OpenGL Performance Tuning

Nihon Silicon Graphics K.K.
Sales Division Graphics Grp
Systems Engineer

Takehiko Terada

1995/07/20 First edition
1995/08/31 Second edition

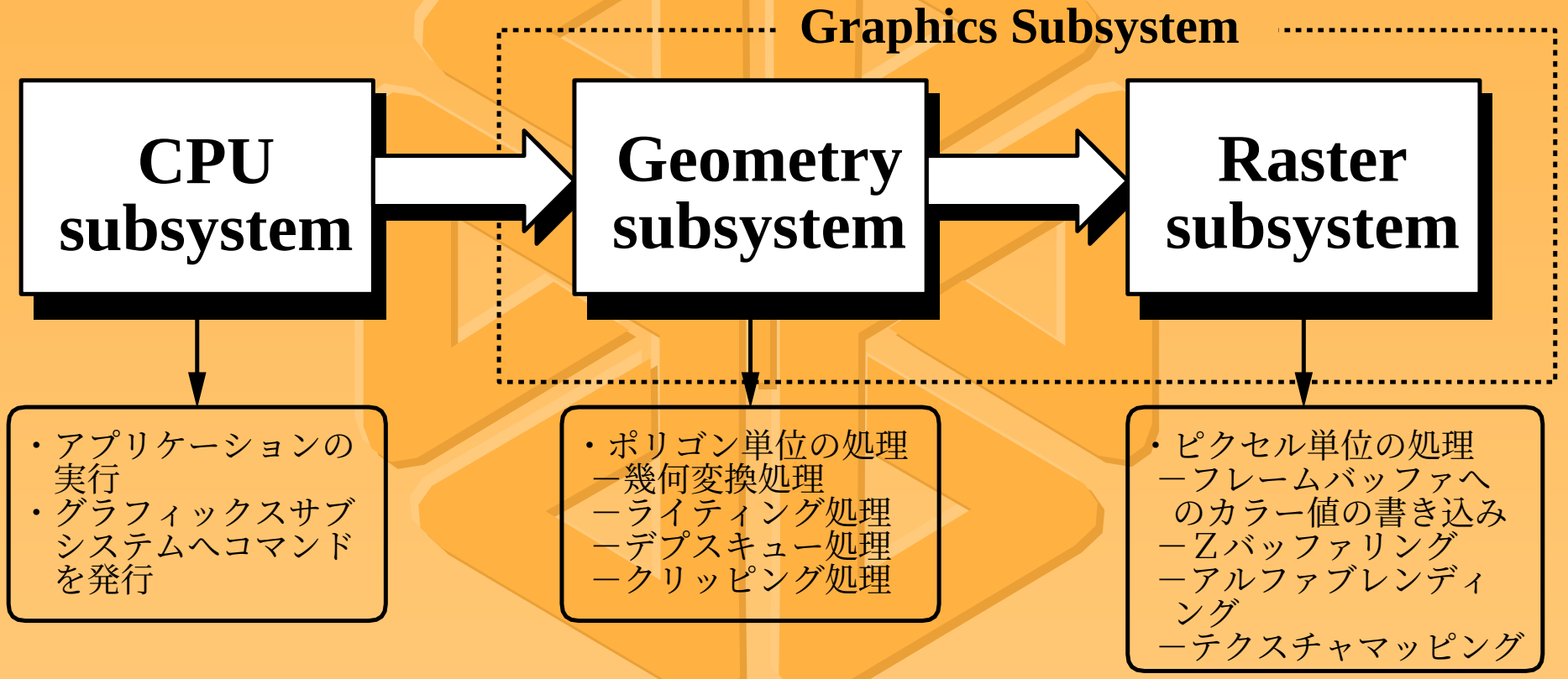
Graphics Performance Tuning

- チューニングの有用性
  －マシンの潜在能力を引き出すにはチューニング(最適化)をする必要がある。
- 描画速度を決定する要素

  －アプリケーションのオーバーヘッド
  －１フレーム内に描画されるポリゴン数
  －現在のモードにおける幾何変換速度
  －画面上のピクセル数
  －現在のモードにおけるピクセル塗りつぶし速度
  －表示画面とZバッファのクリア速度
  －バッファスワップにかかる時間

ハードウェアの構成

- **Graphics Pipelineの構成**
  (Geometry)



パイプラインのチューニング

- ボトルネックの有無を調べる
  - 通常のアプリケーションのチューニングでは**10%**のコードで**90%**の時間を消費するホットスポットを探し、チューニングする。
  - グラフィックスパイプラインのチューニングでは過負荷になっているボトルネックを探す。

- ボトルネックの特定
  - ボトルネックがあった場合には、どのサブシステムのボトルネックであるか特定する。

ボトルネックの種類

- **CPU**ボトルネック
- ポリゴン処理におけるボトルネック
  (Geometry sybsystem)
- ピクセル処理におけるボトルネック
  (Raster sybsystem)

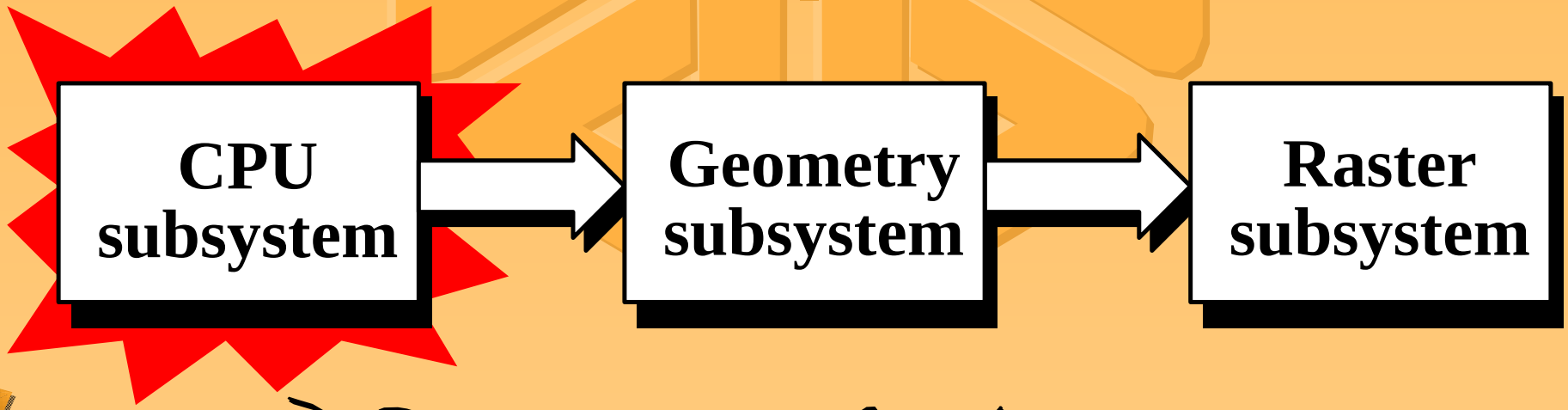
ボトルネックの種類（続き）

・CPUボトルネックの発見方法

－アプリケーションプログラムがグラフィックスサブシステムを十分に活かしていない場合に起こる。このようなプログラムを ***CPU-limited*** と呼ぶ。

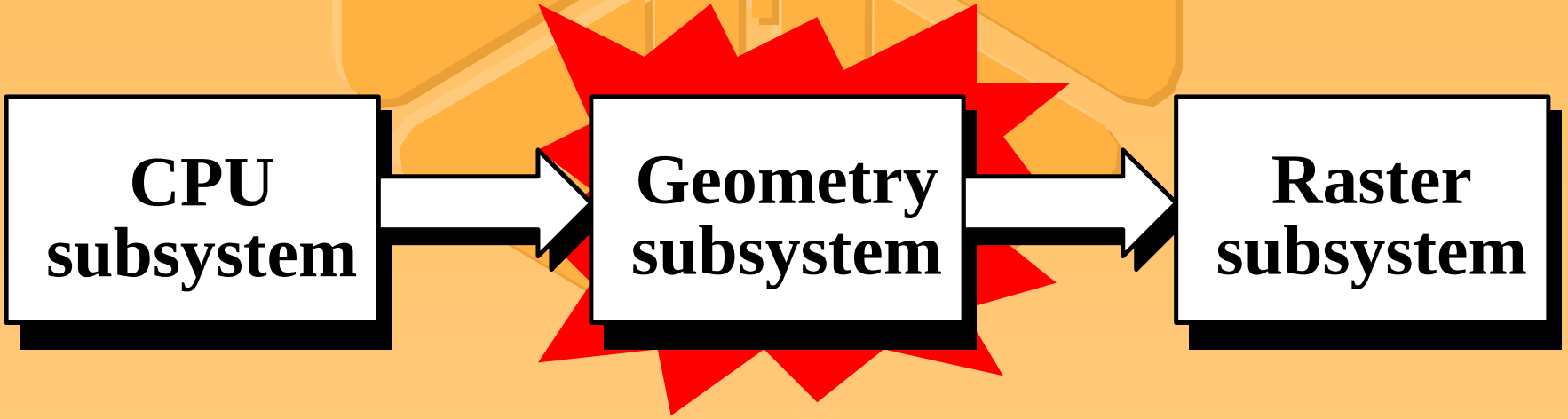
－**CPU**ボトルネックをテストするためには、アプリケーション中のグラフィックスに関する部分を出来るだけ削ってみてその実行速度を計測してみる。もしこれを行ってもアプリケーションの実行速度に劇的な変化が見られないときには、そのアプリケーションは**CPU**にボトルネックがある可能性が高い。

－大抵の場合、これは**OpenGL**の関数をほんの少しだけ変えるだけで行うことができる。例えば、アプリケーション中の全ての**glVertex3fv()**と**glNormal3fv()**を**glColor3fv()**に置き換えることで、アプリケーションにほとんど手を加えること無くグラフィックス部分の抜き出し行うことができる。



ボトルネックの種類（続き）

- ポリゴン処理におけるボトルネックの発見方法
  - **Geometry subsystem**で発生するポリゴン処理段階でのボトルネックを起こすプログラムを ***transform−limited*** という。多数の小さなポリゴンやラインを描いた時に起こりやすい。
  - このボトルネックを発見するには、アプリケーション部分には手をつけず、画面上の塗りつぶされるピクセル数も変えずにポリゴンに対する処理のみを減すことで特定できる。
  - 例えばもしアプリケーションがライティングを使用しているのなら、**glDisable(GL_LIGHTING)** でライティングを一時的にやめてアプリケーションの実行速度を計測する。その結果、パフォーマンスが向上したら、そのアプリケーションはポリゴン処理段階にボトルネックがある可能性が高い。



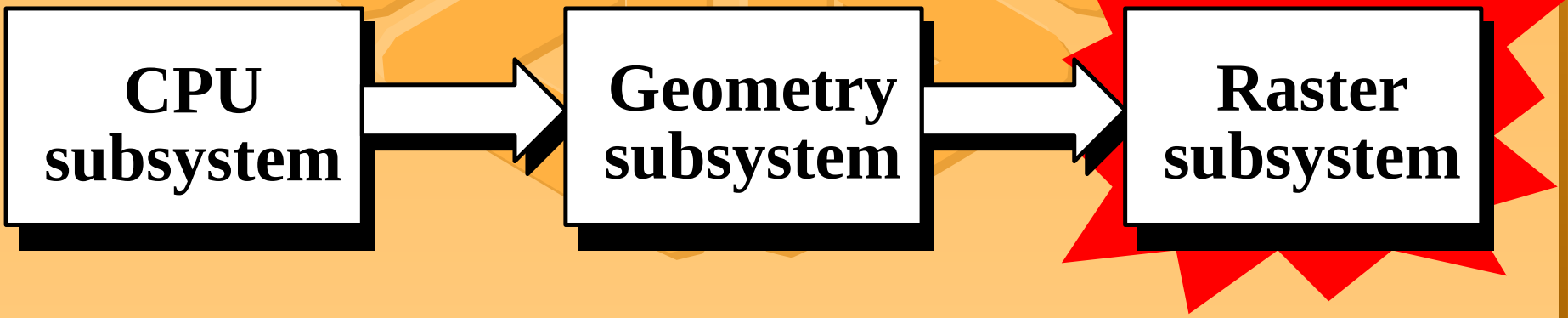
ボトルネックの種類（続き）

- ピクセル処理におけるボトルネックの発見方法

—**Raster subsystem**で発生するピクセル処理段階でのボトルネックを起こすプログラムを***fill-rate limited***という。大きなポリゴンなどを描いた時などに起こりやすい。

—このボトルネックは、表示するオブジェクトまたはウィンドウのサイズを縮小して、画面上に描画されるピクセル数を減らすことによってテストすることができる。もしオブジェクトやウィンドウの大きさがアプリケーションにとって重要なファクターである場合にはこの方法は使用できない。その場合には**Z**バッファリングやアルファブレンディングといったピクセル毎に行われる処理を減すことによっても判定ができる。

—もしそれらの処置を施すことによってパフォーマンスが向上したら、そのアプリケーションはピクセル処理段階にボトルネックがある可能性が高い。



CPUチューニング（基礎）

- オプティマイズオプションをつけてコンパイルする

  ―**’-O2’**をつけてコンパイルする。（**R8000**シリーズの場合は**’-O3’**をつけてソフトウェアパイプラインを有効にする。）
  ―デバッグのため**’-g’**をつけると全てのオプティマイズが無効になるので注意！！
  ―オプティマイズをかけてデバッグを行うときは**’-g3’**と**’-O2’**を両方つける。（但し制約あり）
  ―浮動小数点演算を高速化したいときには**’-float’**オプションをつけてコンパイルする。（**K&R**）

- データ構造を単純化する

  ―データベースの階層構造化を避ける。
  ―配列を単純化する。（１次元にする）

- データベース検索を最適化する

CPUチューニング（続き）

- イミディエイトモードのチューニング

  －*One-Dimensional array:* １次元の配列を使う。

```
Bad:   glVertex3fv(&data[i][j][k]);
Good:  glVertex3fv(dataptr);
```

  頂点の座標値だけでなく、法線やカラーの値も同じ配列の中に入れるとなお良い。

  例）

```
glBegin(GL_POLYGON);
  glNormal3fv(ptr);
  glVertex3fv(ptr+4);
  glNormal3fv(ptr+8);
  glVertex3fv(ptr+12);
  glNormal3fv(ptr+16);
  glVertex3fv(ptr+20);
  glNormal3fv(ptr+24);
  glVertex3fv(ptr+28);
glEnd();
ptr += 32;
```

CPUチューニング（続き）

- イミディエイトモードのチューニング

—***Adjacent structures***:描画用データはアプリケーションで使用される他のデータとは別けて作成し、なるべく連続領域にデータが置かれるようにする。

—***Flat structures***:構造体を階層構造化せず、データの参照が何重にもなることがないようにする。

```
Bad:   glVertex3fv(object->data->vert);
Ok:    glVertex3fv(dataptr->vert);
Best:  glVertex3fv(dataptr);
```

—***Loop unrolling***:**glVertex?fv()**周りの短いループはなるべく展開する。

```
Bad:
  for (i = 0; i < 4; i++) {
    glColor4ubv(poly_colors[i]);
    glVertex3fv(poly_vert_ptr[i]);
  }

Good:
  glColor4ubv(poly_colors[0]);
  glVertex3fv(poly_vert_ptr[0]);
              :
  glColor4ubv(poly_colors[3]);
  glVertex3fv(poly_vert_ptr[3]);
```

CPUチューニング（続き）

- イミディエイトモードのチューニング（続き）
  - ***Loop unrolling:***ループ中でのポインタや変数の値の操作はなるべく減らす。（ループを展開することでこれらの操作を減らすことができる。）

```
Bad:
  glNormalefv(*(ptr++)); glVertex3fv(*(ptr++));
  または
  glNormal3fv(ptr); ptr += 4;
  glVertex3fv(ptr); ptr += 4;

Good:
  glNormal3fv(*(ptr)); glVertex3fv(*(ptr+1));
  glNormal3fv(*(ptr+2)); glVertex3fv(*(ptr+3));
  または
  glNormal3fv(ptr); glVertex3fv(ptr+4);
  glNormal3fv(ptr+8); glVertex3fv(ptr+12);
```

CPUチューニング（続き）

- イミディエイトモードのチューニング（続き）

—***Loop-buffer access***:ループ中にアクセスするバッファの数を最小限にとどめる。（キャッシュヒット率向上のため）

```
Bad:
  glNormal3fv(normaldata);
  glTexCoord2fv(texdata);
  glVertex3fv(vertdata);

Good:
  glNormal3fv(dataptr);
  glTexCoord2fv(dataptr+4);
  glVertex3fv(dataptr+8);
```

—***Loop end condisions***:ループのパラメータはできるだけ単純にする。またループは降順に回した方が効率が良い場合が多い。

```
Bad:
  for (i = 0; i < (end-beginning)/size; i++) {...}
Better:
  for (i = beginning; i < end; i+= size) {...}
Good:
  for (i = total; i > 0; i--) {...}
```

CPUチューニング（続き）

- イミディエイトモードのチューニング（続き）
  - ***Switch statements:***冗長な**if-else-if** 文より **switch** 文を使うようにする。
  - ***Divitsion:***割り算を使わない。論理シフトを使うか逆数との掛け算にする。
  - ***Subroutine prototyping:*****ANSI C**型のサブルーチンプロトタイプ宣言をして、実行時のパラメータのキャストをなくす。

```
void drawit(float *ptr, int count)
{
 ....
}
```

  - ***Typecasting:***実行時発生するようなキャストは行わない。（ポインタのキャストはコンパイル時に行われるのでパフォーマンスには影響しない。）
    - 実行時に発生するキャスト例

```
val = (float)*ptr;
```

    - コンパイル時に発生するキャスト例

```
int *ptr;
*(float *)ptr = float_val;
float_val = *(float *)ptr;
```

CPUチューニング（続き）

- イミディエイトモードのチューニング（続き）

  ー***Pointers:***もしアプリケーションを極限的にチューニングする必要が出てきた場合、**OpenGL**の各関数のアドレスをポインタ変数にコピーをし、そのポインタを介して**OpenGL**関数を呼び出すこともできる。こうすることによって、**Dynamic Shared Object(DSO)**内の関数を呼び出す際に発生するオーバーヘッドを無くすことが出来る。

```
Ok:
  glBegin(GL_TRIANGLE_STRIP);
    glVertex3fv(v);
    glVertex3fv(v+4);
    glVertex3fv(v+8);
  glEnd();

Better:
  void (*vertex)(const GLfloat* v) = glVertex3fv;
  glBegin(GL_TRIANGLE_STRIP);
    (*vertex)(v);
    (*vertex)(v+4);
    (*vertex)(v+8);
  glEnd();
```

CPUチューニング（続き）

- イミディエイトモードのチューニング（続き）

—***Geometry display choices:***表示するオブジェクトが幾つかあり、それらを切り換えて描画をするような場合は、描画ルーチンの中でどれを描画するかを判定するのではなく、描画ルーチンをそれぞれ専用のものを用意して、呼び出す際に判定をするようにする。

例えばフラットとグーローのシェーディングを切り換える場合

```
Bad:
  draw_object(float *data, int npolys, int smooth) {
    int i;
    for (i = npolys; i > 0; i--) {
      glGegin(GL_POLYGON);
        if (smooth) glColor3fv(data);
        glVertex3fv(data + 4);
                :
      glEnd();
    }
  }
```

この処理が無駄!!

・このような場合は２つのルーチンを用意し、呼び出すときに選択する。

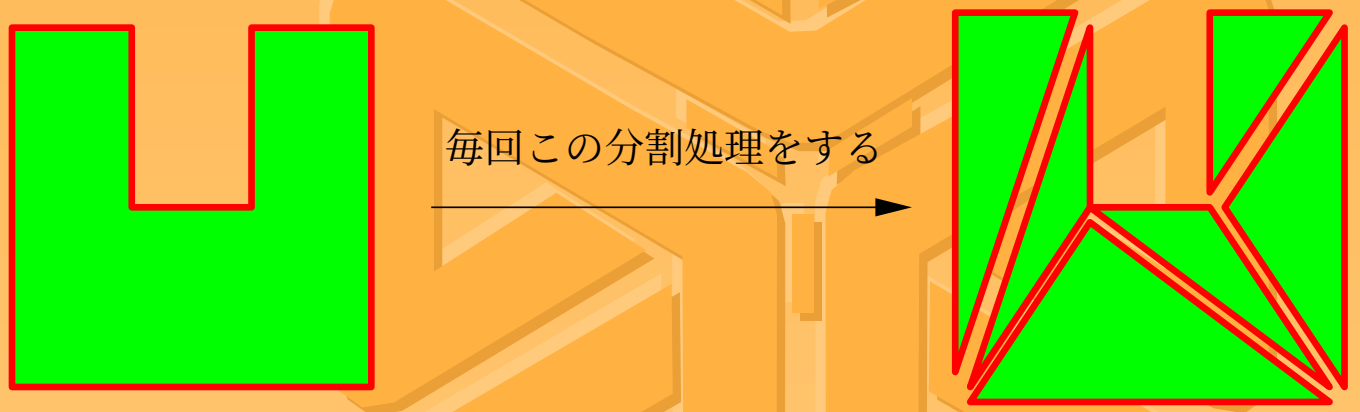
Flat routine

Gouraud routine

CPUチューニング（続き）

・イミディエイトモードのチューニング（続き）

－***Preprocessing data*:**五角形以上のポリゴンを描画するときや凹型のポリゴンなどを描画するときは、あらかじめより単純な形（三角形や四角形）に変換しておいた方が効率が良い。例えば凹型のポリゴンを描画する場合、描画ルーチンが呼ばれる度に三角形分割ルーチンが呼び出されるので余分な時間を消費してしまう。



－**GLUのtessellator**を使用すれば、凹型ポリゴンを三角形に分割したデータを得ることができる。

CPUチューニング（続き）

- イミディエイトモードのチューニング（続き）

—***Preprocessing vertex*:**メッシュプリミティブ（**GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN**）は、描画する際に無駄な計算が発生しないようにあらかじめデータの整理をしておかないとその特徴を活かせない。

以下の例は良く使われている手法であるが、この方法では各頂点を送り出す毎にループの判定処理及び**switch**文の判定処理が必要であり、**triangle strip**の高速性を活かせない。このような頂点毎に何らかの処理を加えているコードは**CPU**ボトルネックを起こしやすい。

```
while(!done) switch(*data++) {
  case BEGINSTRIP:
    glBegin(GL_TRIANGLE_STRIP);
    break;
  case ENDSTRIP:
    glEnd();
    break;
  case EXIT:
    done = 1;
    break;
  default:
    glNormal3fv(dataptr);
    glVertex3fv(dataptr+4); dataptr += 8;
}
```

－上記のプログラムの場合、データを少し整理すれば以下のように書き直せる

```
glBegin(GL_TRIANGLE_STRIP);
for (i = num_verts; i > 0; i--) {
  glNormal3fv(dataptr);
  glVertex3fv(dataptr + 4);
  dataptr += 8;
}
glEnd();
```

－イミディエイトモードの**strip**は、**8,12**頂点を与えたときにピーク性能が発揮されるように設計されている。通常は**glBegin()**～**glEnd()**の間にある頂点は自動的に最適な分割がされる。上記のコードを更に最適化をすると、

```
for (i = N; i > 0; i--) { /* N x 8 三角形毎に描画 */
  glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(dataptr);
    glVertex3fv(dataptr+4);
    glNormal3fv(dataptr+8);
    glVertex3fv(dataptr+12);
    ...       :
  glEnd();
  dataPtr += 80;
}
```

－更に上記のループ内の展開（**unroll**）はマクロを使うことでコンパクトに書くことができる。

```
#define LIT_SMOOTH_VERT(ptr, offset) \
  { glNormal3fv(ptr + offset); \
    glVertex3fv(ptr + offset + 4); }

#define LIT_SMOOTH_MESH_LENGTH_8(dataptr) \
  {\
    LIT_SMOOTH_VERT(dataptr, 0); \
    LIT_SMOOTH_VERT(dataptr, 8); \
    ....              :
    LIT_SMOOTH_VERT(dataptr, 72); \
  }
```

－メッシュデータの展開（unroll）をするループの大きさの目安は最大でも12頂点分くらいである。ループを展開することによってループ処理をするオーバーヘッドは無くなるが、ある一定以上になるとプログラムのコードサイズが大きくなり、インストラクションキャッシュの浪費になってしまう。

CPUチューニング（続き）

- ディスプレイリストモードのチューニング

ー必要のなくなったディスプレイリストは**glDeleteList()**で消去する。

ー同じディスプレイリストのコピーは作らない。
例えば**100**個の大きさの違う球を描くディスプレイリストを作る場合、まず原点に位置している大きさ**1**の球を入れたディスプレイリストを作る。次に目的のディスプレイリストを作り、

1)球のマテリアルを定義する。

2)球のサイズと位置をあわせるためのコマンドを入れる。

3)先ほど作った球のディスプレイリストを**glCallList()**で呼び出す。

4)以上を**100**個の球の分だけ繰り返す。

ーディスプレイリストの階層構造はあまり深くしない。

ーダイナミックに形状が変わるような描画の場合にはイミディエイトモードを使用する。

CPUチューニング（応用）

- グラフィックス部分と計算部分をミックスする

  ―グラフィックス部分と計算部分を分離し、組み合わせることによって効率の良いアプリケーションを組むことが可能である。この組合わせのキーとなる場所は、**glXSwapBuffers(), glClear(),**そして**fill−rate limited**が発生している描画（背景や地面などのように画面上の広い領域を占有するような描画）などの直後である。これらのコマンドを発行した直後はそのコマンドの処理が終わるまでグラフィックス に関する他のコマンドは強制的に待たされることになる。この間にグラフィックスコマンドを発行し続けると、グラフィックスパイプラインのキューが溢れてしまい、結局待たされることになる。
  例えば６０Ｈｚのリフレッシュレートで動いているシステムで**glXSwapBuffers()**を発行した時には、最悪の場合には次の画面の垂直操作までの**16.7msec**を待たされる場合がある。これは例えば毎秒１０フレームの描画を行っているプログラムの場合には、その処理時間の１５**%**を単にバッファの切り換え時間待ちのために消費してしまうことになる。しかしながら、グラフィックスに関係のない他の計算処理は待たされるわけではない。そこで毎フレームごとに必ず実行されなければならない処理のうち、グラフィックスに関係のない処理をここで無駄無く実行することが出来る。

  ―ただし、挿入した計算処理が新たなボトルネックにならないように注意をしなければならない。例えばその新しい処理が大量のデータを扱おうとした場合、キャッシュの中にある描画に必要なデータがメモリにスワップアウトされてしまい、逆に遅くなってしまう場合もありえる。

CPUチューニング（応用）

- **dis**コマンドでアセンブラコードを見る
  - 最も内側にあるループなどはアセンブラコードを見て、無駄なコードが無いかをチェックするのも効果的である。これをするのに**MIPS**のアセンブラのエキスパートになる必要はない。**dis -S** で作られたソースコードには行番号や自分で入れたコメントも表示されるので、それを頼りに無駄なコードがないかをチェックすれば良い。「**mips RISC** アーキテクチャ」などが参考になる。
- 複数プロセッサある場合にはそれらを有効に使う
  - 複数のプロセッサがある場合には、役割分担を決めて**CPU**ごとに違う処理をさせると効果的である。例えば、アプリケーションの実行、データベースの管理、クリッピング処理、グラフィックス描画を別々の**CPU**で行うなど。但し同期をとるために、新たな処理が発生してしまう。

Geometry subsystemチューニング

- より高速な描画モードを使う
  - 可能な限り**glShadeModel(GL_FLAT)**を使う。これによりライティング計算が頂点単位からプリミティブ単位に減り、更にＣＰＵから渡されるデータ量も減らすことが出来る。これは高速なライン描画を行うときには重要である。
- より高速な描画関数を使う
  - 同じ機能を提供するのであれば、より高速な方を使うようにする。例えば、 **glVertex3f()**より**glVertex3fv()**を使うなど。
- より高速なプリミティブを使う
  - **connected primitives**（頂点共有をしているプリミティブ、**GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP**）を出来るだけ使うようにする。使うときには最小でも８頂点を、出来れば１２か１６頂点を渡すようにする。頂点共有プリミティブを使うと、**CPU**からグラフィックスパイプラインに送られるデータ量が減るだけでなく、パイプライン内で行われるポリゴン単位の処理も減すことができる。

Geometry subsystemチューニング（続き）

- より高速なプリミティブを使う（続き）

  ー三角形や四角形などの単純なポリゴンを使う。
  五角形以上のポリゴンまたは凸型でないポリゴンは、一度単純なポリゴンに変換される処理が入るため、若干遅くなる。あらかじめ自分で変換しておくと効率が良い。その際、頂点共有プリミティブに置き換えると更に良い。

  ー**glBegin()/glEnd()**の間では、頂点に直接かかわる処理のみを行う。
  マテリアル、カラーの設定、法線、エッジフラグ、テクスチャコーディネート、サーフェスパラメトリックコーディネート、頂点以外のグラフィックスコマンドは入れない方が良い。他のグラフィックスコマンドを入れると思わぬ結果になってしまったり、またパフォーマンスに重大な影響を与える場合がある。

  ー頂点単位に与えるデータ量を減らす。
  不必要なデータの設定（例えばポリゴン単位の指定だけで十分なカラーの設定を頂点単位で行うなど）はなるべくしないようにする。毎回同じ値でカレントの状態を書き換えるのは非効率的であり、また**CPU**から送るデータ量も増えてしまう。

Geometry subsystemチューニング（続き）

・ライティング処理の最適化

－**OpenGL**は様々なライティング機能を提供しているが、それらの中には描画のパフォーマンスを落してしまう物ある。どのライティング機能がパフォーマンスに影響を与えるかはマシンによって違うので一概には言えないが、可能な限り単純にするのが望ましい。

ライティングをした上でピークパフォーマンスを得るには以下のように設定する。

・平行光源を１つだけ設定する。

・**RGB**モードを使う。

・**glLightModel()**の**GL_LIGHT_MODEL_LOCAL_VIEWER**を**GL_FALSE**にセットする。**(default)**

・**glLightModel()**の**GL_LIGHT_MODEL_TWO_SIDE**を**GL_FALSE**にセットする。**(default)**

・**GL_NORMALIZE**を**Disable**にする。（モデルビューマトリクスをスケーリングする場合などには使えない。）

Geometry subsystemチューニング（続き）

- パフォーマンスの低下があるライティング機能

ー以下はパフォーマンスの低下を招くライティングの機能または使い方である。

・ライトを複数設定した場合。
当然のことながらライトを複数設定した場合には処理時間は増える。

・ローカルライトを使った場合。
ローカルライトは平行光源より処理が重い。

・マテリアルパラメータを頻繁に変えた場合。
マテリアルを頻繁に変える場合は**glColorMaterial()**を使用すると効率が良い。

例えば**ambient**と**diffuse**マテリアルのパラメータを変えるような場合は以下のようにすれば良い。

```
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
...
/* ambientとdiffuseのパラメータをセットする。*/
glColor4f(red, gree, blue, alpha);
glBegin(GL_POLYGON);
...
glEnd();
```

Geometry subsystemチューニング（続き）

- ライティング時には避けた方が良い使い方
  - **glBegin()/glEnd()**間でのライティング関連機能の変更は避けた方が良い。

    可能な限り、**glBegin()/glEnd()**間では**glMaterial()**コールは呼ばないようにする。スペック上は許されているが、パフォーマンスは低下してしまう。
  - **GL_SHININESS**マテリアルパラメータの変更は避けた方が良い。

    GL_SHININESSは新しい値をセットするたびに計算処理を必要とする。
  - ライティングを使用しているときはスケーリングや正規化されていないビューマトリクスを使わない方が良い。

Geometry subsystemチューニング（続き）

- **OpenGL**の高度な機能を使うときは

  －**OpenGL**では様々な特殊機能をすばらしい速度で実行することが可能であるが、それらの機能を使わなかった場合と比較するとやはりある程度のスピードの低下がある。そこで高度な機能を使用しているときの注意点を下記に示す。

  ・必要のないときには特殊機能は**OFF**にしておく。

  ONにしておくと見た目には何の変化をもたらさなくとも、スピードの低下を招く機能がある。例えば**fog**などはたとえ**density**のパラメータが**0.0**でもポリゴンの幾何変換パフォーマンスの低下を招く。

  ・特殊機能のモード切り換えは最小にする。

  あらかじめデータベース側で特殊機能を使うものと使わないものとで分けておいてから一括して描画をすると効率が良い。
  頻繁な**ON/OFF**はパフォーマンスに影響を及ぼす。

Geometry subsystemチューニング（応用）

- チューニングテクニック
  - ―複雑な形状のポリゴンをテクスチャで表現する。

    あらかじめ複雑な形状のポリゴンをテクスチャデータにしておき、それを一枚のポリゴンに張り付けて代用することができる。そのテクスチャポリゴンの必要のない部分は、アルファブレンディングで**glAlphaFunc(GL_NOTEQUAL, 0.0)**と設定することで消すことができる。

  - ―小さなテクスチャをたくさん使用する場合には、一つの大きなテクスチャにまとめてしまう。

    一枚のテクスチャにたくさんのイメージを入れておき、使用する際にテクスチャのコーディネイトを工夫する。こうすることによりテクスチャの切り換え処理を無くすことができる。またテクスチャのロード時間やテクスチャメモリの節約にもなる。

  - ―カリングや裏面消去を他のプロセッサで行う。
  - ―**projection matrix**や**glDepthRange()**の値を変更しない。
  - ―**fog**が**ON**の時には**fog**のパラメータを変更しない。
  - ―平行投影に切り換えるときには**fog**は**OFF**にする。

Raster subsystemチューニング

- ポリゴンの裏面描画を行わない
  - 例えば球を描く場合を考えると、常に半分のポリゴンが裏面であり画面上では隠れて見えないわけである。このような場合には裏面描画を行わない方が効率的である。**Backface/Frontface**の消去は幾何変換処理の後、ピクセル処理の前に行われる。これはこの処理が**Geometry subsytem**で行われることを意味する。よって**Geometry subsystem**側の負荷は大きくなるが、代わりに**Raster subsystem**側の負荷は小さくなる。

- ピクセルモードは効率的なものを選ぶ
  - ピクセルモードによって速度が違うので、必要のないときにはより速度の速いモードにしておいた方が良い。以下に速度の速い順番にピクセルモードを並べたものを示す。

    - **flat-shading**
    - **gouraud-shading**
    - **depth buffering**
    - **alpha blending**
    - **texturing**

    ただし、**depth buffering** は隠れたポリゴンに対する **alpha blending**や**texturing**を減すことがある。

Raster subsystemチューニング（続き）

- **Depth-buffering**の効果的な使い方
  - 例えば背景に大きなポリゴンを描く場合には、**depth buffering**を**OFF**にして描画する。また地面や海面などのように**X-Y**グリッド状になっているものは、奥から手前に向かって描くようにデータを並べて、**glDepthFunc(GL_ALWAYS)**と指定して描くことにより、**depth buffering**の比較処理を無くすことができる。
- ポリゴンサイズとピクセル処理のバランス
  - **Raster subsystem**のパイプラインは通常一辺が１０ピクセル程度のラインやポリゴンの描画に最適化されている。大きなポリゴンを一枚描くときには、小さなポリゴン数枚に分割した方が速い場合がある。この辺りは**Geometry subsystem**と**Raster subsystem**のバランスを考えて決める必要がある。
- バッファのクリア
  - カラーのバッファとZバッファを両方ともクリアする場合には、**glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)**でクリアした方が良い。

アニメーションのチューニング

- アニメーションのなめらかさは**frame rate**に依存する

  －１秒間に描画されるフレーム数が多いほどなめらかな動きが得られる。

  －以下の要素が１フレームの描画に関係する。

  ・画面クリア時間（カラーとＺバッファ）
  ・グラフィックスサブシステムへのデータの転送時間
  ・ライティングやテクスチャなどの処理時間
  ・ビューイング変換処理時間
  ・描画されるピクセル数のフィルにかかる時間
  ・ビデオのリフレッシュ時間（６０Ｈｚの場合は約**16.7msec**）

- チューニングのポイント

  －なめらかな描画を行うにはビデオのリフレッシュ時間内に描画を終わらせるようにする。

  －正確なベンチマークを行うにはシングルバッファモードで行う。

  －**glXSwapBuffers()**の待ち時間を有効に利用する。

ベンチマーク実施時の注意点

- スタンドアローンで行う
  - －マルチユーザモードで良いが、他のユーザがログインしていない方が望ましい。
- 十分な時間をかける
  - －誤差を少なくするために十分な時間をかけたベンチマークをする。
- 動きのない画像で行う
  - －問題点を明確にするため、静的な画像を繰返し描画する。
- 時間計測を行う前後に**glFinish()**を入れる
  - －確実に実行されたのを確認するため。

グラフィックス表示性能概算

- 評価方法

ーカタログスペック等を頼りにアプリケーションがマシンのグラフィックス性能を出しきっているかを大まかに計算することができる。（精度は低いが参考にはなる。）

例えば**Indigo2−Extreme**の大体の性能は**Geometry**が**645K/sec**、**Raster**が**78Mpix/sec**であるので、**200pixel**のポリゴンを**50,000**個描こうとした場合は、

(50,000 polygons) / (645K polygons/sec) = 77.5 msec transform time
(200 * 50,000 pixels) / (78M pixels/sec) = 128.2 msec fill time

よってこの場合には**Raster subsystem**側がボトルネックになる。この場合は**Geometry**側の処理を少し複雑にしても性能が落ちることがない。（例えばライトを１つ増やすなど。）

Tips

- **Triangle Strip**を使用した裏技

ー**TriangleStrip** では一直線上に並んでいるデータは描画されない。
それを利用して離れている物体も一つの**Strip**で表現することができる。

例）　頂点番号