100万個の球を描く10の手法

菊池 巧　脇田 建　佐藤 仁

関係性の可視化にしばしば用いられるグラフ描画技法は情報可視化における基本技術の一つである．われわれは 100 万頂点を有する大規模社会ネットワークの超高次元配置とその対話的操作を目標に開発を行っている．この目標の達成のために，膨大な数の頂点と辺を高効率で描画する技術が不可欠である．本発表では，開発中の可視化システムの基礎技術として多数の球を描画するための 10 の描画技法を紹介するとともに，その利害特質について比較する．具体的には，グラフィックスパイプラインを構成する各種シェーダ，並列計算のための計算シェーダなどを活用し，CPU-GPU 間の通信を削減するための GPU 上のバッファを活用する技術，データ表現などについて議論する．

1　はじめに

本稿の標題を眺めて奇異な印象を持った読者は多いだろう．われわれは，社会ネットワークの対話的な可視化システムに関する研究を実施し，大規模社会ネットワークの構造解析システム [12] [15]，階層的可視化システム Social Cosmo Broser [13] の開発，対話的な高次元可視化システム Social Viewpoint Finder [10] [14] などを開発している．本稿は，社会ネットワーク可視化システムの大規模化を狙って，OpenGL シェーダプログラミング [3] を試みるうちに得られた知見の一部をまとめたものである．OpenGL [8] やシェーダプログラミングについて詳しく解説した良書 [16] [9] は多く，球のような基本的な図形の描画は，ゲームやシミュレーションの実装をしている応用プログラマにとっては目新しいものはないであろう．このように描画研究としての新規性は乏しいが，膨大な数の球の描画に特化して関連する技術をまとめたものは一定の価値があると考え，ここに発表することとした．

Ten different methods of rendering one million spheres
Takumi Kikuchi, 東京工業大学, Tokyo Institute of Technology.
Ken Wakita, Hitoshi Sato, 東京工業大学, JST/CREST, Tokyo Institute of Technology, JST/CREST.

本稿はわれわれが開発している Social Viewpoint Finder の処理能力を飛躍的に高める目的から調査を行ってきた内容の一部である．これまでに開発した Social Viewpoint Finder は，細部が提案した対話的高次元可視化技術 [2] を三次元グラフィックスに拡張したアルゴリズムの有効性を社会ネットワークの可視化に応用を発見して進めてきた．これまでに，三つのプロトタイプをそれぞれ Matlab, Java3D, C++ (WxWidgets と OpenGL) で実装してきた．アルゴリズムの検証を目的として Matlab でファストプロトタイピングしたシステムは，100 頂点程度の小さなネットワークしか処理できなかった．次の Java3D 版で 1000 頂点程度のネットワークの可視化ができるようになった．Java3D 版でさまざまなネットワークを可視化するうちに，社会ネットワークの可視化での有効性が見えてきた．しかし，この程度の規模のネットワークでは既存の静的なグラフ可視化でも処理でき，われわれが提案する対話性の強みを発揮するには十分ではなかった．Java3D 版が十分な性能を発揮できなかった原因は，Java を用いた場合の描画性能が劣っていることにある．C++で再実装した最新版は最大で 5,000 頂点程度の社会ネットワークの可視化に対応できるようになった．このシステムを中規模の社会ネットワークに適用する過程から，大規模ネッ

トワークの可視化で問題となっている巨大な毛玉問題 (*Giant hairball problem*) [1][5][11][7][6] の解決策になりうることがわかってきた．[10][14] 本稿で紹介する試みは，Social Viewpoint Finder の処理性能を 100 倍近く向上することによって，毛玉問題が本質的な意味を持つ大規模社会ネットワークの可視化への応用を見据えたものである．

既存の Social Viewpoint Finder では，いわゆる古典的な OpenGL プログラミングの流儀に沿って実装している．この流儀は，習得が容易ではあるが，現代的なグラフィックプロセッサ (*GPU*) の性能を十分に発揮することができない．Social Viewpoint Finder の性能評価より，性能を律する因子が CPU 側の計算能力ではなく，描画に伴うデータ量，CPU と GPU の間の通信量などにあることが明らかになってきた．これらの問題はシェーダと呼ばれるグラフィックプロセッサの可視化エンジンをプログラムすることによって大幅な改善が期待と予想して，調査を開始した．

本稿は，シェーダを用いた現代的な OpenGL プログラミング技法で基本的なものを多数の球の描画に応用した手法を紹介するものである．実装を行ったコードは GitHub(図 2) で公開しているので，参照していただければ幸いである．

このコードの中で，本稿に対応したモジュールが定義されている．例えば，3.1 説で述べている P1 法のコードは，SGP1.hpp, SGP1.cpp という C++のコードと SGP1.shaders という OpenGL シェーダのコードから構成されている．OpenGL シェーダプログラミングにおいては，アプリケーションごとに複数のシェーダを用い，それらを個別のファイルに保存して管理することが普通である．われわれは複数のシェーダを `==> ... <==`というマークアップで区切って`.shaders` という拡張子を持ったファイルに保存している．

2 現代的な OpenGL プログラミング

OpenGL は高性能グラフィックスのための基盤技術の規格のひとつであり，現在は Khronos Group と称する業界団体が策定している．OpenGL は当初は三角形などの単純な図形を描画するためのさまざまな命令から構成される API であった．OpenGL においては，**固定シェーダ**と呼ばれるグラフィックパイプライン †1 を実現するハードウェアに対して，パラメタを設定した上で，図形のデータを送付することで描画処理がなされる．

2.1 固定シェーダとプログラム可能なシェーダ

2004 年に発表された OpenGL 2.0 においては，シェーダの働きの一部をソフトウェアで記述するための言語として *OpenGL* **シェーディング言語** (以後, *GLSL*) が利用できるようになった．固定シェーダの利用に際しては，描画に必要なデータの準備は主に CPU 側で事前に計算する必要があった．一方，GLSL を利用することで GPU の膨大な計算能力を活かせるようになった．さらに重要なことに，描画に必要なデータを GPU 側で管理し，限定的な CPU-GPU 間の通信を避けることで，ともすると描画処理のボトルネックとなる CPU-GPU 間の通信を大幅に削減できるようになった．

OpenGL は数年おきにメジャーバージョンを更新し，最新の規格 OpenGL 4.x においては**頂点シェーダ** (*Vertex shader*)，**テッセレーションシェーダ** (*Tessallation shader*)，**形状シェーダ** (*Geometry shader*)，**画素シェーダ** (*Fragment shader*) といったグラフィックパイプラインの主立った要素をプログラムできるようになっている．さらに，2012 年に発表された OpenGL4.3 においてグラフィックパイプラインと独立に働く**計算シェーダ** (*Compute shader*) によって汎用 GPU 計算が記述できるようになった．

OpenGL API と GLSL の機能は低レベルかつ煩瑣であるため，本節では重要な概念のみの説明に留める．詳細については，仕様書や参考書を参照されたい [16][9][8][3].

2.2 頂点配列と頂点バッファ

われわれの身の周りには 3D グラフィックスが満ち溢れている．その高度に複雑かつ精細な描写と比較して，OpenGL が提供する機能は恐しく原始的であ

†1 三角形のようなグラフィックプリミティブの配置と座標変換を経て，ピクセルの色を計算する一連の処理

る．たとえば，本稿の主題である球について言えば，球を描くための命令などというものは存在しない．OpenGL が提供するのは，線，折れ線，三角形を描画するための機能に限定すると思ってよい．OpenGL を利用して図形を描画するためには，その図形を十分に細かい三角形に分解し，この三角形の集合について必要な情報（形状，色，模様，法線など）を GPU に送信したのちに，描画命令を発行する．

GPU への図形情報の送信に関わる重要な二つの概念が**頂点配列** (*Vertex array*) と**頂点バッファ** (*Vertex buffer*) である．たとえば，三次元空間における三点 $\{(1,0,0),(0,1,0),(0,0,1)\}$ から構成される三角形の各点をそれぞれ赤，緑，青で着色したものを考えよう．このような情報を JSON 風に表現するならば，以下のような頂点属性の配列を思いつくかもしれない．

```
triangle = [
  { position: [ 1, 0, 0 ],
    color: [ 255, 0, 0 ] },
  { position: [ 0, 1, 0 ],
    color: [ 0, 255, 0 ] },
  { position: [ 0, 0, 1 ],
    color: [ 0, 0, 255 ] } ];
```

OpenGL においては，GPU のメモリアクセス特性にしたがって，このような構造体の配列ではなく，配列の構造体を用いることが一般的である．すなわち，３つの頂点について座標の配列と色の配列を用意した上で，これらふたつの配列を一括して GPU に送付する．ここにおいて，それぞれの配列を抽象化した構造を**頂点バッファオブジェクト** (*Vertex buffer object*，略して *VBO*) と呼ぶ．各 VBO は頂点群の属性を表すと考えるとよい．これら VBO 群をまとめあげたものを抽象化した構造が**頂点配列オブジェクト** (*Vertex array object*，略して *VAO*) である．

```
VAO = {
  positions:
    [[1, 0, 0], [0, 1, 0], [0, 0, 1]],
  colors:
    [[255, 0, 0], [0, 255, 0],
     [0, 0, 255]] };
```

前述のように VBO は頂点群の属性を表す．VBO は，すでに例示した座標と色のほかに，テキスチャ座標 †2，三角形の法線（陰影，反射，屈折などの光学効果の計算に用いる）などの属性を表現するのに頻繁に用いられる．

2.3 グラフィックパイプライン

OpenGL では，VAO を介して与えられる頂点の属性をもとに，グラフィックパイプラインと呼ばれる幾段かの処理を経て，画像が生成される．基本的な構成では，頂点シェーダと画素シェーダのみが用いられる．まず，表現したい図形を三角形に分解し，各頂点についてさまざまな属性を VAO に保存したものを GPU に渡す．GPU は VAO から取り出した各頂点の属性をもとに，その頂点の三次元座標に射影を施すことによって，頂点に対応する画面上の場所を計算する．つぎに，画面上の三角形が覆う領域に含まれるすべての画素（ピクセル）について，画素シェーダを呼び出しその画素の色を定める．画素ごとに色を計算することによってきれいな陰影表現を実現することができるが，三角形の頂点ごとに色を計算し，その色を補間するだけでもそれなりの陰影を施すこともできる．色の補間はハードウェアが自動的に計算するために実行性能がよい反面，凹凸のある図形の場合不自然な陰影を作ることもある．

前述の基本的なシェーダの構成に対して，さらにプログラム可能シェーダの機能を追加できる．テッセレーションシェーダを用いると，与えられた三角形を小さな三角形に細分化することができる．テッセレーションシェーダのひとつの目的は，頂点シェーダに対する陰影づけだけでは不自然な陰影がある場合に，細分化することでその不自然さを修正できる点にある．そもそも，表示したい図形を十分に小さな三角形に分解しておけばよいのだが，その計算を並列性の乏しい CPU にさせるのは非効率であるばかりでなく，頂点数が増加するにつれて CPU-GPU 間の通信時間が問題となる．一方で，画素シェーダを用いて画素ごとに陰影を計算するのはいくら GPU が高並列とはいっても無駄が大きい場合が多い．このような状況を解決するためにシェーダ内で三角形を細分化することは有

†2 三角形に模様を貼り付けるときに各点に別途与える画像内のどの位置を対応させるかという情報

効である．テッセレーションシェーダのもうひとつの重要性は，細分化の度合いを図形の見た目の大きさに応じて調整できる点にある．すなわち，視点の近くにある物体は細かく分解してきれいに描写することが求められる．反対に視点から遠い大部分の物体を高精細に描画しても無駄というものである．むしろ，雑に描くことで処理性能を改善することができる．いわゆる**段階的詳細化** (*Level of detail*，あるいは *LoD*) 技法である．

形状シェーダは，一風変っている．他のシェーダは前段のシェーダから与えられた構造に変換を施しているに過ぎないが，形状シェーダは与えられた図形を別の形状に変形する能力を有する．たとえば，点を三角形に変形したり，逆に与えられた多角形のうち，いくつかの点を削除することができる．この機能は，テッセレーションシェーダをさらに機能強化したように見えるが，同等の処理をする場合の性能はテッセレーションシェーダの方が優れている．形状シェーダは与えられた形状を消滅させることもできる．このため，LoD 処理での，形状シェーダによって視点からの見かけの大きさがあまりに小さな図形の描画を抑制することもできる．

2.4 計算シェーダ

計算シェーダは CUDA や OpenCL に類似した汎用 GPU 計算に該当する機能を提供する．他のシェーダと異なり，計算シェーダはグラフィックパイプラインに属さない．計算シェーダを用いた描画処理においては，アプリケーションはまず計算シェーダを起動する．計算シェーダは計算結果を**シェーダ共有バッファ** (*Shader shared buffer*，あるいは *SSB*) と呼ばれる GPU 内の共有メモリに残す．つぎに，アプリケーションは OpenGL の描画命令を用いてグラフィックパイプラインを起動する．ここで，SSB を参照することによって，計算シェーダの計算結果を利用することができる．

SSB は計算シェーダとグラフィックパイプラインを構成する他のシェーダの間で共有するだけでなく，CPU 上で動作するアプリケーション側から参照・変更することもできる．

3 プログラム可能シェーダの利用

グラフィックスアプリケーションをシェーダを用いて実装する場合，そのアプリケーションは CPU 側で実行するアプリケーションのコードと GLSL で記述する各種シェーダのコードから構成する．アプリケーションのコードはおおまかに，各種の設定を施し，形状情報を VAO に保存するための初期化部と，描画部から構成される．本稿で紹介する実装においては，アプリケーションコードは以下のシグナチャを持つ SG 仮想クラスを導出して作成する．

```
class SG: public Volume {
  protected:
    Json *A;
    Program *program;
    int size;
    GLuint stacks;
    int nSpheres;
    GLfloat r;
    mat4 V, *P;

  public:
    SG(Json *, Program *, mat4 *);
    virtual void render(double);
    virtual void render();
};
```

ここで，コンストラクタが初期化を，render が描画を担う．コンストラクタに渡される三つの引数は順番に，JSON 形式のアプリケーションの環境設定 (A)，GLSL のシェーダ群を表現する抽象データ ($program$)，そして三次元グラフィックスにおける視野を表す射影行列 (P) である．描画を担う render メソッドはアプリケーションが起動してからの経過時間を受け取る版と引数を取らない版が多重化されている．本稿で紹介する例はいずれも経過時間を参照しない．

球の形状表現においては，緯度方向を $stacks$ 個に，経度方向を $slices$ 個に分割してたスイカの輪切りのようなものを用いている．球の両極部はそれぞれ $slices$ 個の三角形の分割となり，他の領域は合計 $((stacks-2)\times slices)$ 個の矩形的領域に分割される．さらに，各矩形領域を対角線で二分する．以上より，全体で $(2\times slices + 2\times(stacks-2)\times slices) = (2\times(stacks-1)\times slices)$ 個の三角形領域に分割することとなる．

3.1 個別の球の描画命令を発行する方法 (P1)

ひとつの球を表現するには，前述の三角形分割について，そこに出現する各々の頂点に関する情報，およびそれらの頂点の組み合わせによって，分割された個々の三角形の形状を表せばよい．頂点数は両極，および各経線ごとに $(stacks-1)$ なので，全部で $(2+(stacks-1)\times slices)$ 個である．それぞれの頂点ごとに 3 次元の座標と 3 次元の法線ベクトルを用いる．三角形の形状の表現については，各三角形ごとにその三角形の 3 頂点に該当する 3 つの頂点番号を用いる．この三角形群の形状を表現するデータは VAO とは独立した VBO を用意する．

複数の球を描画する場合は，それぞれの球について中心座標と半径から，形状を計算してこれらのデータを構成することもできる．しかし，プログラム例ではその代わりに原点に配置した単位球を用意し，それを拡大行列と並行移動行列で変形する方法をとっている．すなわち，移動を表すベクトルを T，拡大率を S とした場合，この変形は以下のモデルビュー変換 (M, MV, MVP) によって表現している．このモデルビュー変換は，個々の球について個別に施すことになる．

```
const ID = mat4(1);
for all (x, y, z) {
  vec3 T = vec3(x, y, z);
  mat4 M =
    translate(ID, T) *
    scale(ID, vec3(r));
  mat4 MV = V * M;
  program->setUniform("MV", MV);
  program->setUniform("MVP", (*P) * MV);
  glDrawElements(GL_TRIANGLES, ...);
}
```

このコードのなかで，setUniform は，シェーダに対して定数値を設定する処理である．glDrawElements を用いて同一の原点に配置された単位円を描画しているが，シェーダのなかで相異なる (x, y, z) についてモデルビュー変換を施すことによって，画面には相異なる場所に配置された複数の球が配置される．

さて，この実装の性能を左右する要素について考察しよう．この実装の主要な要素は，(A) CPU で実行するアプリケーションコードにおける計算，(B) CPU から GPU に送信する形状情報の通信，そして，(C) GPU におけるシェーダの実行である．

各球に対するモデル変換行列の計算は個別には，四次元行列の積をいくつか計算するだけの簡単なものだが，この計算はすべての球に対して実行しなくてはならない．CPU 並列処理能力は限定され，さらに OpenGL は CPU のマルチスレッドに対応していないために，球の数が多い場合にはこの処理も負荷が高くなる．

CPU から GPU に受け渡されるデータ量はどれだけであろうか．送信されるデータには，ふたつのモデル変換行列と，球の形状を表す情報の二種がある．前者のデータ量は $(4\times4\times4)\times2=128$ バイトと小さい．一方，球の形状を表す情報は，球の本質が球の中心座標と半径に過ぎないことに対すると，思いの外大きい．VAO に含まれる座標と法線ベクトルに対応したふたつの VBO はともに 3 次元の **float** 型のベクトルである．つまり，このベクトルの大きさはそれぞれ 12 バイトということになり，頂点あたり 24 バイトの大きさになる．$stacks = slices = 24$ の場合，球を構成する頂点の数は $2+(24-1)\times24=554$ となる．データ量としては，$12\times554=6648$ バイトである．二種のデータを併せると約 6.8KB である．たとえば 100 万個の球を 60fps で描画しようと思っても，1 フレームの描画ごとの通信量は約 6.8GB であり 8Gbps という PCI Express x16 バスの性能から考えて，とても描画できるものではない．

膨大なデータ転送量に加えて，多数の OpenGL 命令の発行が求められる点は本方式のもうひとつの問題である．上述のコードにおいて，各球ごとにふたつの setUniform 命令とひとつの glDrawElements 命令を発行する．これらの命令実行において CPU と GPU のあいだで同期処理がなされ，その待機時間が無視できない．

3.2 雛形を転写して球をまとめて描画する方法 (P2)

個別の球の描画命令を発行する手法 (P1) の非効率性として以下の三点があることを述べた．(A) 描画に際して CPU で実行するアプリケーションコードの

負担，(B) CPU から GPU に送信するデータ量，(C) アプリケーションコードから発行する OpenGL 命令の数.

このうち (A) と (B) の問題は，OpenGL の描画命令が三角形程度の単純な図形にしか対応していないことに起因する．すなわち，球を多数の三角形に分解せざるを得ず，CPU における分解の手数と，多数の三角形の情報を GPU に送付するための手数がそれぞれ (A) と (B) に該当する．ここで紹介する手法では，アプリケーションコードでは球を三角形には分解せず，球の使用，つまり中心の座標と半径の情報を GPU に送付することでこれらの手数を除去する．

(C) の問題については，glDrawElementsInstanced 命令を用いて，原点に配置された単位球を複写することとする．glDrawElementsInstanced 命令の使用にあたっては，P1 手法と同様に球を三角形に分解した形状情報を VAO に収める．さらに，この VAO とは独立な VBO に描画する球群の中心位置の情報を収める．

このまま単純に描画すると，複数の単位球を原点に重ねて描画してしまうが，頂点シェーダで VAO の情報を参照して，雛形に適宜モデル変換を施すことで意図した位置に意図した大きさの球を表示することができる．

```
layout (location = 0) in
  vec3 position_vs;
layout (location = 5) in
  vec3 spherePosition_vs;
uniform mat4 MVP;
out vec3 LightIntensity;

void main() {
  gl_Position = MVP *
    vec4(spherePosition_vs + position_vs,
         1);
  LightIntensity = ...;
}
```

このコードにおいて，*position_vs* と *spherePosition_vs* はそれぞれ球の中心の座標と，雛形となる球の頂点の座標を表す.

この実装方式は方式 P1 と比較して圧倒的に効率的である．ひとつのフレームの書き換えのときに，(A) アプリケーションコードは実質的に glDrawElementsInstanced 命令を一回発行するだけである．(B) CPU から GPU に送信するデータ量は雛形 (すでに述べたように 6.8KB) と球の中心座標のみである．球の数が多い場合には，これは実質，球ごとに中心の座標 12 バイトを送るだけですむ．P1 方式では球ごとに 6.8KB を要したことに比較して，約 1/567 のデータ量ですむ．100 万個の球を 60fps で描画する場合の転送データ量は 5.8Gbps となり，PCI Express x16 バスでかろうじて転送できる転送量に収まる．(C) の命令発行数に関しては glDrawElementsInstanced 命令ひとつで多数の球の描画を担えるため，問題とならない．

3.3 テッセレーションを用いる方法 (P3)

テッセレーションは与えられた基本図形を細分化する処理である．ここで紹介する方式 P3 では，以下のようにアプリケーションコードからは球群の中心座標に相当する点群を描画する命令を発行する．

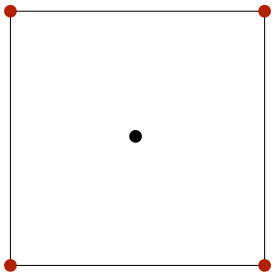
```
glDrawArrays(GL_PATCHES, 0, 3 * N);
```

ここで，GL_PATCHES はテッセレーションを利用するための基本形状であり，N は球の数，$3*N$ は各基本形状が 3 つの要素，つまり，中心の位置に相当する (x, y, z) から構成されることを意味している．

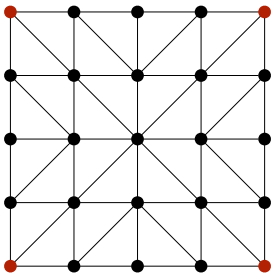
P3 方式のグラフィックパイプラインは，頂点シェーダ，テッセレーションシェーダ，そして画素シェーダから構成される．頂点シェーダは VAO から与えられる球の中心座標をテッセレーションシェーダに伝えるのみである．

テッセレーションシェーダには，点の座標 (x, y, z) が与えられるが，まず，この点を中心として，視点に対して垂直に配置された単位正方形と見做し (図 1(a))，それを三角形に細分化する (図 1(b))．ここで同心円上に配置された頂点のそれぞれについて中心からの距離 (r) について，高さを $\sqrt{1-r^2}$ を与えることによって，円盤を細分化したものが半球を細分化したものに変形することができる．

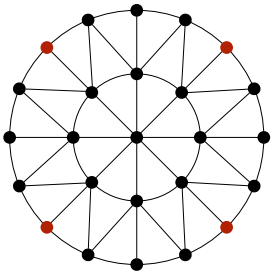
あとは，これらの頂点群が構成する三角形群を通常どおりに描画すれば半球を描くことができる．もともと，正方形を視点に垂直な面に配置したので，視点か



(a) VAO から取得した点は，仮想的にそれを囲む単位正方形として扱う



(b) 単位正方形にテッセレーション処理を施し細分化する



(c) 単位正方形を膨らませて単位円に変形する

図 1: 半球を描くためのテッセレーション処理

らは見えない半球は描く必要がない．

この手法のアプリケーションコードは球の中心座標を VAO に与えるための単純な処理である．また，テッセレーションシェーダがハードウェアで実施する細分化処理によって，手法 P1, P2 でやってきたような細分化処理の記述からも開放されるためシェーダ群の記述も比較的単純である．

このようにこの手法はエレガントに記述でき，P1 手法よりも遥かに効率的ではあるが，雛形を複写するだけの P2 手法に比べると性能はやや劣る．

3.4 点スプライトを用いる方法 (P4)

われわれのようにグラフの描画のために，OpenGL の点の描画機能を試して，多くの人は点ではなく四角形が描画されてがっかりすることだろう．根気のある人も OpenGL のマニュアルを調べて，点を円盤として描画するためのオプションが存在しないことにがっかりする．

OpenGL の点スプライトは，主に粒子系の描画を目的として与えられたシェーダの機能である．この機能を用いるときは，点の座標とその大きさを画面上のピクセル数として与え，点の周辺の正方形領域を画素シェーダで描画する．特に画素シェーダは **discard** という画素を出力しない文を提供しているため，正方形領域のうち中心からの距離が一定以上離れた画素を **discard** することで，画面上に円を描くことができる．厳密には，三次元投影した球の形状は円ではないが，球が視点から十分に離れているときはその差は無視できる．

```
void main() {
  vec2 N = vec2(gl_PointCoord * 2 - 1);
  if (1.0 - dot(N.xy, N.xy) < 0)
    discard;

  FragColor = ...;
}
```

画素シェーダが無視しない円盤内の画素については，対応する球面上の点を想定して，その法線と周辺光などより陰影を計算することによって擬似的に球面らしい模様を表現することができる．

点スプライトを用いる方法は，P2 方式や P3 方式と同様にアプリケーションコードは球の中心座標のみを与える．一方，シェーダは擬似的に円盤を描き，そのために球は円盤を三角形に分割することすら行っていない．このためシェーダの処理も極めて効率的であり，描画速度において雛形を用いる P2 方式すら凌駕する．

3.5 その外の方法

ここまでの手法については，GitHub で参照実装を公開している．以下では，これをさらに発展させたいくつかの手法を紹介する．

3.5.1 SSB の利用

ここまでに紹介した方法はいずれも VAO に球の座標を与えて描画する．シェーダに球の座標を与え

るためには，OpenGL 4.4 で追加されたシェーダ共有バッファ(SSB) を利用することもできる．従来の OpenGL プログラミングにおいては，シェーダのデータを CPU 側から参照する唯一の手段はフレームバッファであったが，OpenGL 4.4 からは SSB を用いることでシェーダ間だけでなく，シェーダとアプリケーションコードの間でもデータを共有できるようになった．このため，いったん，球の座標や球の形状情報のような大量な定数値を SSB に保存し，シェーダでそこを参照させることで，その後の描画処理での CPU から GPU への大量のデータ転送を避けることができるようになった．

3.5.2 計算シェーダの利用

前述の SSB の方法を採用したときは静的なシーンの描画は効率的である．しかし，動的なシーン，すなわち時々刻々と球が移動するようなアプリケーションの場合は，移動のたびに SSB を更新しなくてはならないため，P1〜P4 と同様の問題を抱えることになる．このような場合，OpenGL 4.3 で採用された計算シェーダを利用することができる．計算シェーダは汎用の並列計算言語であり，GPGPU の簡易版と見做すことができる．ほかのシェーダと同様に計算シェーダも SSB を共有することができるため，球の移動を計算シェーダで計算し，その結果を SSB を用いて共有することで，球の座標データへのアクセスを GPU にまとめることができる．なお，SSB は GPU 内の共有記憶として実装されているため，シェーダからのアクセスは比較的効率的である．

3.5.3 テッセレーションシェーダで動的 LoD を用いる方法

テッセレーションシェーダでは，基本形状を細分化することを述べたが，分割数はいくつかのパラメタを用いて制御することができる．描画対象の図形の見かけ上の大きさに応じて分割数を制御する方式は**動的な段階的詳細化技法** (*Dynamic LoD*) として知られる重要な技術である．画面を埋める膨大な数のオブジェクトを効率的に描画する目的でよく用いられる．[4]

3.5.4 テクスチャマッピングする方法

点スプライトは極めて効率のよい描画手法だが，陰影の計算のために画素シェーダを酷使する点でさらに改善の余地がある．図形を小三角形に細分化する方式の場合には，小三角形の頂点の陰影のみを計算し，それを線形補間することで近似的に陰影表現を効率的に計算できる．点スプライト方式において，同様のことをするには事前に球の陰影を計算して得られた画像を SSB に保存し，それを点スプライトに貼り付けることが考えられる．光源が無限遠にある場合は，これで十分だが，そうでない場合には，視野内のいくつかの場所の球のみについて陰影を計算し，それを補間することで概ね良好な陰影が得られる．

4 クリックされた球を識別する方法

われわれが開発している Social Viewpoint Finder は社会ネットワークを可視化するだけでなく，ポイント＆ドラッグ方式でグラフを直接操作できる機能を提供する．このためにはユーザが球をクリックしたときに，たくさんある球のいずれがクリックされたのかを判定する機能が必要となる．二次元の画面でのクリックは，われわれが日常において，遠くの物体を指差す行動に相当する．このとき，われわれの指は対象物に接触しておらず，方角を指し示している．周囲の人間は指先の延長線との交差をもって，指示された物体の認知を試みる．

三次元グラフィック処理においては，ディスプレイ上でクリックされた点の方角の延長線と交差する，もっとも手前の物体を探せばよい．ただし，この方法は，本来 OpenGL が自動的に実施している各種の空間変換を時前で計算することとなり，間違いが介在する可能性が高く，また，どこかシステムの構成に冗長性があるようで気持ちが悪い．以下では，より洗練された二つの判定方法を紹介する．

4.1 球ごとに色分けする方法 (I1)

この方法では，各球を相異なる色で着色するやや乱暴な方法である．OpenGL は，標準で 8bit RGB をサポートしているため，24bit，すなわち 400 万個の物体を塗り分けることができる．この方法を用いるときは，クリック後の 1 フレームだけを用いるため，十分なフレームレートが出ているときは，このおかしなフレームは目に止まらないだろう．仮にこの奇妙な

フレームが目に止まったとしたら，陰影処理も行わないため，平面的な円盤として描かれたものが見えるはずだ．

アプリケーションコードはクリックされた画面上の座標を把握しているので，この奇妙なフレームを得られたら，該当する画素の色を検査することでクリックされた球を同定することができる．

フレームレートが劣化している場合，あるいは完全主義者には，この方式は好かれないかもしれない．そのような場合は，OpenGL の描画対象として一時的にオフスクリーンな画像を指定することで，ユーザに色分けされた円盤を盗み見られる可能性を避けることができる．

4.2 フラグメントシェーダで判定する方法 (I2)

OpenGL 4.4 に対応したグラフィックカードを持っているならば，SSB を用いてもっと洗練された方法でクリックされた図形を同定できる．まず，アプリケーションコードは SSB にクリックされた画面上の座標を保存する．そして画素シェーダでは，当該画素の座標が SSB に指定された座標と一致するか否かを検査し，合致する場合には球の番号を SSB に保存すればよい．ただし，見掛け上，複数の図形が同じ個所に重なりあって見える場合を考慮して，画素の z-座標が大きいものを優先して保存する必要がある．以下が画素シェーダのコード断片である．

```
layout (std430, binding = 0) buffer SSB {
  int   pick_oid;
  float pick_z;
};

void main() {
  FragColor = /* 点スプライトの計算 */;

  if (gl_FragCoord.z < pick_z &&
      distance(gl_FragCoord.xy,
               clickedPosition) < 1) {
    pick_oid = vertexID_fs;
    pick_z = gl_FragCoord.z;
  }
}
```

5 まとめ

ここまで，膨大な数の球を描画する技法ならびに，その表示システムに直接操作をするための基礎的な技術となるクリックされた球の同定方法を中心に議論を展開してきた．本来ならば，ベンチマーク結果とともに紹介すべきだが，目下，詳細な調査を行っている途中であり，正確なデータを公表することができない．感覚的な結果となるが，NVIDIA GTX-860M という GPU を搭載した Dell Alienware 13 というゲーム用ノートパソコンで実行した印象では，**P1** ≪ **P3** < **P2** < **P4** という結果である．実用的なフレームレートととして，10fps 以上の描画が可能な限界は，**P1** 方式では 3 万個程度まで，**P3**, **P4** は数十万個まで，**P4** では百万個程度となる．実験では，ひとつの光源を用いた ADS シェーディングを実施している．今後は，フレームレートだけでなく，アプリケーションコードとシェーダのオーバーヘッド，PCI Express バス上の通信量，CPU の同期オーバーヘッドなどについて調査をしたい．

今回の調査を通して，球の描画以外の面についても，大規模社会ネットワーク向けの対話的可視化システムの実装に OpenGL シェーダプログラミングを利用することの利点が見えてきた．Social Viewpoint Finder は社会グラフの三次元閲覧機能のほかに，ネットワーク指数に応じたフィルタ機能や，高次元回転機能を備えている．フィルタ機能は，OpenGL のインデックス化描画機能を用いること簡易かつ高効率に実装することができる．高次元回転機能は計算シェーダを用いて用意に実装することができる．これまでに得た感触から，これまではほぼすべてを CPU 側で実装してきた機能の大部分をシェーダで実装することができること，実装が CPU 側での実装よりも容易であること，そして，最も重要なことに圧倒的に効率を改善できることなどの知見を得た．

謝辞

本研究の一部は科学技術振興機構戦略的創造研究推進事業 (JST, CREST) の支援を受けています．過去半年間，始めてシェーダプログラミングを始めてからウェブ上の多くのコンテンツに支えられてきまし

プログラム例 `https://github.com/wakita/gldojo/tree/feature/jssst2015-paper/spheres`
本稿更新版 `https://bitbucket.org/kwakita/jssst-2015/src/paper-*.pdf`

図 2: 本稿とプログラム例のリポジトリ

た．参考にした情報の一部は，リポジトリ（図 2）の README に記載しました．

参 考 文 献

[1] Auber, D. et al.: Multiscale Visualization of Small World Networks, *IEEE Symposium on Information Visualization, 2003 (INFOVIS 2003)*, Seattle, 2003, pp. 75–81.

[2] Hosobe, H.: A high-dimensional approach to interactive graph visualization, *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, New York, NY, USA, ACM, 2004, pp. 1253–1257.

[3] Kessenich, J., Baldwin, D., and Rost, R.: *The OpenGL Shading Language, Language Version 4.40, Document Revision: 9*, The Khronos Group Inc., June 2014.

[4] Luebke, D., Reddy, M., Cohen, J. D., Varshney, A., Watson, B., and Huebner, R.: *Level of Detail for 3D Graphics*, Series in Computer Graphics, Morgan Kaufmann, July 2002.

[5] Melanon, G. and Sallaberry, A.: Edge metrics for visual graph analytics: A comparative study, *Information Visualisation*, IEEE, 2008, pp. 610–615.

[6] Nocaj, A., Ortmann, M., and Brandes, U.: Untangling Hairballs, *Graph Drawing*, Springer, 2014, pp. 101–112.

[7] Satuluri, V., Parthasarathy, S., and Ruan, Y.: Local graph sparsification for scalable clustering, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ACM, 2011, pp. 721–732.

[8] Segal, M. and Akeley, K.: *OpenGL Graphics System: A Specification*, The Khronos Group Inc., version 4.5, core profile edition, May 2015.

[9] Sellers, G., Richard S. Wright, J., and Haemel, N.: *OpenGL SuperBible: Comprehensive Tutorial and Reference*, Addison-Wesley Professional, 7th edition edition, July 2015.

[10] Takami, M., Hosobe, H., and Wakita, K.: A Projection-Based Method for Interactive Visual Exploration of Complex Graphs in A Three-Dimensional Space, Research Reports on Mathematical and Computing Sciences C-278, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, February 2012.

[11] von Landesberger, T. et al.: Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges, *Computer Graphics Forum*, Vol. 30, No. 6(2011), pp. 1719–1749.

[12] Wakita, K. and Tsurumi, T.: Finding community structure in a mega-scale social networking service, *proc. WWW/Internet 2007*, Vila Real, Portugal, October 2007, pp. 153–162.

[13] WAKITA, K., Hosobe, H., Takami, M., and MURATA, T.: Taming Interactive Visualization of A Large-Scale Real-Life Social Graph of a Million Nodes, *Taming Interactive Visualization of A Large-Scale Real-Life Social Graph of a Million Nodes*, (2012).

[14] Wakita, K., Takami, M., and Hosobe, H.: Interactive high-dimensional visualization of social graphs, *Visualization Symposium (PacificVis), 2015 IEEE Pacific*, IEEE, 2015, pp. 303–310.

[15] Wakita, K. and Tsurumi, T.: Finding Community Structure in Mega-Scale Social Networks: [Extended Abstract], *Proceedings of the 16th International Conference on World Wide Web*, New York, NY, USA, ACM, 2007, pp. 1275–1276.

[16] Wolff, D.: *OpenGL 4 Shading Language Cookbook*, Packt Publishing, second edition edition, December 2013.