F-028

GPU を用いた見通し判定の高精度化方法について

Accurate and Fast LoS Detection Method by Using GPU

倉本　健介†
Kensuke Kuramoto

1. 緒論

近年、シミュレーションシステムは非常な発展を見せ、防衛分野においても欠かすことのできない技術となっている。シミュレーションシステムが対象とする環境には様々なものがあるが、市街地のような複雑な環境を表現する場合、ユニット間での見通し判定（Line of Sight：LoS）は非常に重要となってくる。そのため、多くの防衛用シミュレーションシステムが LoS の機能を有している。

一方で、LoS は負荷の大きい処理であり、シミュレーションによっては計算負荷のほとんどを LoS が占める場合もある。そのため、LoS 計算の高速化が期待される。Salomon らは GPU を利用した 2 点間の高速な LoS 判定が可能であることを述べた[1]。

現在、ほとんどのシミュレーションシステムにおける LoS 判定は、点と点との間で、見えるか見えないかの 2 値である。登場するユニットを質点で扱うような場合は、これで十分であると言える。

ところが、実際の環境においては、目標となる物体が地形や障害物によって部分的に隠される状況が多く発生する。このような状況をシミュレーション中で表現するためには、物体を大きさや形状を持つユニットとしてモデル化するする必要がある。その場合、見えるか見えないかだけではなく、「どの程度見えているのか」という情報が発生し、それを処理する必要が生じる (図 1)。

一つの方法としては、大きさをもった目標をメッシュ状に細分化し、それぞれの点に対して LoS を繰り返すことが考えられるが、これは近年の高速な計算機にとっても負荷の大きな処理となる。

本研究では、GPU を利用することにより、高精度な LoS 判定を高速かつ汎用的な方法により取得できる方法を提案する。実装は多くの GPU がサポートしている OpenGL2.1 の範囲で行い、実装のためのプログラムには JavaSDK6 を利用し、JOGL2.0 を経由して OpenGL API を呼び出した。

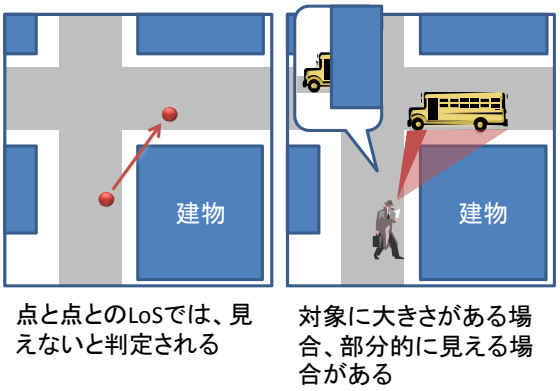

図 1　大きさを持った物体に対する LoS

2. シミュレーションにおける LoS の従来手法

2.1 ステップ法による 2 点間 LoS

防衛用シミュレーションシステムの多くは、基本的な環境情報として標高メッシュによる地形データを保持している。これは、地形をある大きさの 2 次元メッシュに分解し、それぞれのメッシュに標高情報を格納したものとなる。標高メッシュによる表現は、地形情報をポリゴンで描画する場合に比べて簡易な方法であり、トンネル状の地形を表せないなど、表現力において制限がある。その一方、標高情報の獲得が高速となる利点がある。

シミュレーションで LoS を判定するために一般的に用いられている方法は、図 2のようにステップごとに標高をサンプリングするものである（以後ステップ法と呼ぶ）。

ステップ法は図 3に示すようにアルゴリズムが簡便であり、地形データがメッシュ構造であれば計算量も少ない。一方で、標高情報を離散的に取得するため、精度がステップ長に依存し、地形の複雑さに対してステップ長が大きい場合には LoS 判定を誤る可能性がある。また、ステップ法は基本的に点と点との間での LoS を判定するものである。

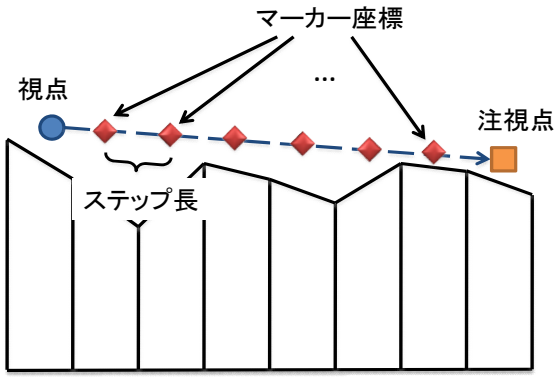

図 2　ステップ法による LoS 判定

† 防衛省技術研究本部

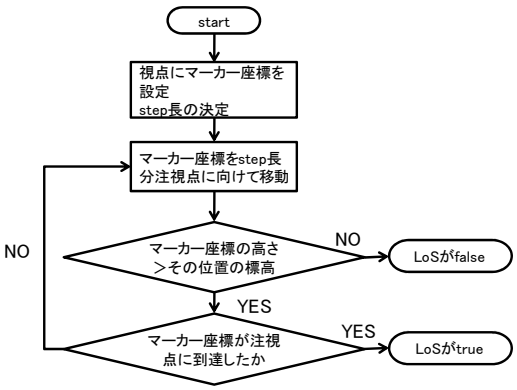

図 3　ステップ法のフローチャート

2.2　Salomon らの 2 点間 LoS

Salomon らによる手法は、GPU のレンダリング処理を利用することで 2 点間の LoS を高速化するアルゴリズムである。

ポリゴンで構成された地形情報を標高の 2 次元メッシュに変換し、それを深度情報として VRAM 上に描画する。次いで、LoS を判定したい 2 点間に線を描画するコマンドを発行する。その際、現在描こうとしているピクセルが、既に描画されている地表よりも向こう側にあるときのみ描画するように設定する。

このとき描画されるのは、本来であれば地形の内側に飲み込まれるピクセルであり、空中を通る部分は描画されない。そのため、描画コマンド終了後、実際に描画されたピクセルがあるか GPU に問い合わせ、存在すれば LoS は false であると判定できる(図 4)。

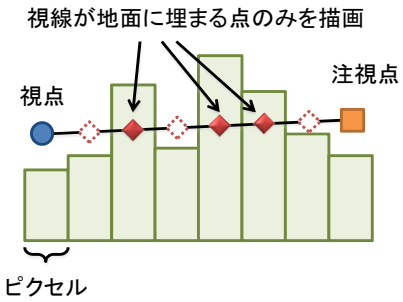

図 4　Salomon らのアルゴリズム

3.　面積 LoS

本研究で提案する LoS が、面積 LoS である。これは点と点との間で LoS を行うものではなく、視点から対象を見たときに、そのうちどれだけが露出しており、どれだけが地形に隠れているかを判別するものである（以下面積 LoS と呼ぶ）。そのため、結果は true/false ではなく、実数値によって与えられる。

視点から対象が「どの程度見えるのか」を求めるためには、対象の面積と、障害物に隠されている部分の面積を求めればよい。以下、面積を求めるための分解能を LoS の解像度又は単に解像度と呼ぶ。

3.1　GPU を利用したアルゴリズム

まず、視点から見た対象をビデオメモリに確保した仮想画面に描画し(1 度目の描画)、描画したピクセル数(対象ピクセル数)を取得する。ここでは、地形は描画せず、対象のみを描画する。

この時点でいったん深度バッファをクリアし、次に地形を描画する。それから、深度テストを有効にした状態で再度対象を描画し(2 度目の描画)、描画したピクセル数(可視ピクセル数)を取得する。

2 度目の描画の際には深度テストが行われるため、地形に隠れるピクセルは描画されない。そのため、対象ピクセル数と可視ピクセル数を比較することで、対象が見えている割合を判断することができる(図 5)。

Occlusion query の機能に対応している GPU であれば、描画からピクセル数の計測までをビデオカード内で実行可能である。

以下、GPU 法と記した場合には本項で説明したアルゴリズムのことを言う。

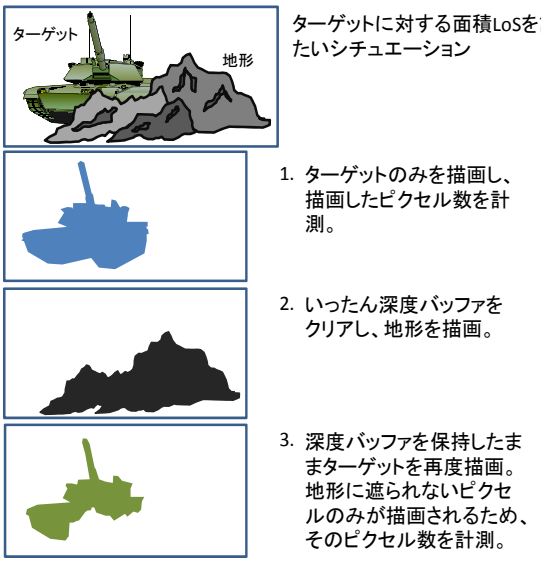

図 5　GPU を用いた面積 LoS のアルゴリズム

3.2　実装

本研究では、GPU 法による実験用のプログラムを Java と JOGL の組合せによって実装した。GLPBuffer を利用して、仮想画面を作成して描画処理を行う。

LoS 判定のための地形を描画する場合、例えば視点から見て目標の向こう側の頂点は LoS に影響しないため、これらの頂点に対する座標演算を行うのは無駄である。座標演算の量のみから考えると、視線の経路上にあるメッシュだけを描画するのが最良である。

一方で、性能のためには、地形を構成する頂点情報をあらかじめ VRAM に転送しておき、まとめて一括描画することが望ましい。地形を構成するメッシュを 1 枚ずつ描画しようとすると、CPU やバス帯域がボトルネックとなり、GPU の性能を発揮できないからである。

これらの条件を両立させようとする場合、地形をある程度の大きさのエリアに分割し、エリア単位で描画を管理するというのが現実的な解答となる(図 6)。

今回は、地形を縦横 4 等分した 16 エリアに分割し、エリアごとに OpenGL の描画単位である頂点バッファオブジェクトとして登録した。視線の経路にあるエリアのみを描画することで、座標演算の負荷を軽減するよう実装した。

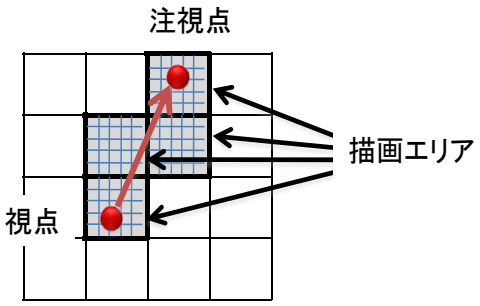

図 6　地形の分割

4.　実験方法

今回、仮想空間上の座標に配置された板について、この板が視点から地形に遮られずに見える面積を、割合で示すこととした。板の大きさは高さ 2m、幅 1m とした。板は水平面に垂直であり、視点の方向に正対するように配置するものとする。また、視点の高さは地表から 1.5m とした。

計算環境としては、計算機(1)（デスクトップ PC であり、4 コアの CPU とチップセット統合型 GPU を搭載）と、計算機(2)（ノート PC であり、2 コアの CPU と、単独の GPU を搭載）の 2 種類で測定した。それぞれの計算機の仕様と性能を表 1と表 2に示す。

表 1　計算機(1)の性能

CPU	Intel Core2 Quad Q8200 2.33GHz
GPU	Intel GMA X4500
VRAM	256MB (メインメモリに確保)
3Dmark05 スコア	1143
OS	Windows Vista
メインメモリ	4GB

表 2　計算機(2)の性能

CPU	Intel Core2 Duo T9300 2.5GHz
GPU	NVIDIA GeForce8600M GT
VRAM	256MB
3Dmark05 スコア	6463
OS	Windows Vista
メインメモリ	4GB

4.1　テスト用地形データ

実験用の地形データとして国土地理院のデータを利用し、計算機上に仮想環境を構築した。テスト地形は縦横 500m、メッシュサイズ 5m であり、100×100 の格子によって構成されている。最大標高差は約 15m である。

なお、地球の丸みは考慮しないものとした。

4.2　面積 LoS の計算

CPU によるステップ法と GPU 法により、面積 LoS を計算した。ステップ法では、対象(今回は板)をメッシュ状に分割し、各メッシュに対して LoS 判定を行うこととした。LoS の解像度が 10×10 だった場合、100 回の LoS 判定を行う。解像度の数だけ実行した LoS 判定のうち、LoS が成立したものの割合によって面積 LoS を求める。ステップ長はメッシュサイズと同じ 5m とした。

また、プログラムはマルチスレッドで動作するものとし、計算機の CPU コアの数だけスレッドを立ち上げるようにした。

4.3　精度の検討

CPU と GPU によって LoS を計算する際の精度について考える。2 項で述べたステップ方式による LoS 判定を行う場合、精度は高度サンプリングのステップ長に依存し、ステップ長を短くすれば精度は向上するが、その分計算量が増大することになる。

GPU 法を利用して LoS を判定する場合、ステップ長という概念は存在せず、精度は GPU の計算精度に依存することになる。多くの場合、GPU は単精度浮動小数をサポートし、内部での座標変換等に利用している。

2 点間の LoS 判定について、ステップ方式と GPU による方式とで結果を比較した。点と点との LoS であるため、結果は true/false となり、GPU 法の場合、1 ピクセルの面積 LoS を行うのと同じことである。

後述する表 4の座標セットを用いて 10000 回の試行を行った。両者で違う結果が出た回数を表 3に示す。ステップ長は、メッシュサイズと同じ 5m から、メッシュサイズの 1/10000 にあたる 0.0005m まで、10 倍刻みで設定した。

ステップ長を短くすると両者の結果は近づく。ただし、ステップ長を短くしても両者が完全に一致するわけではない。これは、メッシュの補間方法の違い(ステップ法の場合は近傍 4 点を利用したバイリニア補間。GPU 法の場合は近傍 3 点を利用した線形補間)によるものと考えられる。

今回用意した環境は規則正しい地形メッシュによって構成されているが、ポリゴンで構成された建物等が視界を遮蔽する物体として存在する場合、ステップ法ではステップ長をより細かく設定する必要が生じる可能性がある。

表 3　2 点間 LoS を 10000 回実施した場合

ステップ長	CPU と GPU で違う結果が生じた回数
5m	34 回
0.5m	10 回
0.05m	7 回
0.005m	6 回
0.0005m	6 回

5. 実験結果

計算機(1)による結果を表 5と表 6に、計算機(2)による結果を表 7と表 8に示す。入力として、視点位置と目標位置の組を 10000 セット用意し、面積 LoS を計算してその計算時間を計測した。視点と目標の位置はランダムに設定し、最少距離は 10m 以上になるようにした(表 4)。

計算時間は 10,000 回の LoS 計算をした合計値である。

表 4　視点座標と目標座標

セット数	10,000
平均距離	259.35
最大距離	670.23
最少距離	10.69

表 5　ステップ法による解像度と計算時間の関係(1)

LoS 解像度	計算時間(msec)
64×64	62536
128×128	255484
256×256	992657
512×512	3790350

表 6　GPU による解像度と計算時間の関係(1)

LoS 解像度	計算時間(msec)	ステップ法との速度比
64×64	5275	11.86
128×128	5399	47.32
256×256	7305	135.89
512×512	18162	208.70

表 7　ステップ法による解像度と計算時間の関係(2)

LoS 解像度	計算時間(msec)
64×64	97369
128×128	367995
256×256	1361898
512×512	5677559

表 8　GPU による解像度と計算時間の関係(2)

LoS 解像度	計算時間(msec)	ステップ法との速度比
64×64	4151	23.46
128×128	4408	83.48
256×256	5222	260.80
512×512	8500	667.95

6. 考察

GPU を利用することで、見えるか見えないかだけではなく、「どの程度見えているのか」を実用的な速度で表現する方法について提案し、その効果を確認した。CPU で計算するステップ法に対して、GPU 法による大幅な高速化が確認できた。また、その際の速度性能は主に GPU の性能に依存することが確認された。

高速な GPU を搭載する計算機(2)においては、512×512 の解像度であっても平均 1msec 以下で処理できている。これは、シミュレーションシステムに 100 のユニットが登場し、それぞれ 10 のユニットに対して面積 LoS の処理を行うとしても、全体の LoS 処理を 1 秒以下で終了できることを意味する。

また、GPU を利用する場合には解像度を細かくしても計算時間は比例しないことが分かった。これは、処理のうち座標演算の計算量が解像度に依存しないためと考えられる。

7. 応用と今後の課題

今回、単純な矩形のポリゴンに対して LoS を行ったが、原理的には矩形である必要はなく、自由な形状のポリゴンモデルに対して LoS 判定を行うことができる。また、今回は LoS 対象として単色で塗りつぶされたポリゴンを利用したが、例えば人間の形状をしたモデルを利用し、頭部や胴体、四肢について別個に計算することもできる。部分によって重みを設定することで、「頭部は判別しやすく、四肢は見えていても判別しにくい」等の処理も容易に可能である。

さらに、透明色を含んだテクスチャマッピングを利用することで、テクスチャで表現された木立の向こうを透かし見るような表現も可能である。環境においても、今回はメッシュで表現された地形のみを対象としたが、より複雑な市街地等においても、建造物等をポリゴンで表現することで同様に処理を行うことができる。

また、視点と目標との 1 対 1 での対応に限らず、視点からの視界範囲全体を仮想画面に描画し、視界内の複数の目標について、まとめて LoS を行うことも可能である。

本報告では、汎用性を考慮して実装は Java6 及び OpenGL2.1 の範囲で行った。GPU の汎用的な利用のため、近年は CUDA や ATI Stream 等の API も提供されている。これらは GPU のハードに依存し、標準化されたものではないが、標準仕様となる OpenCL も普及しつつある。GPU を利用するための汎用的な枠組みが定まることで、さらなる応用が期待できる。

参考文献

[1] Brian Salomon et.al, “ACCELERATING LINE OF SIGHT COMPUTATION USING GRAPHICS PROCESSING UNITS “ , Proc. of Army Science Conference,2004

[2] OpenGL 策定委員会 松田晃一訳 “OpenGL プログラミングガイド 原著第 5 版”,ピアソンエデュケーション,2006