の内容をソースプログラムに追加してください (続けて 1 行で入力してください)。

```
#pragma comment(linker, "/subsystem:¥"windows¥" /entry:¥"mainCRTStartup¥"")
```

補足：ライブラリのリンクについて

ライブラリのリンク方法には、スタティックリンクとダイナミックリンクという二つの方法があります。スタティックリンクは、ライブラリファイルに登録されている関数をプログラムのリンク時にプログラム自体に組み込む方法です。一方ダイナミックリンクは、ライブラリファイルを別に用意しておき、プログラムの実行時にそのライブラリファイルに登録されている関数を呼び出す方法です。この別に用意したライブラリファイルのことを、ダイナミックリンクライブラリ (Dynamic Link Library, DLL) と呼びます。

ライブラリファイルに登録されている関数は複数のプログラムから利用されるため、スタティックリンクを行うと同じ関数のコピーが異なるプログラム内に存在することになり、メモリやディスクなどが無駄に使われます。これに対してダイナミックリンクでは、関数の実体はひとつになるため、スタティックリンクのような無駄がありません。また、ライブラリが更新されたときは DLL を入れ替えるだけで済み、スタティックリンクのようにリンクし直す必要がありません。

このようにメリットの多いダイナミックリンクですが、本書ではスタティックリンク用のライブラリである glfw3.lib をリンクしています。この理由は、ダイナミックリンクでは DLL を「コマンドの検索パス」に含まれるディレクトリや「作業ディレクトリ」に置くことを嫌ったためです。また、DLL のバージョンの不一致によるトラブルを避けることにも配慮しています。

しかし、Visual C++ においてこれらのスタティックリンク用のライブラリをリンクすると、Visual C++ によって暗黙的にリンクされる他のライブラリと競合しているという警告が表示されることがあります。この警告はダイナミックリンクを行えば抑制することができます。

GLFW をダイナミックリンクするには glfw3.lib の代わりに glfw3dll.lib をリンクします。これはプログラムから DLL (glfw3.dll) を呼び出すためのライブラリです。そして、この glfw3.dll を「コマンドの検索パス」に含まれるディレクトリか、作成したプログラムを実行する際の「作業ディレクトリ」に置いてください。前者の場合は glfw3.dll を「システムのディレクトリ」である C:¥Windows¥System32 (32bit 版 Windows) あるいは C:¥Windows¥SysWOW64 (64bit 版

Windows) に置くか、「システムの詳細設定[2]」で「システム環境変数」の Path に glfw3.dll を置いたディレクトリを追加してください。

● ソースファイルの追加

プログラムのソースファイルを作成します。「プロジェクト」のメニューから「新しい項目の追加」を選んでください (図 44)。

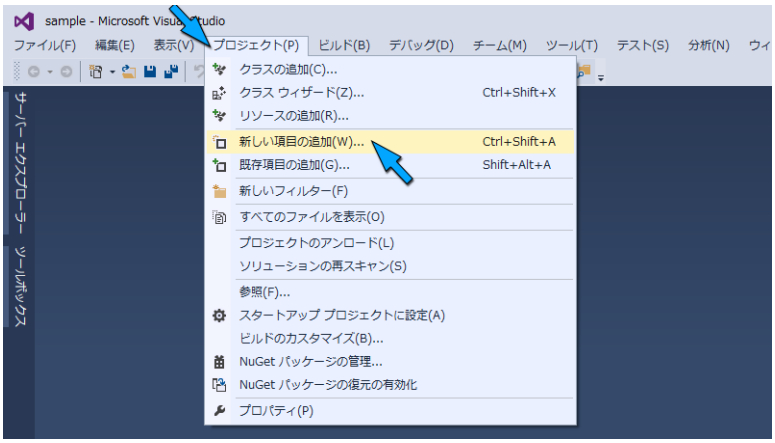

図 44 新しい項目の追加

「新しい項目の追加」のウィンドウ (図 45) の左側のペインで「Visual C++」を選び、中央のペインで「C++ ファイル (.cpp)」を選んでください。また、その下の「名前」の欄にソースファイルのファイル名を入力してください。その後、「追加」をクリックしてください。

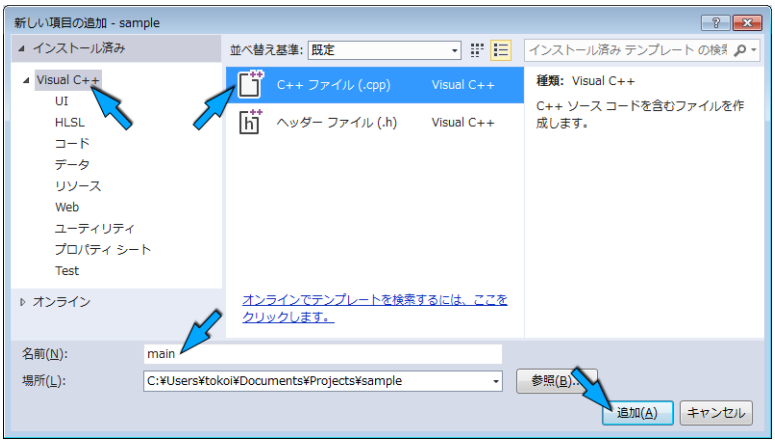

図 45 C++ のソースファイルの追加（新規作成）

テキストエディタのウィンドウ (図 46) が開きます。ここにソースプログラムを入力します。

[2] Windows 7:「スタートメニュー」から「コントロールパネル」「システムとセキュリティ」「システム」の順に選んで「システムの詳細設定」を選択します。Windows 8: デスクトップから「チャーム」を開き「設定」「PC 情報」の順に選んで「システムの詳細設定」を選択します。

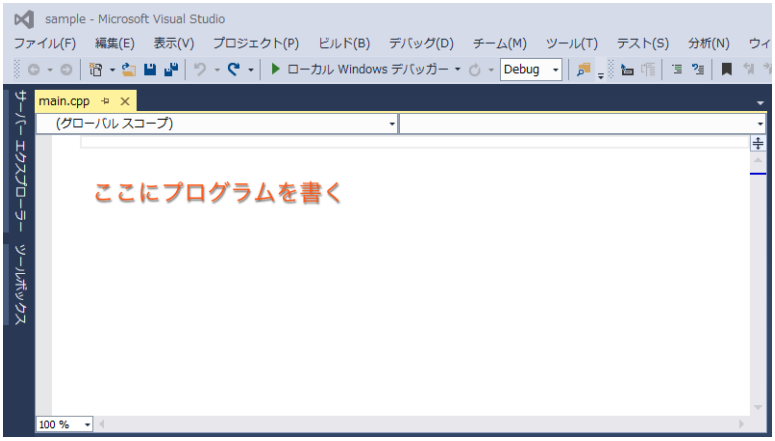

図 46 ソースプログラムの編集

3.2.2 Mac OS X

● **プロジェクトの新規作成**

Mac OS X では Xcode のバージョン 5 の例について説明します。Xcode を起動し、スプラッシュウィンドウ (図 47) の “Create a new Xcode project” をクリックするか、“File” メニューの “New” から “Project” を選んでください。

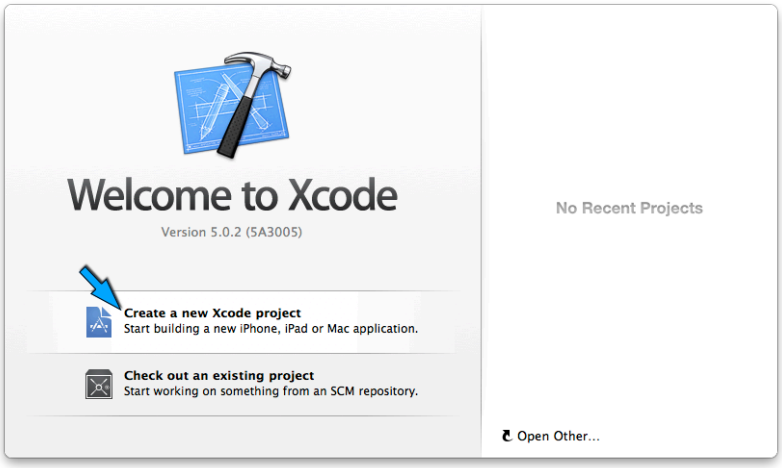

図 47 スプラッシュウィンドウ

プロジェクトのテンプレートとして “OS X” の “Application” から “Command Line Tool” を選び、“Next” をクリックしてください (図 48)。

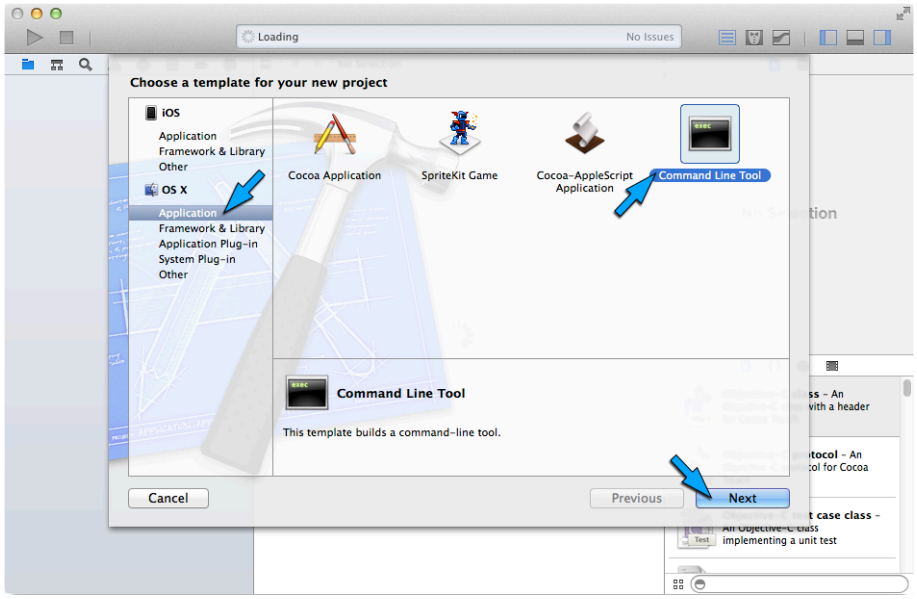

図 48 プロジェクトのテンプレートの選択

プロジェクトのオプションは、“Product Name” だけ設定してください。“Organization Name” や “Company Identifier” は既定値が設定されています。“Type” にはもちろん C++ を選んでください。その後 “Next” をクリックしてください (図 49)。

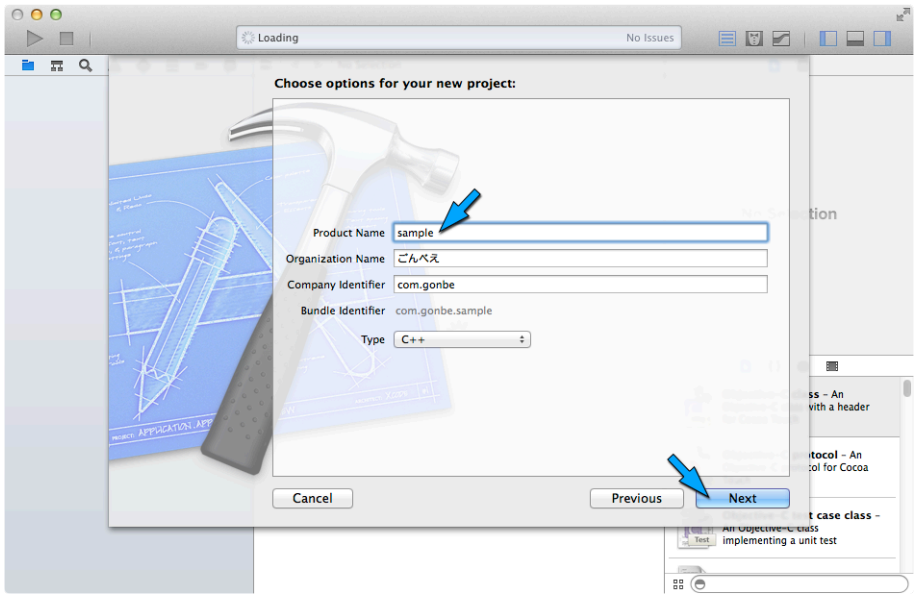

図 49 プロジェクトのオプションの設定

プロジェクトのディレクトリを作成する場所を指定します。適当なところを選んで、“Create” をクリックしてください (図 50)。

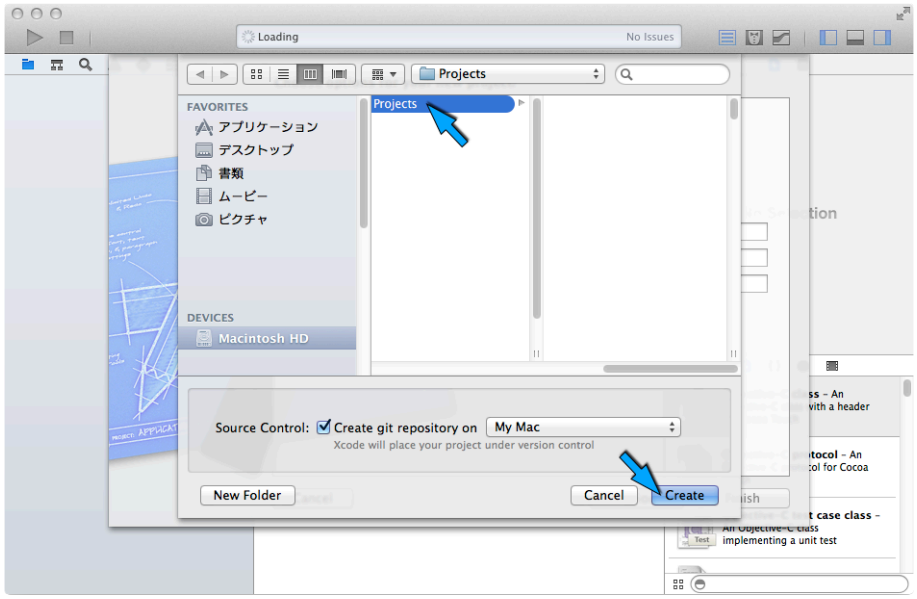

図 50 プロジェクトのディレクトリの保存先

● プロジェクトの設定

まず、ヘッダファイルとライブラリの検索パスを設定します。図 51 のウィンドウの左側にあるプロジェクト名をクリックし、その右のポップアップメニューから “Targets” を選んでください。次に中央部の “Build Settings” を選択します。ここには設定項目が大量にあるので、検索窓に “search” などを入力して “Header Search Paths” という項目を探し、その右側をダブルクリックします。すると入力ウィンドウがポップアップしますから、左下の “+” をクリックして欄を追加し、そこに GLFW のヘッダファイルをインストールした場所 (/usr/local/include) を設定してください。

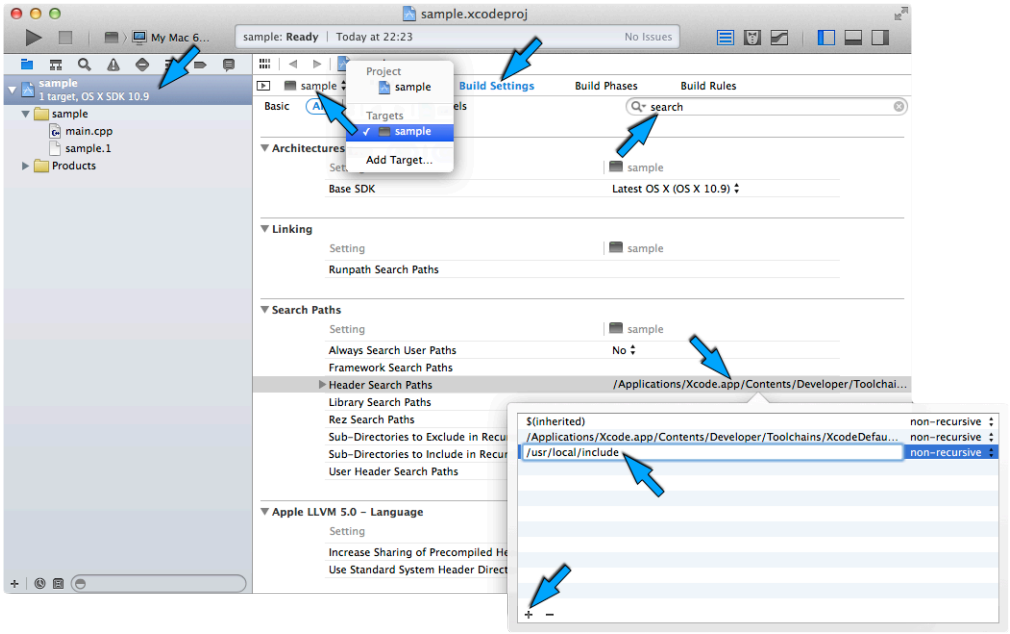

図 51 ヘッダファイルの検索パスの設定

ポップアップしたウィンドウは ESC キーをタイプするか、そのウィンドウ以外のところをク

リックすれば消えます。次に “Library Search Paths” の右側 (図 52) をダブルクリックし、同様に GLFW のライブラリファイルをインストールした場所 (/usr/local/lib) を設定します。

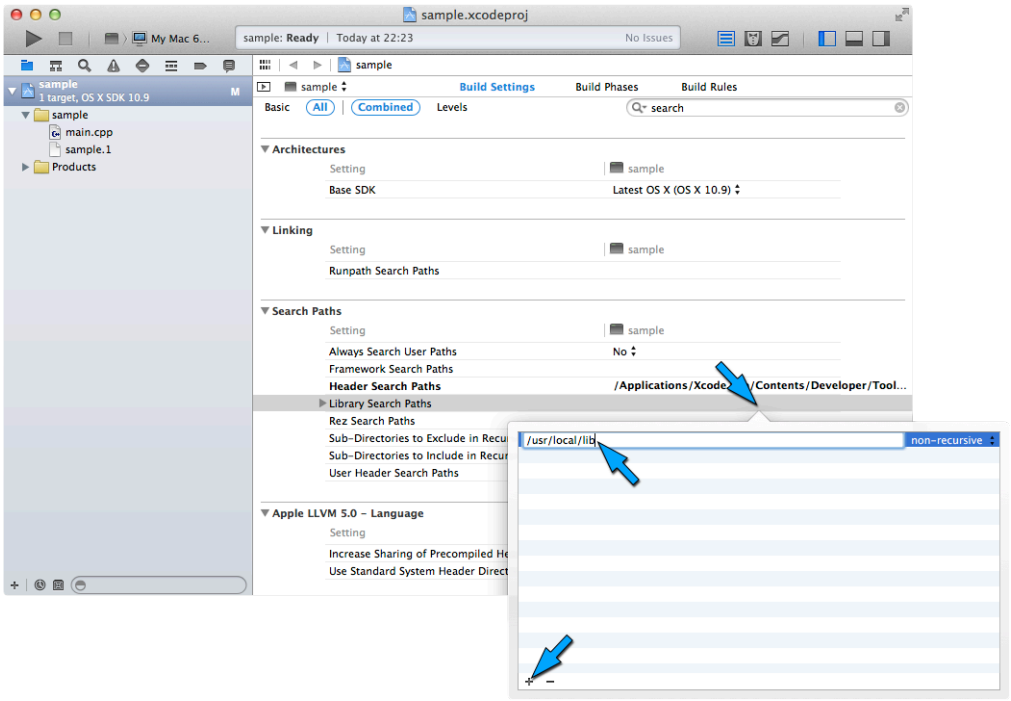

図 52 ライブラリファイルの検索パスの設定

リンクするライブラリとフレームワークを設定します。検索窓に “linker” などを入力して “Other Linker Flags” という項目を探し、その右側をダブルクリックします (図 53)。入力ウィンドウがポップアップしたら、左下の “+” をクリックして欄を追加し、そこに以下の内容を入力してください。これは 1 行で入力しても、ひとつずつ欄を作っても構いません。

```
-lglfw3 -framework Cocoa -framework IOKit -framework OpenGL -framework CoreVideo
```

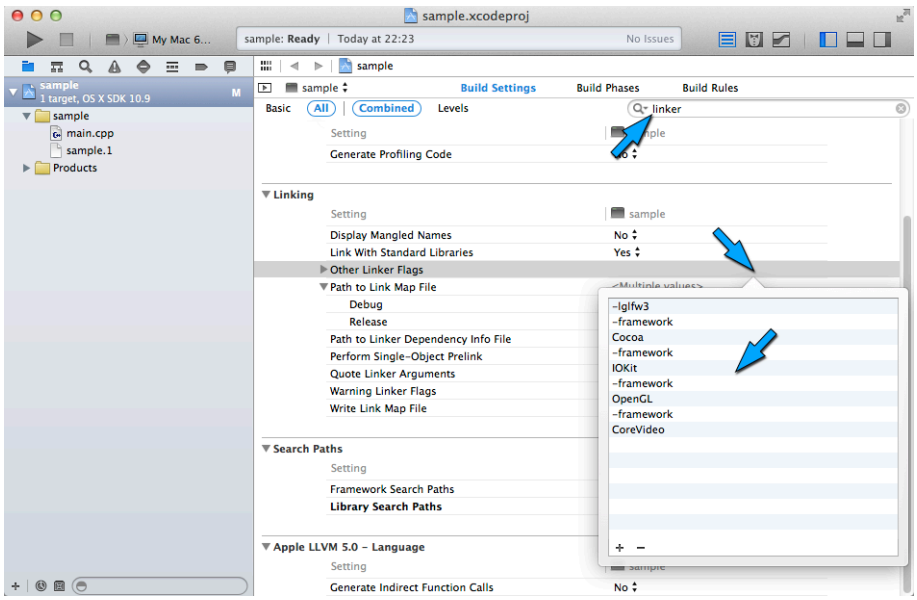

図 53 リンクするライブラリとフレームワークの指定

このテンプレートでは “main.cpp” というファイル名のソースファイルが自動的に作成されます (図 54)。これに main() 関数が定義されています。また “プロジェクト名.1” は Unix や Linux で使われる man 形式のマニュアルのソースファイルです。本書ではこれは使わないので、右クリックして選択して “Delete” を選んで削除しても構いません。

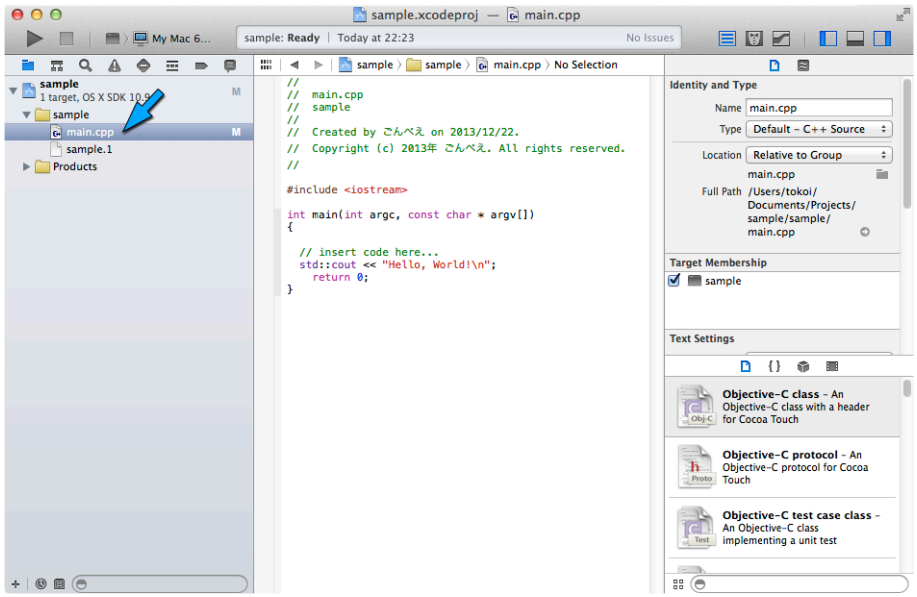

図 54 ソースプログラムの編集

● Makefile を作る

Xcode を使用せず、シェルでコマンドを使ってプログラムを作成する場合は、Makefile を用意しておくと手間が省けます。まず、mkdir コマンドなどを使って、ソースファイルを置く空のディレクトリをひとつ作成してください。‘%’ はシェルのプロンプトを表します。

```
% mkdir sample
```

次に、テキストエディタを使って以下の内容のファイルを Makefile というファイル名で作成し、このディレクトリに保存してください。また、このファイルの行頭の空白 (網かけの部分) には、スペースではなくタブを使ってください。

```
CXXFLAGS = -I/usr/local/include -Wall
LDLIBS   = -L/usr/local/lib -lglfw3 -framework Cocoa -framework IOKit ¥
         -framework OpenGL -framework CoreVideo
OBJECTS  = $(patsubst %.cpp,%.o,$(wildcard *.cpp))
TARGET   = sample

.PHONY: clean

$(TARGET): $(OBJECTS)
        $(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@

clean:
        -$(RM) $(TARGET) $(OBJECTS) *~ .*~ core
```

このファイルを作成しておけば、このディレクトリで make コマンドを実行することにより、ソースプログラムがコンパイル・リンクされて、実行プログラムが作成されます。

3.2.3 Linux

● Geany を使う

Linux にもさまざまな統合開発環境があり、また、本書の執筆時点ではどれが主流というわけでもなさそうです。テキストエディタとコマンドラインコンパイラだけで開発することも可能なのですが、ここではシンプルで軽量な開発環境の Geany (http://www.geany.org) の例を紹介します。

ターミナルを開き、次のコマンドで Geany を起動して、テキストファイルを新規作成します。この例では main.cpp というファイル名のソースファイルを作成します。'%' はシェルのプロンプトを表します。

```
% geany main.cpp
```

あるいは、Geany の「ファイル」メニューから新規作成することもできます。C++ のソースファイルであれば、「テンプレートから新規作成」の main.cxx を選びます (図 55)。拡張子が ".cxx" になってしまいますが、Linux ではこれも C++ のソースファイルの拡張子として認識されます。

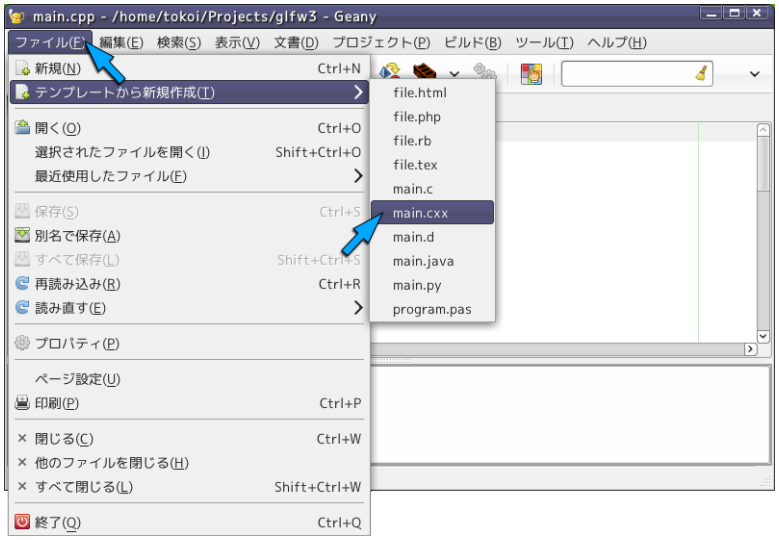

図 55 ソースファイルの新規作成

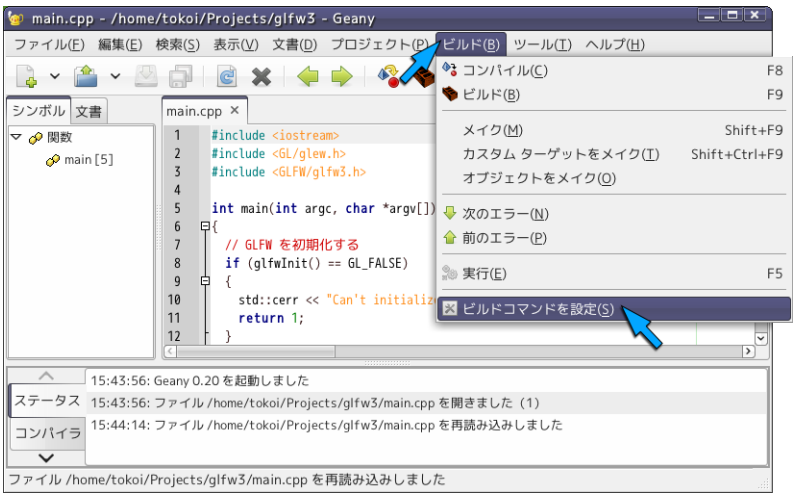

図 56 ビルドコマンドの設定ウィンドウの呼び出し

次に、コンパイルオプションの設定を行います。Geany の「ビルド」メニューから「ビルドコマンドを設定」を選んでください (図 56)。「ビルドコマンドを設定」のウィンドウの「ビルド」ラベルのコマンドの欄 (図 57) に、既に入力されているものの後にスペースをあけて次の内容を追加してください。-lm は GLFW あるいは OpenGL では利用されませんが、本書のプログラムでは数学ライブラリを使うので、ここで追加しておきます。

```
-lGL -lglfw3 -lXi -lXrandr -lXxf86vm -lX11 -lrt -lpthread -lm
```

設定が終わったら「OK」をクリックしてください。

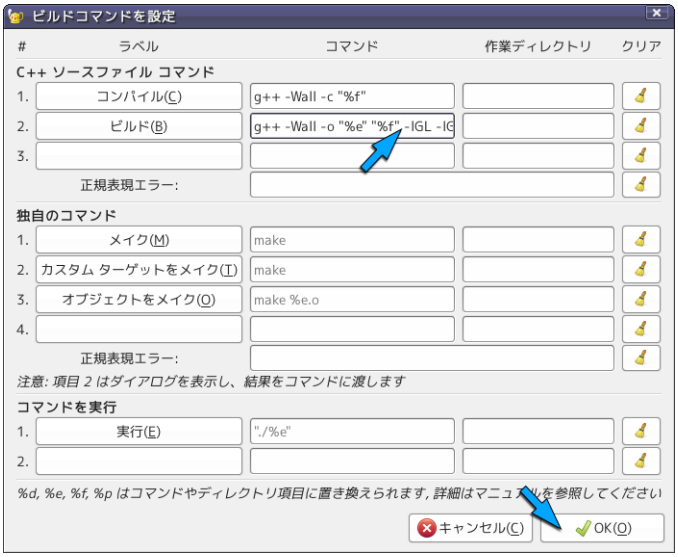

図 57 ビルドコマンドを設定

● Makefile を作る

Geany などの統合開発環境を使用せず、シェルでコマンドを使ってプログラムを作成する場合は、Makefile を用意しておくと手間が省けます。まず、mkdir コマンドなどを使って、ソースファイルを置く空のディレクトリを作成してください。'%' はシェルのプロンプトを表します。

```
% mkdir sample
```

次に、テキストエディタを使って以下の内容のファイルを Makefile というファイル名で作成し、このディレクトリに保存してください。また、このファイルの行頭の空白 (網かけの部分) には、スペースではなくタブを使ってください。

```
CXXFLAGS = -Wall
LDLIBS   = -lGL -lglfw3 -lXi -lXrandr -lXxf86vm -lX11 -lrt -lpthread -lm
OBJECTS  = $(patsubst %.cpp,%.o,$(wildcard *.cpp))
TARGET   = sample

.PHONY: clean

$(TARGET): $(OBJECTS)
	$(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@

clean:
	-$(RM) $(TARGET) $(OBJECTS) *~ .*~ core
```

このファイルを作成しておけば、このディレクトリで make コマンドを実行することにより、ソースプログラムがコンパイル・リンクされて、実行プログラムが作成されます。

3.3 ソースプログラムの作成

3.3.1 処理手順

この講義では、講義内容をもとに実際にプログラムを作成し、講義内容の理解と定着を図ります。理論的な部分は OpenGL を用いて実装しますが、それを実際に動作させるためにはアプリケーションプログラムとしての枠組みが必要になります。そのためのツールキットとして、この講義では GLFW を採用します。GLFW を使ったプログラムの処理手順は次のようになります。

(1) GLFW を初期化する (glfwInit())
(2) ウィンドウを作成する (glfwCreateWindow())
(3) ウィンドウが開いている間くり返し描画する (glfwWindowShouldClose())
(4) ダブルバッファリングのバッファの入れ替えを行う (glfwSwapBuffers())
(5) ウィンドウが閉じたら終了処理を行う (glfwTerminate())

そこで、最初に「最小の」C++ のプログラムを考えてみます。以下の網かけの部分を (使用するソフトウェア開発環境の) テキストエディタに打ち込んでください。

```
int main()
{
}
```

このプログラムは、プログラムのエントリポイント (プログラムの実行を開始する場所) である main() 関数しかなく、その中身も空なので、実行しても何も起こらずに終了します。

3.3.2 GLFW の初期化

main() 関数に GLFW の初期化処理を追加します。ソースプログラムの冒頭で GLFW のヘッダファイル GLFW/glfw3.h を #include し、main() 関数の最初の部分、すなわちプログラムの実行開始直後に glfwInit() 関数を実行します。これにより、このプログラムで OpenGL を使用するための準備が行われます。glfwInit() 関数の戻り値が GL_FALSE のときは GLFW の初期化に失敗していますから、エラーメッセージを出してプログラムを終了するようにします。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }
}
```

int glfwInit(void)

GLFW を初期化します。他の全ての GLFW の関数を実行する前に実行する必要があります。初期化に成功すれば GL_TRUE、失敗すれば GL_FALSE を返します。

3.3.3 ウィンドウを作成する

GLFW の初期化が成功したら、glfwCreateWindow() 関数を使って OpenGL による描画を行うウィンドウを作成します。glfwCreateWindow() 関数の戻り値は、レンダリングコンテキストと呼ばれるウィンドウ固有の情報を含むオブジェクトのポインタです。ここで何らかの理由によりウィンドウが作成できなかった場合には、これが NULL になりますので、そのときはエラーメッセージを表示してプログラムを終了するようにしておきます。なお、この時点では既に glfwInit() による GLFW の初期化が完了していますので、プログラムを終了する直前、すなわち return 文の前で、glfwTerminate() 関数を実行して GLFW の終了処理を行います。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }
}
```

GLFWwindow *glfwCreateWindow(int width, int height, const char *title, GLFWmonitor *monitor, GLFWwindow *share)

GLFW のウィンドウを作成します。戻り値は作成したウィンドウのハンドルです。ウィンドウが開けなければ NULL を返します。

width

作成するウィンドウの横幅の画素数で、0 より大きくなければなりません。

height

作成するウィンドウの高さの画素数で、0 より大きくなければなりません。

title

作成するウィンドウのタイトルバーに表示する文字列です。文字コードは UTF-8 です。

monitor

ウィンドウをモニタ (ディスプレイ) の全面に表示するとき (フルスクリーンモード)、表示するモニタを指定します。フルスクリーンモードでなければ NULL を指定します。

share

引数 share に他のウィンドウのハンドルを指定すれば、そのウィンドウとテクスチャなどのリソースを共有します。NULL を指定すれば、リソースの共有は行いません。

void glfwTerminate(void)

GLFW の終了処理を行います。glfwInit() 関数で GLFW の初期化に成功した場合は、プログラムを終了する前に、この関数を実行する必要があります。この関数は GLFW で作成した全てのウィンドウを閉じ、確保した全てのリソースを解放して、プログラムの状態を glfwInit() 関数で初期化する前に戻します。この後に GLFW の機能を使用するには、再度 glfwInit() 関数を実行しなければなりません。

3.3.4 作成したウィンドウを処理対象にする

ウィンドウを作成することに成功したら、glfwMakeContextCurrent() 関数の引数にそのハンドルを指定して、そのウィンドウのレンダリングコンテキストを処理の対象にします。開いたウィンドウに対する設定や図形の描画などは、この後に行います。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);
}
```

void glfwMakeContextCurrent(GLFWwindow *const window)

引数 window に指定したハンドルのウィンドウのレンダリングコンテキストをカレント (処理対象) にします。レンダリングコンテキストは描画に用いられる情報で、ウィンドウごとに

保持されます。図形の描画はこれをカレントに設定したウィンドウに対して行われます。

window

OpenGL の処理対象とするウィンドウのハンドルを指定します。

3.3.5 OpenGL の初期設定

glfwMakeContextCurrent() 関数で OpenGL による描画を行うウィンドウを指定すれば、ようやく OpenGL の機能が使用できるようになります。ここでは glClearColor() 関数により画面を消去する色を指定します。表示領域を消去する色 (背景色) は、ここでは白にします。なお、関数名が gl〜 で始まるものは OpenGL の API です (GLFW の関数名は glfw〜 で始まります)。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
}
```

void glClearColor(GLclampf R, GLclampf G, GLclampf B, GLclampf A)

glClear(GL_COLOR_BUFFER_BIT) でウィンドウを塗り潰す色を指定します。

R，G，B

それぞれ赤、緑、青色の成分の強さを示す GLclampf 型 (float 型と等価) の値で、0〜1 の値を持ちます (図 58)。1 が最も明るく、この三つにそれぞれ 0, 0, 0 を指定すれば黒色、1, 1, 1 を指定すれば白色になります。

A

アルファ値と呼ばれ、OpenGL では不透明度として扱われます (0 で透明、1 で不透明)。ここではとりあえず 0 にしておきます。

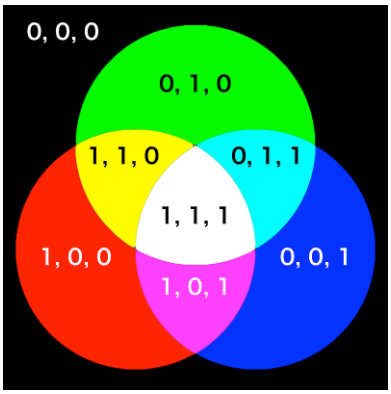

図 58 R, G, B の値と色

3.3.6 メインループ

ウィンドウを開いても、このままではすぐに main() 関数の最後に到達して、プログラムが終了してしまいます。そこで、ウィンドウが閉じられなければプログラムが終了しないように、while によって処理を繰り返します。ウィンドウが閉じられたかどうかは、glfwWindowShouldClose() const 関数で調べることができます。

このループの中では、最初に glClear() 関数を使って画面の表示領域を消去します。その後、そこに OpenGL により図形の描画を行います。描画が終わったら glfwSwapBuffers() 関数を実行して、図形を描画したカラーバッファと現在図形を表示しているカラーバッファを入れ替えます。この処理はダブルバッファリングといいます。

最後に、このプログラムが次に何をすべきか判断するために、この時点で発生しているイベントを調査します。glfwWindowShouldClose() const によるウィンドウを閉じるべきかどうかの判断も、このイベントの調査にもとづいて行われます。イベントの調査には、マウスなどで操作する対話的なアプリケーションの場合は、イベントが発生するまで待つ glfwWaitEvents() 関数を用います。これに対して、時間とともに画面の表示を更新するアニメーションなどの場合は、イベントの発生を待たない glfwPollEvents() 関数を用います (表 2)。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
```

```
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

int glfwWindowShouldClose(GLFWwindow *const window)

window に指定したウィンドウを閉じる必要があるとき、戻り値は非 0 になります。

void glClear(GLbitfield mask)

ウィンドウを塗り潰します。mask には塗り潰すバッファを指定します。フレームバッファは色を格納するカラーバッファのほか、隠面消去処理に使うデプスバッファ、図形の型抜きを行うステンシルバッファなどの複数のバッファで構成されており (図 26)、これらがひとつのウィンドウに重なっています。mask に GL_COLOR_BUFFER_BIT を指定したときは、カラーバッファだけを glClearColor() 関数で指定した色で塗り潰します。

void glfwSwapBuffers(GLFWwindow *const window)

window に指定したウィンドウのカラーバッファを入れ替えます。図形を描画した後にこの関数を実行しなければ、描画したものは画面に表示されません。

void glfwWaitEvents(void)

マウスの操作などのイベントの発生を待ちます。イベントが発生したら、それを記録してプログラムの実行を再開します。アニメーションのように画面表示を常に更新し続けるような必要がなければ、コンピュータの負荷を低減するために、この関数を用います。ただし、この関数はメインのループ以外で実行すべきではありません。

void glfwPollEvents(void)

マウスの操作などのイベントを取り出し、それを記録します。この関数はプログラムを停止させないので、アニメーションのように連続して画面表示を更新する場合に使用します。

表 2 イベントの取り出し

glewWaitEvents()	glewPollEvents()
イベントが発生するまで待つ (プログラムの実行を停止する)。イベントが発生すれば、最初のイベントを取り出してプログラムの実行を再開する。	イベントが発生していれば、それを取り出す。イベントの有無にかかわらず、次に進む (プログラムの実行を停止しない)。

● 補足：バッファについて

バッファは一般に緩衝装置と訳されます。これは何か二つのものが接続されており、一方からもう一方に何らかの影響を与えるような状況にあるとき、この二つの間に入って影響の仲立ちをするもののことをいいます。

OpenGL におけるバッファは、データを次の処理に引き渡すために用いる、メモリのことを指します。OpenGL ではいろいろな種類のバッファを使用しますが、ここで用いているバッファは、描画した図形を画面に表示するために用いるフレームバッファのカラーバッファです。

図形はフレームバッファのカラーバッファに描画されます。画面上への表示は、このカラーバッファの内容を読み出しながら映像信号を発生することにより行います。ここでフレームバッファへの描画と読み出しを同時に行うと、表示にちらつきが発生してしまいします。そこでカラーバッファを二つ用意しておいて、一方を表示している間にもう一方に描画するようにします。そして、描画が完了した時点でこの二つのカラーバッファを入れ替えます (図 59)。この処理をダブルバッファリングといいます。

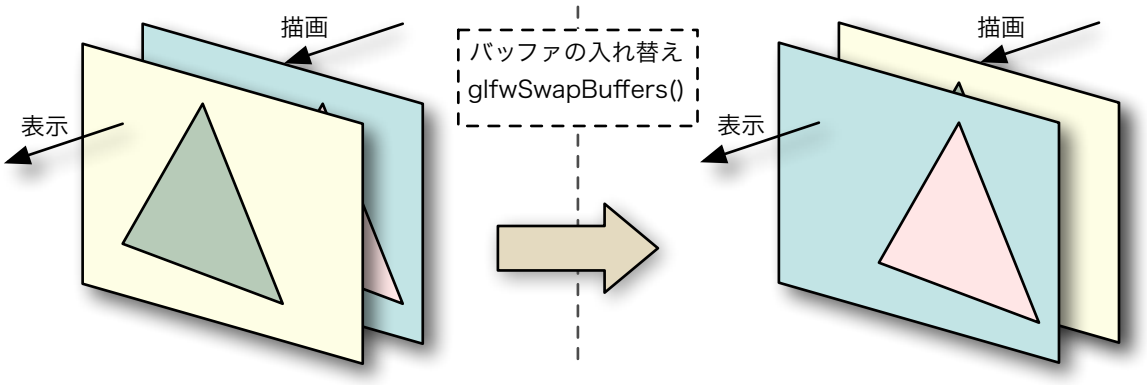

図 59 ダブルバッファリング

● 補足：イベントについて

画面上に複数のウィンドウが存在する場合、図形を表示しているウィンドウの上に重なっている他のウィンドウが閉じられた場合には、下のウィンドウの表示内容を描き直す必要があります。また対話的なアプリケーションでは、マウスなどの操作に対応した処理を随時実行する必要があります。このように、ある処理の実行のきっかけとなる出来事を、イベントと呼びます。

発生したイベントに対応する処理の実行方法には、画面表示のたびに glfwWaitEvents() 関数や glfwPollEvents() 関数を用いてイベントを取り出す方法 (ポーリング方式) と、特定のイベントが発生したときに実行する関数をあらかじめ登録しておく方法 (コールバック方式) があります。

3.3.7 終了処理

glfwInit() 関数による初期化に成功していれば、プログラムを終了する前に glfwTerminate() 関数を実行する必要があります。main() 関数の最後の return 文の直前に glfwTerminate() 関数の呼び出しを追加します。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }

  glfwTerminate();
}
```

● 補足：atexit() による glTerminate() の実行

プログラムの終了時には必ず glTerminate() 関数を実行する必要があります。そのため、このプログラムでは main() 関数の最後の return 文の前のほか、glfwCreateWindow() 関数のエラー処理のところでも実行しています。

しかしプログラムを追加していくと、これら以外の場所でもプログラムを終了させなければならない場合が発生するかも知れません。そのようなとき、プログラムのあちこちで glTerminate() 関数を実行することは無駄に思われますし、呼び出しを忘れる可能性もあります。そこで、プログラムの終了時に必ず glTerminate() 関数が実行されるように、atexit() 関数を用います。

プログラムの先頭で atexit() 関数を定義しているヘッダファイル cstdlib を #include し、glfwTerminate() 関数を実行する関数 cleanup() を定義します。そして glfwInit() 関数が成功した後で、atexit() 関数により cleanup() 関数を登録します。また、既に main() 関数に埋め込んでいた glfwTerminate() 関数は削除します。

```
#include <cstdlib>
#include <iostream>
#include <GLFW/glfw3.h>

// プログラム終了時の処理
static void cleanup()
{
  // GLFW の終了処理
  glfwTerminate();
}

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(cleanup);

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

```
  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

int atexit(void (*function)(void))

atexit() 関数は、引数 function に指定した関数を、プログラム終了時に実行するよう登録します。atexit() 関数を複数回呼び出して、複数の関数を登録することができます (少なくとも 32 個の関数が登録できます)。その場合、関数は登録した順の逆順に実行されます。戻り値として、関数の登録に成功した時は 0、失敗した時は 0 以外の値を返します。

3.3.8 OpenGL のバージョンとプロファイルの指定

この講義では OpenGL 3.2 以降の機能を使用してプログラムを作成します。そのため、OpenGL のバージョンやプロファイルを指定してウィンドウを作成します。これは glfwCreateWindow() 関数でウィンドウを作成する前に、glfwWindowHint() 関数を用いて行います。

```
《省略》

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(cleanup);

  // OpenGL Version 3.2 Core Profile を選択する
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
```

```
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

《省略》
```

void glfwWindowHint(int target, int hint)

この後に glfwCreateWindow() 関数によって作成するウィンドウの特性を設定します。デフォルトの設定に戻すには glfwDefaultWindowHints() を呼び出します。

target

ヒントを設定する対象。以下のものが指定できます (一部を抜粋)。

GLFW_RED_BITS, GLFW_GREEN_BITS, GLFW_BLUE_BITS, GLFW_ALPHA_BITS

それぞれカラーバッファの赤色・緑色・青色・アルファに割り当てるビット数を hint に指定します。デフォルトはいずれも 8 です。

GLFW_DEPTH_BITS

デプスバッファに割り当てるビット数を hint に指定します。デフォルトは 24 です。

GLFW_STENCIL_BITS

ステンシルバッファに割り当てるビット数を hint に指定します。デフォルトは 8 です。

GLFW_SAMPLES

hint にマルチサンプル時のサンプル数を指定します。0 を指定するとマルチサンプルが無効になります。デフォルトは 0 です。

GLFW_STEREO

hint に GL_TRUE を指定すればステレオモードになります。デフォルトは GL_FALSE です。これを GL_TRUE にできるかどうかは、ハードウェアに依存します。

GLFW_CLIENT_API

hint に GLFW_OPENGL_ES_API を指定すれば OpenGL ES の API を使用します。デフォルトは GLFW_OPENGL_API です。

GLFW_CONTEXT_VERSION_MAJOR

OpenGL のメジャーバージョン番号を hint に指定します。バージョンが 3.2 ならば 3 です。デフォルトは 1 です。

GLFW_CONTEXT_VERSION_MINOR

OpenGL のマイナーバージョン番号を hint に指定します。バージョンが 3.2 ならば 2 です。デフォルトは 1 です。

GLFW_OPENGL_FORWARD_COMPAT

OpenGL の Forward Compatible Profile (前方互換プロファイル、古い機能が使えない) を使う場合は、hint に GL_TRUE を指定します。デフォルトは GL_FALSE です。

GLFW_OPENGL_PROFILE

使用する OpenGL のプロファイルを指定します。Compatible Profile を使う場合は hint に GLFW_OPENGL_COMPAT_PROFILE 、 Core Profile を使う場合は hint に GLFW_OPENGL_CORE_PROFILE を指定します。デフォルトは 0 で、この場合はシステムの設定に依存します。

グラフィックスハードウェアが glfwWindowHint() 関数で指定した特性に対応していなければ、その後の glfwCreateWindow() 関数の実行は失敗してウィンドウは開かれません。

3.3.9 作成したウィンドウに対する設定

一方、作成したウィンドウに対する設定は、glfwCreateWindow() 関数によるウィンドウの作成に成功した後に行います。ここでは glfwSwapInterval() 関数によって、ダブルバッファリングにおけるバッファの入れ替えタイミングを設定します。

```
《省略》

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(cleanup);

  // OpenGL Version 3.2 Core Profile を選択する
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 作成したウィンドウに対する設定
  glfwSwapInterval(1);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

《省略》

void glfwSwapInterval(int interval)

ダブルバッファリングにおける、カラーバッファの入れ替えのタイミングを指定します。

interval

少なくとも interval に指定した回数だけディスプレイの垂直同期タイミング (V-Sync、帰線消去期間) を待ってから、カラーバッファの入れ替えを行います。普通は 1 を指定します。0 を指定するとディスプレイの垂直同期タイミングを待たなくなるため、数値の上では fps (frame per second) が上昇しますが、完全な画面表示が行われるわけではありません。

3.3.10 補助プログラム

3.3.8 節において、OpenGL のバージョンとプロファイルに OpenGL 3.2 Core Profile を使うように設定したので、従来の OpenGL の固定機能が使えません。図形を描画するためには、シェーダプログラムを作成する必要があります。

加えて、プラットホームによってはグラフィックスハードウェアが備えているすべての機能の API を用意していない場合があります (Windows に標準搭載されている OpenGL はバージョン 1.1 までの API しか対応していません)。その場合は、グラフィックスハードウェアのドライバから足らない API のエントリポイントを取り出してくる必要があります。これには GLEW (The OpenGL Extension Wrangler Library, http://glew.sourceforge.net) という便利なライブラリがあるのですが、ここでは自前で準備することにします。

なお、Windows / Mac OS X / Linux でソースを共通化するためには他にも追加のコードが必要になりますが、ここではそれは本題ではないので、この部分を gg.cpp と gg.h という二つのファイルにまとめています。これらのファイルをソースファイルと同じディレクトリ (フォルダ) に置き、プロジェクトに追加してください。

また、これらの他に glext.h (http://www.opengl.org/registry/api/glext.h) もダウンロードして、ソースプログラムと同じディレクトリ (フォルダ) に入れてください。

● 追加するファイル

- gg.cpp
- gg.h
- glext.h

そして #include <GL/glfw.h> の一行を #include "gg.h" と using namespace gg; の二行に置き換え、glfwInit() から glfwSwapInterval(1) までを ggInit() に置き換えてください。なお、ggInit() の引数は glfwOpenWindow() と同じで、デフォルト引数としてこれまでの設定と同じ (0, 0, 0, 0, 0, 0, 0, 0, 0, GLFW_WINDOW) を与えています。

● ソースプログラムの変更点

```
#include <cstdlib>
#include <iostream>
```

```
#include <GLFW/glfw3.h>

// 補助プログラム
#include "gg.h"
using namespace gg;

// プログラム終了時の処理
static void cleanup()
{
  // GLFW の終了処理
  glfwTerminate();
}

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(cleanup);

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 作成したウィンドウに対する設定
  glfwSwapInterval(1);

  // 補助プログラムによる初期化
  ggInit();

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
```

```
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

● 補足：補助プログラムを使わない場合

この講義では補助プログラムの使用を前提に説明を行いますが、補助プログラムを用いないでプログラムを作成する場合は、次のように設定してください。

OpenGL 3.2 の Core Profile では、gluLookAt() などを含む GLU (OpenGL Utility) ライブラリのいくつかの機能が使えないので、GLU を使わないよう GLFW_NO_GLU を #define します。

Windows

補助プログラムを使わない場合、OpenGL 1.2 以降の機能をアプリケーションプログラムから呼び出せるようにするためには、GLEW を使用すると便利です。glfwCreateWindow() で GLFW のウィンドウを開いた後、「補助プログラムによる初期化」を行っている ggInit() の代わりに、glewExperimental = GL_TRUE; という代入を行った後で glewInit() を呼び出してください。

```
#incoude <GL/glew.h>
#incoude <GLFW/glfw3.h>

...

  // 補助プログラムによる初期化
  glewExperimental = GL_TRUE;
  if (glewInit() != GLEW_OK)
  {
    // エラーメッセージ等
    return 1;
  }
```

Mac OS X

Mac OS X では glfw3.h を #include する前に GLFW_INCLUDE_GLCOREARB を #define しておきます。また gl3ext.h も #include しておきます。

```
#define GLFW_INCLUDE_GLCOREARB
#include <GLFW/glfw3.h>
#include <OpenGL/gl3ext.h>
```

Linux

Linux では X11 にグラフィックスハードウェアメーカーのプロプライエタリドライバがインストールされていることを前提にしています。C++ の場合は GL_GLEXT_PROTOTYPES も #define しておくといいと思います。

```
#define GL_GLEXT_PROTOTYPES
#include <GLFW/glfw3.h>
#include <GL/glext.h>
```

以降では gg.h / gg.cpp の使用を前提に説明します。

3.4 プログラムのビルドと実行

ソースプログラムをコンパイルしてオブジェクトプログラムを生成し、リンクによりオブジェクトプログラム同士やライブラリを結合して実行プログラムを生成する一連の作業をビルドと呼びます。これは簡単なプログラムであればコンパイラのコマンドを用いて実行することができますが、プログラムが複雑になるとコンパイラのコマンドだけでビルドするのは手間がかかります。そこで、統合開発環境や make コマンドなどの補助的な手段を用いてビルドします。

3.4.1 Windows

Visual Studio 2013 でビルドするには、「ビルド」メニューから「ソリューションのビルド」を選んでください (図 60)。その下の「“プロジェクト名” のビルド」でも構いません。

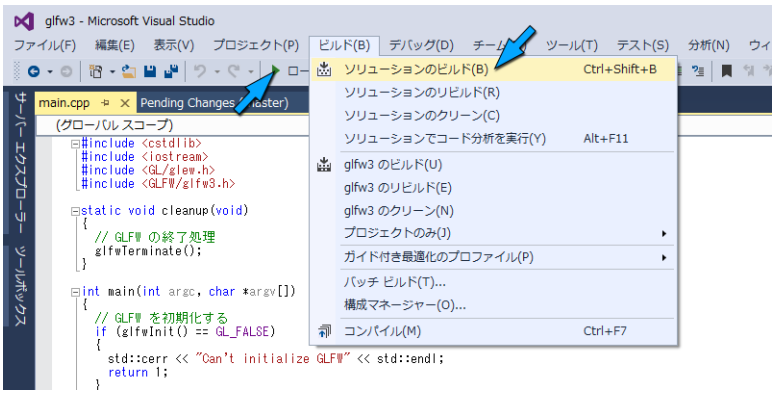

図 60 Visual Studio 2013 によるプログラムのビルド

プログラムの実行は、「デバッグメニュー」の「デバッグ開始」を選ぶか、ファンクションキーの F5 をタイプしてください。ツールバーにある「緑色の右向き三角▶」をクリックしてもデバッグ実行を開始します。ソースプログラムの修正後、ビルドせずにプログラムを実行した時は、先にビルドを実行します。

なお、ソリューション構成が「Debug」のとき、ビルド時に「LINK : warning LNK4098: defaultlib 'MSVCRT' は他のライブラリの使用と競合しています。/NODEFAULTLIB:library を使用してください。」という警告が出ることがあります。これはここでは無視してください。GLFW をダイナミックリンクすれば、この警告を抑制できます。

3.4.2 Mac OS X

Xcode では、左上の黒い右向き三角▶をクリックすれば、プログラムのビルドと実行を続けて行います (図 61)。Command キーを押しながら R のキーをタイプしても同様です。

図 61 Xcode によるプログラムのビルドと実行

コマンドラインでビルドする場合は、c++ コマンドに -I/usr/local/include -L/usr/local/lib -lglfw3 -framework Cocoa -framework IOKit -framework OpenGL -framework CoreVideo オプションを追加してください。‘%’ はシェルのプロンプトを表します。

```
% c++ main.cpp -I/usr/local/include -L/usr/local/lib -lglfw3 ¥
-framework Cocoa -framework IOKit -framework OpenGL -framework CoreVideo
```

さすがに長過ぎるので、Makefile を作って make コマンドを実行するのが賢明だと思います。Makefile を用意していれば、make コマンドを実行するだけです。

```
% make
```

3.4.3 Linux

Geany の場合は、「ビルド」メニューの「ビルド」を選んでください (図 62)。ビルドされたプログラムを実行するには、同じ「ビルド」メニューの「実行」を選んでください。これらはそれぞれファンクションキーの F9 と F5 でも実行できます。また、ツールバー上にも対応するボタンがあります。

図 62 Geany によるプログラムのビルドと実行

コマンドラインでビルドする場合は、c++ コマンドに -lGL -lglfw3 -lXi -lXrandr -lXxf86vm -lX11 -lrt -lpthread -lm オプションを追加してください。‘%’ はシェルのプロンプトを表します。

```
% c++ main.cpp -lGL -lglfw3 -lXi -lXrandr -lXxf86vm -lX11 -lrt -lpthread -lm
```

長いので、Makefile を作って make コマンドを使用することを勧めます。Makefile を用意していれば、make コマンドを実行するだけです。

```
% make
```