2021 年度 卒業論文

Bachelor's Thesis

音による状況変化を用いたゲーム実現のための空間音響システムの開発

Development of a spatial audio system for realizing games that use sound to change the situation

琉球大学工学部工学科知能情報コース

185761E　多和田 真都

指導教員　河野　真治

要旨

ゲームにおいて空間音響は, ユーザーに特定の場所を印象付けたり空気感の演出などの補助的な役割で用いられてきた. 音の回折による音源方向の変化をゲームロジックに組み込むことで, 空間音響を補助的な役割のみならず, ゲームの遊びを構築するコアな要素にする事ができると考えられる.

本研究では, 上記のようなゲームを構築するために必要な音が壁を回り込むことで音源の位置が変化する現象を, 経路探索による回折経路の計算で実現できるかどうかの検証を行なった. また, 動的に変化する環境に対応するために, 経路探索用のグラフをリアルタイムで再構築できるよう実装した.

Abstract

In games, spatial audio has been used as a supplementary role to impress users with a specific location or to create a sense of atmosphere. By incorporating changes in the direction of the sound source due to sound diffraction into the game logic, spatial acoustics can be made not only a supplementary role but also a core element for building game play.

In this study, we verified whether the phenomenon of sound source position change due to the sound going around the wall, which is necessary to construct the above game, can be realized by calculating the diffraction path by path finding. In addition, to cope with the dynamically changing environment, we implemented a method to reconstruct the pathfinding graph in real time.

目次

第 1 章

序論

1.1　空間音響をゲームロジックに取り入れる

昔からゲームに空間音響を取り入れることはよく行われていた. 例えば, プレイヤーが洞窟の中に居るのならば音を響かせて狭い空間に居ることを演出したり, 音源とプレイヤーとの間に遮蔽物があれば音を篭らせるなどである. 比較的最近発売されたゲームでは NiaR:Automata[1] が空間音響をゲームに取り入れた例として代表的である.　こういったゲームでは空間音響を用いて特定の地点をより印象付けたり, プレイヤーがフィールドを探索する際にシームレスに音響が変化していくことで, より現実に近い自然な体験ができる様になっている. その一方で, 空間音響はゲーム体験を補強する役割のみにとどまっておりゲームを構成する要素で最も重要なゲームロジックまでは関わってこない. 仮に空間音響をゲームロジックに組み込む事ができれば次の様な体験ができると考えられる.

まず 2 人以上で対戦する FPS(First Person Shooting) ゲームを想定する. プレイヤー同士が壁を挟んで相対している状況において, 空間音響が有効でない場合音が壁越しに直接聞こえ, プレイヤーは相手の位置を正確に完全にすることが出来る. その結果, 音を目安に壁を飛び出して正確に相手を撃つ, という精密さが求められるゲームとなる. 空間音響が有効な場合, 音は壁の切れ目から回り込んで聞こえるためプレイヤーは相手の位置を正確に掴めず, 壁越しに敵がいるという情報しか得られない. これによってプレイヤーは慎重に行動する様になるため, 精密さが重要だったゲームから慎重さとリスク管理が求められる緊迫感のあるゲーム体験に変様する. このように, あくまで補助的な要素だった空間音響はゲームロジックに組み込むことでゲーム性そのものを変化させることのできる要素になる可能性がある. 上記の空間音響を利用したゲームではプレイヤーが遮蔽物を設置して音の聞こえ方を変化させたり, 壁に穴を開けて敵の油断をついたりなどの要素が追加で

きると考えられる. その場合, 遮蔽物の増減に合わせて音響はリアルタイムで変化していくため, その度に音響の再計算を行う必要がある. そこで本研究では, 経路探索を用いた回折現象の再現や音響パラメータの計算の軽量化などの実装を行いリアルタイムで音響計算が行えるシステムの実現を目指す.

第 2 章

基礎概念

2.1　OpenAL について

OpenAL とはクロスプラットフォームに対応し, ゲームやその他の多くのオーディオアプリケーションで利用できる 3D オーディオ API である. いくつかの実装が存在し, 現在は OpenAL-Soft が主流な実装となっている. 3 次元空間上での音の表現に適しており, 簡素な記述で立体的な音場を表現する事ができる.

OpenAL の構造は以下の様になっている.

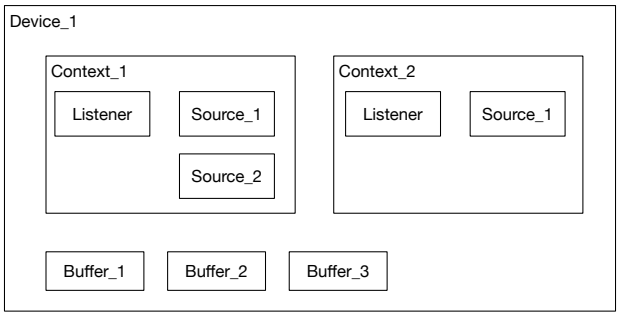

図 2.1　OpenAL の基本構造

Device の下にコンテキストと, 音源の実際の信号などを保持している Buffer が存在する.Context は複数存在し, それぞれの Context に音を受け取る役割を持つ Listener が 1 つある. Source は Context 内に複数個設ける事ができ,Buffer からの情報を読み取り実際に音声を再生する役割を持つ.

2.2 Unity について

Unity は Unity Tecnologies が開発しているクロスプラットフォームに対応したゲームエンジンである．C#でのプログラミングが可能で, レンダリングの機能も充実しているためゲーム制作以外でも, 研究用途やコンピュータグラフィックスの分野でも用いられる．本研究では C++ を用いて OpenAL による 3 次元音響のシステムを構築するため C++ によるコードを Unity 上で利用できる様にする必要がある．Unity には Unity native plugin という機能があり,C や C++ その他の言語で書かれたプログラムに対して C のインターフェースを介してアクセスする事ができる．

2.3 レイキャストについて

コンピューターにおいて, ある地点から特定の方向に障害物が存在するかを判定する際にレイキャストが用いられる．走査したい方向に仮想的な光線を飛ばし, その光線と物体を構成するポリゴンと呼ばれる 3 角形が交差しているか否かで障害物の有無を判定する方法が代表的である．この様な実装を工夫をせずに行うと全てのポリゴンに対して光線との交差判定をする必要があるため無駄な処理が多い. そのため通常は後述する空間分割などを用いて判定するポリゴンを減らす工夫がなされる．

2.4 Any-angle path planning

ゲーム上のある地点からある地点までの最短距離を求めたい場合, ダイクストラ法 [2] を用いた経路探索アルゴリズムが用いられる．ダイクストラ法ではノードとそれらを繋ぐエッジで構成されたグラフ構造を基に最短経路を求める．3 次元空間上ではノードが空間上の位置を表し, エッジはその空間同士が接続されているか (ノード間に障害物がない) を表現している．ゲームではダイクストラ法の中でも A*アルゴリズムがよく利用されているが, 欠点も存在する．それは,A*は探索を始めるノードから隣接するノードを辿っていくことで経路を計算するため, ノード間の角度がグラフの構造に依存してしまうのである. 例えばグラフ上のそれぞれのノードが隣接する上下左右の 4 方向のノードのみと接続されている場合, 計算された経路は 90 度間隔でしか他のノードと接続できない．これを解決するアルゴリズムが Any-angle path planning である．Any-angle path planning はパスがグラフの構造に制限される事なく, 任意の角度でノード同士を接続することができるアルゴ

リズムである.Any-ange でない経路探索アルゴリズムと比較してより適切なパスを計算できる.

2.5 Theta*アルゴリズム

Any-angle path planning の一つである Theta*は親ノードを設定する際に, 隣接しているノードの親ノードとの間に障害物があるかどうかを探索し障害物がなければ親ノードにその親ノードを設定する.

以下に Theta*の疑似コード [4] を示す. Theta*で特徴的なのが, アルゴリズム 1 の 19 行目の引数のノード間の障害物の有無を返す LineOfSight 関数である. 自分の親ノードと隣接ノードの LineOfSight を判定し, 障害物がない場合隣接ノードに自分の親ノードを直接設定する. そのため,Goal ノードから親ノードを順にたどってパスを生成する際に余分なノードを経路に含めずに計算する事ができる.

Algorithm 1 Theta*

```
1: function MAIN(void)
2:     g(start) := 0
3:     parent(start) := start
4:     open := ∅
5:     closed := ∅
6:     open.Insert(start, 0)
7:     while open ≠ ∅ do
8:         n = open.Pop()
9:         if n = n_goal then
10:            return "Find path"
11:        close := close ∪ n
12:        for all n' ∈ neighbors(n) do
13:            if n' ∉ close then
14:                if n' ∉ open then
15:                    g(n') = ∞
16:                    parent(n') = NULL
17:                UpdateVertex(n, n')
18: function UPDATEVERTEX(n, n')
19:    if LineOfSight(parent(n), n') then
20:        if g(n) + c(n, n') < g(s') then
21:            g(parent(n)) := g(n) + c(parebt(n), n')
22:            parent(n') := parent(n)
23:            if n' ∈ open then
24:                open.Remove(n')
25:            open.Insert(n',g(n')+h(n'))
26:    else
27:        if g(n) + c(n, n') < g(n') then
28:            g(n') := g(n) + c(n, n')
29:            parent(n') := n
30:            if n' ∈ open then
31:                open.Remove(n')
32:            open.Insert(n', g(n') + h(s'))
```

2.6 空間分割

2.3 の最後で述べたように空間を離散化せずにレイキャストなどの衝突判定を行うことは計算量の観点から見て非効率なため, 同時に空間分割を用いた最適化がなされる場合が多い. 空間分割の方法には 4 分木や kd 木,BSP 木など分割の仕方が異なる方法が多数存在するが, 基本的には空間を分割しそれぞれの空間に属するオブジェクトを登録し, 分割した空間をさらに分割して子空間を生成, 子空間に属するオブジェクトをその子空間に登録, という操作を繰り返してオブジェクトが所属する空間を小さくしていくというのが主な目的である.

以下に 4 分木での具体例を示す.

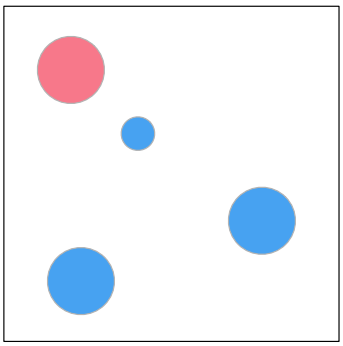

図 2.2 空間分割なし

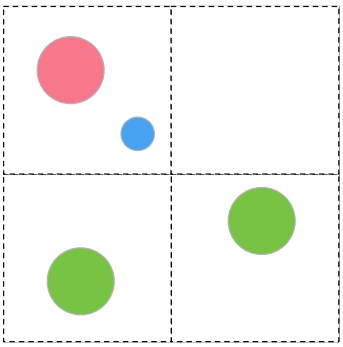

図 2.3 空間分割あり

図 2.6 では, 赤色のオブジェクトが他の物体と衝突しているかを判定するためには, 青色のオブジェクト全てに対して計算を行わななければならないが, 図 2.3 では 4 分木アルゴリズムを使用し空間が 4 つに分割されており, 他の空間に属している緑色のオブジェクトに関しては衝突していないことが保証されるため, 同じ空間に属する青色のオブジェクト 1 つに対してのみ計算することで衝突する可能性のあるオブジェクトの判定を行う事ができる. 分割は任意の数だけ行う事ができ, 分割した空間の一つをさらに 4 つに分解して粒度を細かくする事が可能である. オブジェクトの数が多い際は分割数を増やすことで衝突判定の計算量を減らせるが, メモリの使用量が増加するため空間の特性に応じた適切な分割数を用いる必要がある.

2.7 8 分木

8 分木は 4 分木を 3 次元空間上で利用できるようにしたアルゴリズムである. 一つの空間を 8 つの立方体に分割することを繰り返していくことで, それぞれのオブジェクトの所属空間を決定する.

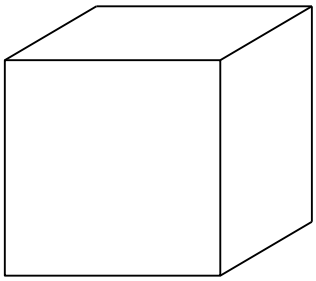

図 2.4 分割数 0

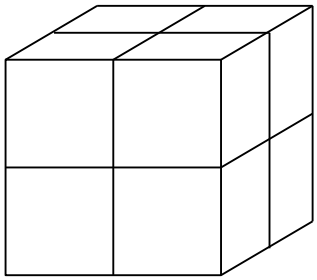

図 2.5 分割数 1

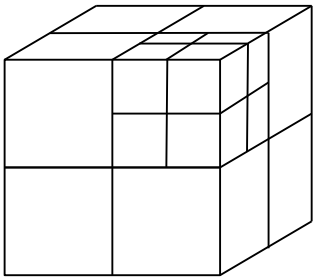

図 2.6 分割数 2

第 3 章

Kingdom Spatial Audio

3.1　概要

本章ではリアルタイムで音響を再計算, 再生するシステムとして制作した Kingdom Spatial Audio(以降,KSA) の構成について説明する. KSA では聴覚的な影響が大きい反射と, 音が壁を回り込んでくる回折現象を主に再現するシステムとなっている. それぞれの音響効果をシミュレートするにあたって,KSA ではレイキャストと経路探索を用いてシステムを構築している.

レイキャストを用いることで周囲の遮蔽物の状態を取得する事が可能となり, 音の反響時間や反射による音の増幅の計算をすることが可能になる. 経路探索では音が回り込むために生成される回折経路を近似し, 音が聞こえる方向を変化させる効果と音の高周波成分が減衰する現象を再現する.

また, 経路探索は計算に時間を要するため, メインスレッドで処理を行うと同じくメインスレッドで動作している Unity の動作が停止し画面が固まってしまうなどの悪影響が発生する. そのため, 経路探索はサブスレッドで処理を行う様にし, 複数の経路探索が同時に行われても問題なく実行できるように構築している.

3.2　システムの流れ

図 3.1 にシステムの大まかなフローチャートを示す. まずゲーム開始時に初期化処理としてワールドのオブジェクト情報を読み込む. 次に, そのオブジェクト情報から 8 分木を構築する. 初期化処理の完了後は, 毎フレーム実行されるゲームループでリスナーの位置を更新し, 計算した回折経路を元にオーディオプレイヤーの位置を変更する. レイキャス

トを用いた反響の計算を行い, 必要に応じて 8 分木とグラフの更新を非同期で実行する.

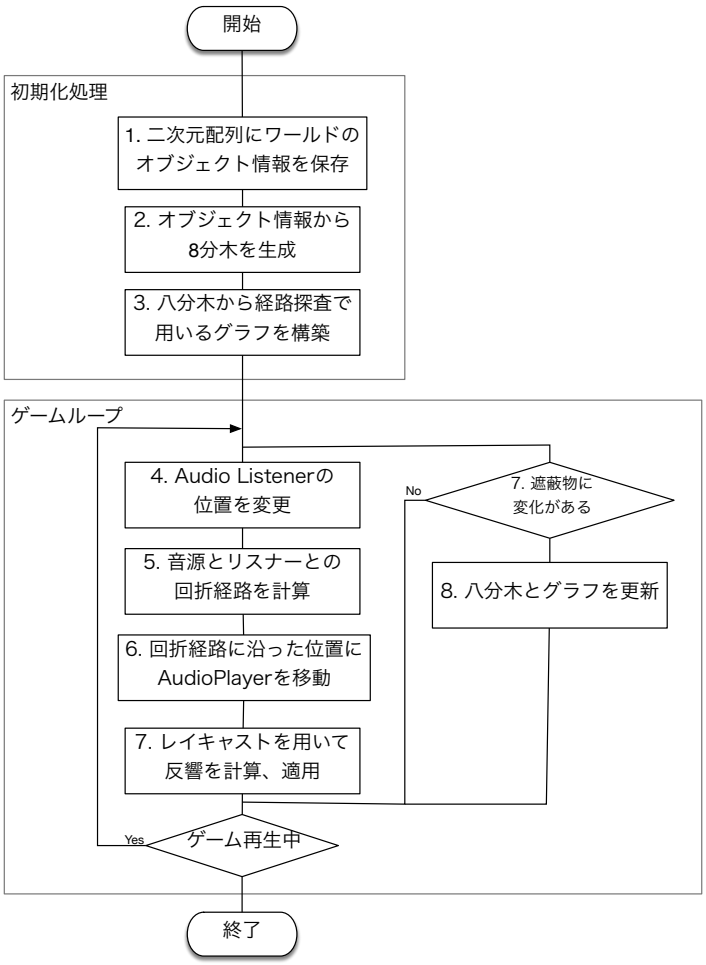

図 3.1　KSA のフローチャート

3.3　反射の再現

音は障害物に衝突すると, 一部は吸収され残りは反射されて空間に放出される. それが繰り返された結果, その場にいる人間には音が反響して聴こえる様になる. これはコンピュータ上ではリバーブエフェクトとして再現される. そこで,KSA ではリバーブエフェクトに渡すパラメータを計算することで音の反射を再現する.

リバーブのパラメーターには反射音の遅延時間や反響音の持続時間などを渡す必要がある. 反射音の遅延時間は音が音源から発生し壁に反射した後リスナーに届くまでの時間で

計算できる. 反響音の持続時間は以下に示される Sabine の式 [3] を利用して求める事が可能である.

$T = \frac{0.161V}{S*A}$

T:残響時間

S:空間の表面積

A:吸音率

どちらのパラメータを求めるにも音源の周囲の環境を測定する必要があり, 本システムではレイキャストを用いてこれらの計算を行う.

空間分割を行わずにレイキャストなどの衝突判定を行うことは計算量が多くなってしまうため最適ではない. そこで, レイキャストを行う前に 8 分木を用いた空間分割を実行する.

図 3.2 は 4 分木の分割の様子を示している. 可視化のしやすさのために 4 分木を用いているが, 基本的な考え方は 8 分木も同様である. まず空間が空白ブロックと遮蔽ブロックで構成されているとする. 分割レベル 0 では一つの空間に灰色の遮蔽ブロックと空白ブロックが混ざっている. そこでさらに分割数を増やす. 分割レベル 1 では空間 1 と空間 3 が同じブロックのみで構成されているためこれ以上分割する必要はない. 分割レベル 2 では空間 0 と空間 2 の分割数をさらに増やし, 一種類のブロックのみで空間が構成される様にする. この様な分割を行うことで, 空間 1 と 3 関してはその中に含まれる 4 つのブロックに対して個別に衝突判定を行う必要がなくなり, 空間そのものと衝突判定を行えば良くなる.

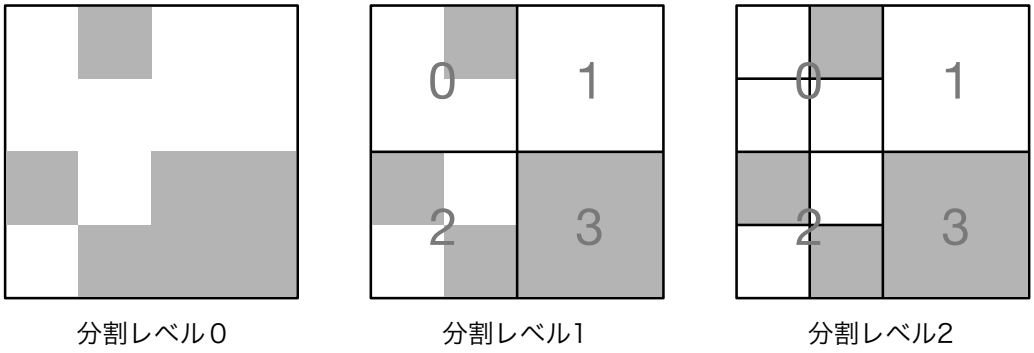

図 3.2 4 分木の分割の様子

次に,4 分木を用いたレイキャストの方法について説明する. 図 3.3 に示すように,4 分木

の分割の際と同じ環境を用いて, 赤色の直線がオブジェクトに遮られていないかを判定する. ここでは遮られているかどうかの判定に直線が通っている空間の中に遮蔽ブロックが存在しないことを条件として用いる. 図 3.4 では, 分割レベル 1 の 4 つの空間の中から遮蔽オブジェクトが含まれている可能性のある空間 0,2,3 の空間と交差判定を行っている. 実際に交差している空間 0 にはさらに分割された子空間が存在するため, その子空間に対しても交差判定を行う. 子空間 0' 3' の中から遮蔽ブロックが含まれる空間 1' と交差判定を行う. 直線は空間 1' と交差していないため直線が交差している全ての空間の中に遮蔽オブジェクトが含まれていないことが確認できる. よって直線は遮蔽オブジェクトに遮られていないと判定できる. 上記のようなレイキャストの実装を行うことで, 通常は遮蔽物の数だけ行う必要があった交差判定が, 空間 0,2,3,1' の計 4 回で済むようになり高速化が見込める.

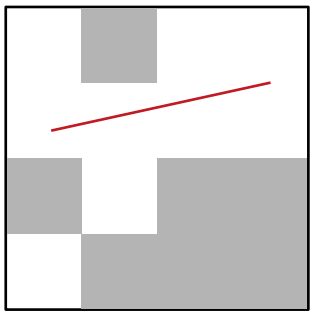

図 3.3　レイキャストの環境

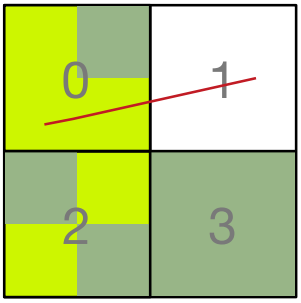

図 3.4　空間 0,2,3 との交差判定

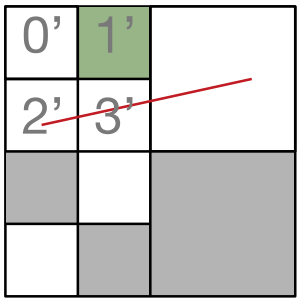

図 3.5　空間 1' との交差判定

これを 3 次元に拡張し 8 分木に適応したプログラムを KSA に実装した.

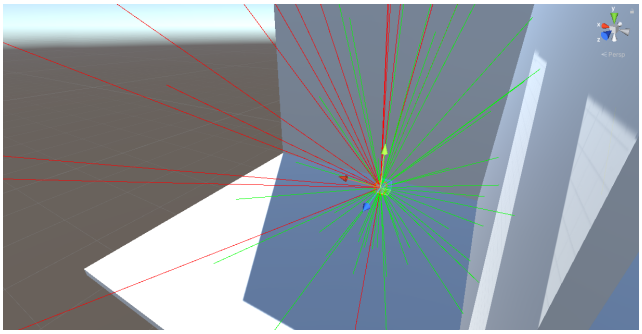

図 3.6　レイキャスト

3.4　回折の再現

音は障害物の端から回り込んで伝播する性質を持っている. 高周波の音ほど回折は起こりにくいため, 壁を挟んで音源の反対側にいる人物には音が篭って聞こえる. また音の方向も音源自体ではなく回り込んでくる壁の端から聞こえるようになる. 本システムでは経路探索を用いて音の回折経路と回折してきた音の方向を計算する.

回折経路を算出する際, 通常のグラフ探索アルゴリズムでは計算された経路がグラフの離散化された位置に依存してしまうため, 図 3.4.1 のようにギザギザの経路が算出されてしまい, 期待する結果よりも経路長が長くなってしまう. そこで KSA では Any-angle path planning アルゴリズムと edge-corner graph を用いた最短経路探索を行う.

3.4.1　Any-angle path planning

Any-angle path planning アルゴリズムの一つである Theta*[4] は A*アルゴリズムをベースに, 間に障害物がないノード同士を接続することで図 3.8 のような直線的で自然な経路を生成している.

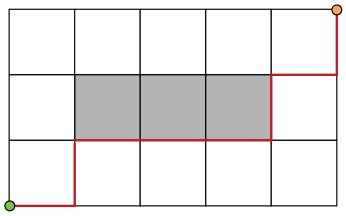

図 3.7　A*による経路生成

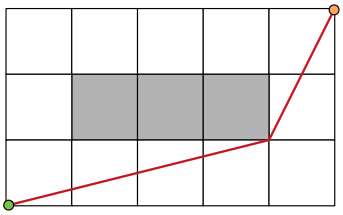

図 3.8　Theta*による経路生成

3.5　8 分木によるグラフの構築

8 分木は一つの空間に含まれる情報を少なくするために, 一つの 3 次元空間を 8 つの空間に分割しこれを再帰的に繰り返して木構造を構築するアルゴリズムである. 本システムで構築した 8 分木は, 空間に含まれるオブジェクトの種類が複数存在する場合に分割を行い, 一つの空間に含まれるオブジェクトの種類が一つになるまで空間を小さくしていく方法を用いている.

構築した 8 分木空間の頂点にノードを生成し, 遮蔽物で遮られていない隣接するノード

同士を接続していきグラフを構築する．この様なグラフを edge-corner graph と呼び, 何もない場所が多いような疎な空間において生成されるノードの密度が低くなり, グラフの探索効率が上昇する.

3.6 経路探索のサブスレッド化

経路探索は本システムの計算処理の中でもかなり計算量が多いアルゴリズムとなっている．グラフのサイズが 128 の場合, 経路探索に 27ms かかっている. 一般的なゲームが最低 60fps(1 フレーム 16.6ms) で動作することを鑑みると, この計算時間はフレーム落ちやフレームレートの不安定化の原因になるため避けたい現象である．そこで経路探索をサブスレッドで行うようにし, メインスレッド動作している Unity の処理を停止させないようにする.

表 3.1 経路探索の計算時間

グラフの一辺のサイズ (m)	計算時間 (ms)	経路長 (m)
128	27.8668	341
64	5.7981	170
32	4.5226	88
16	1.5909	44

メインスレッドで行っていた処理をサブスレッドに移行するにあたって, 複数のサブスレッドから同時にアクセスされる可能性を考慮する必要がある．複数のスレッドから同時に書き込みが発生すると, 最後に書き込まれた情報以外は消失してしまうことになるためデータの整合性が取れなくなったり, プログラム上で意図しないエラーが発生する．これを解決するために, ソースコード 3.1 の 45 行目で行なっているグラフにスタートノードとゴールノードを書き込んでいた処理を削除し, ソースコード 3.2 の 42 行目のように別の変数に保持しておき, 探索時に適切なタイミングで前述の二つのノードを読み込む様に変更した.

Listing 3.1 グラフに直接ノードを書き込む場合

```
1 VertexNode* ThetaStar::CreateEndNode(Utility::Vector3 endPos,
      VertexGraph* graph, Octree* octree)
2 {
3     VertexNode* endNode = new VertexNode();
```

```
4       endNode->id = -2;
5       endNode->nodePosition = endPos;
6       endNode->h = 0;
7
8       Utility::Vector3Index index = Utility::Vector3Index{ (int)floorf(
            endPos.x), (int)floorf(endPos.y),(int)floorf(endPos.z) };
9       int morton = OctreeUtility::Index2Morton(index);
10
11      int maxlevel = octree->getMaxOctreeLevel();
12
13      OctreeNode* targetNode = octree->getOctreeRoot();
14      for (int i = 1; i <= maxlevel; i++)
15      {
16          if (targetNode->attribute == OctreeNodeAttribute::Branch)
17          {
18              int childSpatialIndex = (morton & (7 << ((maxlevel - i) *
                    3))) >> ((maxlevel - i) * 3);
19              targetNode = &targetNode->childNodes[childSpatialIndex];
20          }
21      }
22      int lowestMorton = (int)pow(8, maxlevel - targetNode->nodeLevel) *
             targetNode->mortonNumber;
23      Utility::Vector3Index lowestIndex;
24      OctreeUtility::Morton2Index(lowestMorton, maxlevel, &lowestIndex);
25
26      int nodeScale = (int)pow(2, maxlevel - targetNode->nodeLevel);
27
28      for (int x = 0; x <= 1; x++)
29      {
30          for (int y = 0; y <= 1; y++)
31          {
32              for (int z = 0; z <= 1; z++)
33              {
34                  int edgeIndex = x + (2 * y) + (4 * z);
35
36                  int graphIndex_x = lowestIndex.x + (x * nodeScale);
37                  int graphIndex_y = lowestIndex.y + (y * nodeScale);
38                  int graphIndex_z = lowestIndex.z + (z * nodeScale);
```

```
39
40                  int index = Utility::Vec3IndexToLinerIndex(graph->
                        getGraphSize(), graphIndex_x, graphIndex_y,
                        graphIndex_z);
41
42                  graph->vertexGraph[index].neighbors.resize(7);
43                  graph->vertexGraph[index].neightborDistance.resize(7);
44
45                  graph->vertexGraph[index].neighbors[6] = endNode;
46                  graph->vertexGraph[index].neightborDistance[6] =
                        Utility::Distance(endPos, Utility::Vector3{ (float
                        )graphIndex_x, (float)graphIndex_y, (float)
                        graphIndex_z });
47              }
48          }
49      }
50
51      return endNode;
```

Listing 3.2　グラフにノードを書き込まない場合

```
1  VertexNode* ThetaStar::CreateEndNode(Utility::Vector3 endPos,
       VertexNode* graph, unordered_map<int, float>* nodeAdjacentGoal,
       Octree* octree)
2  {
3      VertexNode* endNode = new VertexNode();
4      endNode->id = -2;
5      endNode->nodePosition = endPos;
6      endNode->h = 0;
7
8      Utility::Vector3Index index = Utility::Vector3Index{ (int)floorf(
           endPos.x), (int)floorf(endPos.y),(int)floorf(endPos.z) };
9      int morton = OctreeUtility::Index2Morton(index);
10
11     int maxlevel = octree->getMaxOctreeLevel();
12
13     OctreeNode* targetNode = octree->getOctreeRoot();
14     for (int i = 1; i <= maxlevel; i++)
```

```
15      {
16          if (targetNode->attribute == OctreeNodeAttribute::Branch)
17          {
18              int childSpatialIndex = (morton & (7 << ((maxlevel - i) *
                    3))) >> ((maxlevel - i) * 3);
19              targetNode = &targetNode->childNodes[childSpatialIndex];
20          }
21      }
22      int lowestMorton = (int)pow(8, maxlevel - targetNode->nodeLevel) *
         targetNode->mortonNumber;
23      Utility::Vector3Index lowestIndex;
24      OctreeUtility::Morton2Index(lowestMorton, maxlevel, &lowestIndex);
25
26      int nodeScale = (int)pow(2, maxlevel - targetNode->nodeLevel);
27
28      for (int x = 0; x <= 1; x++)
29      {
30          for (int y = 0; y <= 1; y++)
31          {
32              for (int z = 0; z <= 1; z++)
33              {
34                  int edgeIndex = x + (2 * y) + (4 * z);
35
36                  int graphIndex_x = lowestIndex.x + (x * nodeScale);
37                  int graphIndex_y = lowestIndex.y + (y * nodeScale);
38                  int graphIndex_z = lowestIndex.z + (z * nodeScale);
39
40                  int index = Utility::Vec3IndexToLinerIndex(vertexGraph
                        ->getGraphSize(), graphIndex_x, graphIndex_y,
                        graphIndex_z);
41
42                  nodeAdjacentGoal->insert(make_pair(graph[index].id,
                        Utility::Distance(endPos, Utility::Vector3{ (float
                        )graphIndex_x, (float)graphIndex_y, (float)
                        graphIndex_z })));
43              }
44          }
45      }
```

```
46
47      return endNode;
48  }
```

3.7 レイキャストの実行結果

以下に Unity で Theta*による経路探索を行なった結果を示す.

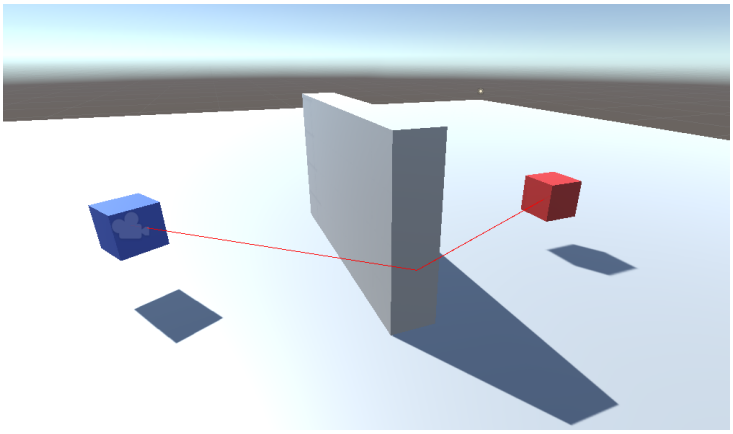

図 3.9 Theta*による経路の生成

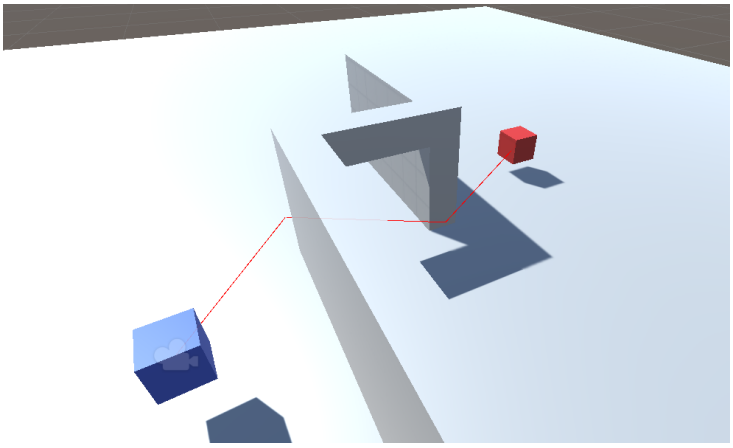

図 3.10 複雑な経路の生成

図 3.7 では音源である赤いブロックと, リスナーを表す青いブロックの間に遮蔽物が存在しており, 計算された経路は遮蔽物を横から回り込むようなパスになっている. 図 3.10 は音源とリスナーとの間に遮蔽物が 2 枚あるより複雑な環境を示しており, それぞれの壁で一度づつ経路が曲がって到達している事がわかる.

第 4 章

まとめ

本研究では空間音響の中の反射と回折を主に取り扱いシステムを構築した．経路探索によって音の回り込みを計算することができるため, 序論で述べたような空間音響をゲームロジックに取り入れたゲームの実現も可能であると考えられる．

しかし現時点での欠点として, 経路探索の再計算は一定間隔で行なっているため, 回折経路が変化するようなオブジェクトの変化が起きても, 再計算の時間になるまで音響が変化しない問題がある. これは回折経路が変化するかどうかを検知し, 即座に再計算を行うようにする必要がある．

別の問題として現実世界での回折経路は一つではなく複数存在する場合もあり, 本システムでは最短で到達する一つの回折経路のみを計算しているため, 最短の回折経路が別の経路になった際に音の方向が急激に変わってしまう現象が存在する. そのため, 回折経路を複数計算するか回折経路が一つのみになるような環境に制限したりなどの工夫が必要だ．

第 5 章

謝辞

本論文の制作にあたり, ご多忙の中にもかかわらず研究の方向性について幾度も道を示してくださった指導教官の河野真治准教授に深く感謝いたします. また, 研究を行うにあたって精神的にも支えになってくれた研究室の全メンバーに感謝を申し上げます.

参考文献

[1] SQUARE ENIX, NieR:Automata — SQUARE ENIX, https://www.jp.square-enix.com/nierautomata/

[2] Dijkstra, E.W. (1959). A note on two problems in connexion with graphs.

[3] "Acoustics Engineering - Sabin", https://www.acoustics-engineering.com/html/sabin.html#:~:text=Sabin%20is%20a%20powerful%20CAD,a%20measure%20of%20speech%20intelligibility.

[4] Kenny Daniel, Alex Nash, Sven Koenig, Ariel Felner, Theta*: Any-Angle Path Planning on Grids, 2014.

[5] Ruoqi He, Chia-Man Hung, PATHFINDING IN 3D SPACE - A*, THETA*, LAZY THETA* IN OCTREE STRUCTURE,pp.8-9, 2016.