山口大学教育学部研究論叢　第 68 巻　289 頁～ 302 頁　2019 年 1 月

プロシージャル方法による VR コンテンツの開発

熊谷　武洋

Study on Development of VR Contents with Procedural Method

KUMAGAI Takehiro

（Received September 28, 2018）

１．はじめに

近年、VRデバイスの機能的な性能向上に伴い、多くの対応コンテンツが、ゲーム分野などの娯楽分野をはじめ、教育、不動産、観光などの多岐にわたる分野で流通している。そのようなVRコンテンツの仕組みの多くは、視覚や聴覚に関わる膨大なアセットをスタティックなデータとして用意しておき、プログラムで逐次展開していくといった方式である。

このような方式を採用する各分野での典型例としては、エイリアンシューティング、歴史遺構デジタルアーカイブ、賃貸部屋VR内見システム、世界バーチャル観光ツアーなどといったコンテンツが列挙できる。このようなタイトルが一般利用者に対するVRコンテンツの印象や理解へと浸透していき、そのイメージは、もはや定着しつつある。

しかし、VR表現の可能性としては、これらの既存のシステム構成や表現様式以外にも、当然ながら多く残されている。とは言え、VRコンテンツ自体が、まだ普及期の段階にあり、その大部分が受託開発によるものであるため表現上の可能性の探求の余地はなく、その探求事例も多くはないという現状である。

そこで、典型事例に多く見られるような既存イメージの再現や標本化ではなく、形や音が実在しないものを現前化させ、VRデバイスの効力により仮想的な存在感を与えたときにどのような表現が可能になり得るか、VRというデバイスの特性自体に着目して実験的制作を試みた。

本稿は、試作したコンテンツの開発過程を述べるものである。

２．作品コンセプト

前述した典型的なVRコンテンツは、視聴後に何らかの主観的印象を抱くことを作用目的として、映像展開があり、背景音楽も一般的な音楽のセオリーに従って意図的に構成、編集したものが構成要素となる。いわば、ミュージックビデオやCM、マニュアルや映画の類である。VRであるからといって何か特殊な形式を有しているわけでない。

今回のコンセプトは、VRが持つ仮想的な事物を現前化させるという特性に着目し、表現形式の模索とする実験作品と定めた。よって、コンテンツの展開に何らかの具体的なゴールや目標設定は定めず従来型の構成要素は極力排することとした。

次に先行事例についてである。「Horizons」という興味深いタイトルがDaydremeアプリとしてリリースされている（図１）。

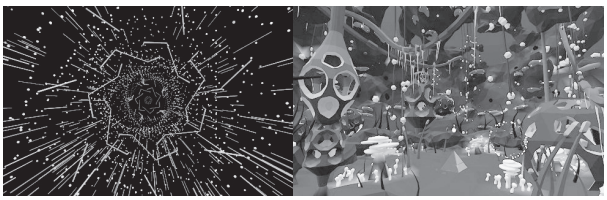

図１　Horizonsのプレイ画面

ジャンルとしては、オーディオビジュアライザーに区分される。背景音楽を聴きながら、スタイライズされた仮想空間を、ファーストパーソン視点で飛翔するといった内容である。インタラクティブ性は有しており、一部プロシージャル生成であるが、全体としてはアセットをプログラムで管理している従来方式である。

商業ベースでは数少ない例であるためにコンセプト的には大変興味深いタイトルではあるが、視覚表現と聴覚表現の実装方法は従来型であり、演出も新規性を有するものではないと言える。

本作においては、制作動機の点から、まず視覚表現において採用する方法として、ポリゴンメッシュ以外の方法を検討した。ボクセルや、ラインシェーディングなど様々な候補を検討したが、従来とは異なる表現を指向する上で重要な点は、オブジェクトモデルの定義を標本化

によるサンプリングではなく、人為によるモデリングでもないものでなければならないと考えた。そうなると必然的にプロシージャル生成であることが第一の条件となる。

そこで、視覚情報を生成するものとして、数理造形的手法を採った。その中でも視覚化の手法について多くの事例が存在し、以前に手掛けたモチーフとしてフラクタル図像を用いることとした［1］。

次に聴覚表現である。空間全体に漂う音のようなものを背景音楽や環境音として加えるにおいて、聴覚表現においても視覚表現と同様なコンセプトで検討した。前提となる類例は、アンビエント音楽やサウンドスケープである。つまり、アブストラクトな音響特性や音楽性を有するものを指向した。

そこで視覚表現と相乗的な効果を持つための聴覚表現の要素として、12音階といった西洋音楽的な要素を排し、雑音ではなく楽音でもない音色が、明確なメロディーやリズムを持たずに、曖昧かつ輪郭のない揺らぐようなシーケンスが循環するというイメージを着想した。

こうしたサウンド生成を行う場合においても、オーディオ信号をコーディングによって一貫して制御するプロシージャル手法によって行う必要がある。

次にVRコンテンツ化についてである。開発環境として、ゲームエンジンであるUnityを選定し、VRデバイス候補として、PCVRであるOculus Rift DK2とDaydreamプラットフォームであるLenovo Mirage Solo（以後、LMS）を検討した。PSVRは開発環境の問題から候補から当初から除外し、性能と価格、利便性のバランスを考慮した結果、LMSを選定した。次に実装のための手法を述べる

３．プロシージャル生成による視覚表現

３－１　レンダリングパイプライン

従来のUnity4.Xから、Unity5にバージョンアップされ、様々な機能が加えられた。その中でも、ビジュアルイメージングプロセスに大きく寄与する機能として、Command Buffer機能とDeferred Shading（遅延シェーディング）機能が付加された。これらの機能を組み合わせて利用することにより、Unityのレンダリングパイプラインを柔軟に拡張して多彩な表現が可能となる。具体的には次のとおりである。

Command Bufferを用いることによって、Deferred RenderingプロセスでのG-Buffer 内の情報を直接制御することができるようになるため、G-Bufferに対して、Raymarchingなどのポリゴンメッシュベースでなく、距離関数（Distance function）による数理造形の計算結果を出力することが可能となる。

そのことにより、異なる表現方式の形状同士を同じ空間上でレンダリングが可能となるだけでなく、Unityのライティング機能も一貫して行い、１つのスクリーンイメージとして出力させることが可能となるのである。

そこで、これらの特性を最大限に活かせるモチーフとして、前述のとおり今回はフラクタル図像を選定した。

マンデルブロー集合に代表される自己相似形のフラクタルは、古くからコンピュータグラフィックスのベンチマークとして用いられてきた古くて新しいモチーフである。これらは、あくまで数理上の存在であり最終的にそれらを3Dの幾何的なデータ、つまりポリゴンメッシュにラスタライズするとなると、膨大なRAWデータに膨張してしまう。しかし、数理上の式を可視化するだけにとどまると、フラクタル以外のイメージを付加することができない。この制限は、表現としてはかなり単調なものにならざるを得ない。そこで、今回のレンダリングパイプラインを活用することにより、フラクタル図像を実在感のある空間上のオブジェクトとして、リアルタイムに立体的に表現できるだけでなく、標本化された実在感のあるオブジェクトを混在させることが可能となるので視覚表現としては従来にないものを期待できる。

３－２　Raymarchingによるビジュアライズ

Raymarchingとは、視点に入ってくる光の経路を追跡して画像化するレイトレーシングアルゴリズムの一種である。Raymarchingアルゴリズムの概要は、カメラ視点からレイを飛ばし、その始点から進みながらシーン内のオブジェクトとの交点を求めて最短距離を返し、さらに段階的進めて、最短距離が０になると、レイがヒットしたことになる。この０になる点の集合から、オブジェクトの位置や表面といった空間内の状況を算出するというものである。

空間内のオブジェクトは前述した距離関数によって記述される。距離関数を解釈し、それをレンダリングできるプログラム上で表現する場合、例えば、C言語をベースとした高レベルシェーディング言語であるGLSL（OpenGL Shading Language）上で行う場合は、球体は、以下のように数行で記述が可能である。

```
float distanceF_Sphere（ vec3 p）{
 return length（p）-sphereSize;
}
```

そしてレンダリング出力結果は、開発環境やデバイスのスクリーン解像度に最適化された大変滑らかな理想的な球体となる（図２）。

図２　レイマーチングによる球体距離関数のレンダリング結果

一方、ポリゴンメッシュベースで定義し、ラスタライザーによって輪郭表面を滑らかにレンダリングするとなると、比較的単純な球体形状であっても膨大なデータになる。このように、Raymarchingと距離関数によるレンダリング手法は、比較的少ないコードで、フラクタルモデルのようなプロシージャルによる複雑な形状の豊富な画像を作成することができる事に加えて、ポリゴンメッシュベースのラスタライザーと比較して、データサイズをコンパクトにすることが可能である。しかし、汎用的にどんな形状でも生成できるというわけでなく、ポリゴンメッシュほどには自由形状を生成するのは困難である。例えば、特定のアニメキャラクターや、複雑な形状の人工物などである。反面、仮想的な地形や都市形状、単純な植物や生物の形状、数理モデルなど、数学的にパラメトリックに定義できるものは可能である。

そのようなことから、距離関数として記述が可能であるフラクタルモデルは、Unityの新しいパイプライン上のプロシージャルな手法によるVR表現としても最適なモチーフであると言える。

３－３　Unity環境への実装

人間が見たときの美感を有する景勝地を備えたフラクタルの距離関数ファイルは、すでに多くのタイプが存在している。今回は、Hartverdrahtet Projectが、GLSLで書かれたシェーダを即時実行して全世界にブラウザベースで共有できるサービスであるGLSL Sandboxに公開しているフラクタル用距離関数ファイルを基に改造を加えることとした。

UNITYへの実装における全体フローは以下である。

オブジェクトを作成し、それにマテリアルと、スクリプトとアタッチする。マテリアルには、シェーダーファイルをアタッチし、シェーダーファイルは、cgincファイルをインクルードする。

オブジェクトは、描画をするエリアを定義したり実際のモデルであったりUnityにおける処理過程上の上位的な存在である。マテリアルは、オブジェクトの属性として色や形を定義するものである。Ｃ言語によるスクリプトは、Command Bufferや、遅延シェーダーを用いてレンダリングパイプラインを制御し、Raymarchingによる描画を可能にし、その他オブジェクトやマテリアルの属性や変数を制御する重要な制御や処理に関する本体部分である。シェーダーファイルは、cgincファイルを切り替えて呼び出すなどの管理をするものである。

cgincファイルは、距離関数の中身が定義されたファイルである。

このように、Unity上で実装する場合には、各機能やファイルが分化し、相互依存かつ階層的な構造を有している（図３）。

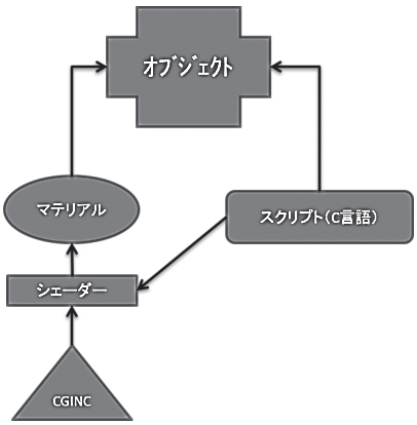

図３　Unityにおける各要素の相関図

次に各フローについて、ボトムアップしながら詳細を述べる。

まず、距離関数ファイルについてである。公開されているGLSLによるフラクタルの距離関数ファイルをそのままUnityへインポートして直接使用することはできない。GLSLをHLSLに変換し、Unityに移植する中間手順が必要となる。

Unityにおけるシェーダ言語は、記述の方法としては、HLSL（High-Level Shading Language）相当である。

wgld.org［２］上にて公開されている円の距離関数のサンプルファイルを参考にした基本的なGLSLファイルの中身を示す。

このコードをコンパイルすると、単純な白色の円が表示される（リスト１）。

リスト１　円を表示する距離関数の処理

```
1    precision mediump float;
2    uniform float time;
3    uniform vec2   mouse;
4    uniform vec2   resolution;
5
6    const float sphereSize = 1.0;
7    float distanceF_Sphere( vec3 p){
8      return length(p)-sphereSize;
9    }
10
```

```
11    void main(void){
12        vec2 p = (gl_FragCoord.xy * 2.0 - resolution) / min(resolution.x, resolution.y);
13
14        vec3 cPos = vec3(0.0,   0.0,   2.0);
15        vec3 cDir = vec3(0.0,   0.0, -1.0);
16        vec3 cUp   = vec3(0.0,   1.0,   0.0);
17        vec3 cSide = cross(cDir, cUp);
18        float targetDepth = 1.0;
19
20        vec3 ray = normalize(cSide * p.x + cUp * p.y + cDir * targetDepth);
21
22        float distance = 0.0;
23        float rLen = 0.0;
24        vec3   rPos = cPos;
25        for(int i = 0; i < 16; i++){
26            distance = distanceF_Sphere(rPos);
27            rLen += distance;
28            rPos = cPos + ray * rLen;
29        }
30
31        if(abs(distance) < 0.001){
32            gl_FragColor = vec4(vec3(1.0), 1.0);
33        }else{
34            gl_FragColor = vec4(vec3(0.0), 1.0);
35        }
36    }
```

uniformが付与されるものは、JavaScriptからの変数である。time、mouse、resolutionは、文字通り、時間、マウス座標、解像度の情報が渡される。

11行目のmain関数がフラグメントシェーダに関する処理部分であり、この箇所をUnityのフラグメントシェーダとして抽出した。

そしてHYPERでんちサイトにて公開されているGLSL/HLSL命令対応表を参照し、以下のように書式を変換した［3］。大きな違いは以下のとおりである（表1）

表1　命令対応の比較（一部）

GLSL	HLSL
Float	float
vec2/vec3/vec4	float2/float3/float4
mat3	float3x3
mat4	float4x4
mat3x4	float3x4

次に、シェーダーファイルを作成した。unity上でのシェーダー言語によるプログラムファイルであるcgincを作成する場合、基本的には各cgincを呼び出して、各種様々な距離関数ファイルによるビジュアルイメージをスイッチするのであるが、今回は視覚的な効果を意図し、動的に変数が時間遷移する機能を加えた。詳細については後段にて詳述する。

次にスクリプトのコーディングである。スケールなどの基本情報や変数を動的に変化させる等の様々な処理をスクリプト内にて一括して行った。

今回は、前述したように、パーリンノイズを用いて動的変化を加えた。なおパーリンノイズの詳細については後述する。

i-saint氏のprimitive: blogのコードを参照してRaymarchingで用いる座標系とUnity のシーンの座標系を一致させた［4］。そのうえで、カメラの方向に対してレイを飛ばして延長する。そして、Raymarchingのルーピング処理において距離関数によって定義され与えられた値に従い、 次々にレイの位置と法線を求めて得られた距離情報をピクセルシェーダーによって画像情報化し、それらをG-Bufferとして出力して格納した。

後段のライティングや、ポスト処理は、Unityの標準パイプライン上で行った。このような過程を経て、ファイナルルックを得た（図4）。

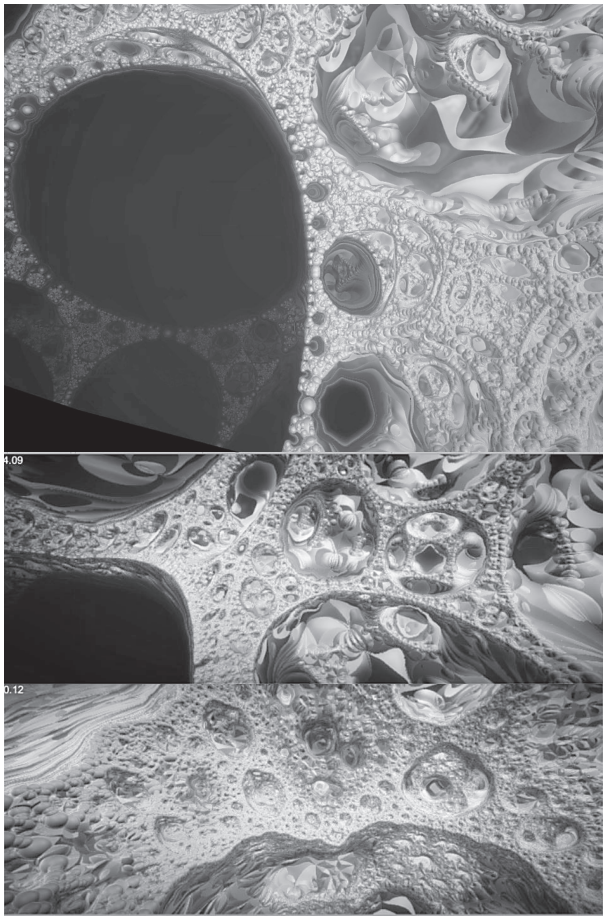

図4　フラクタル距離関数によるレンダリング画像

3－4　Boidsの混在

ポリゴンメッシュベースのオブジェクトも混在できることから、視覚上の演出効果を得るため、アクセントキャラクタとして群れで徘徊するBoidsを配置した（図5）。

Boidsとはオブジェクトの集合体という意味である。1987年にCraig Reynolds氏が発表したアルゴリズムである。このアルゴリズムは、基本的なルールを定義するだけで、生物の群れのような複雑な動きを表現することができる。

今回のプログラムは、TechBloghを参照し、時間遷移して変化する移動パスを生成して、各Boidの行動を組み込んだ［5］。

結果として、最初は散在していたBoidが徐々に集まり、着かず離れずの距離を保ちながら、群化していく様を表現できた（図5）。

今回は衝突判定を行っていないため、フラクタルオブジェクトとは干渉しないため、突き抜ける。しかし、Unityのレンダリングパイプラインに同乗しているため、ライティングやシャドーイングを一貫して適用した。

その結果、距離関数によって生成されたフラクタルオブジェクトにBoidsによる影を落とせたため、空間内の関連性や同一性を強調し、視覚的な演出効果を高めることができた。今回は試行の為、角度や方向が分かりやすいようにボックス形状をダミーとした。

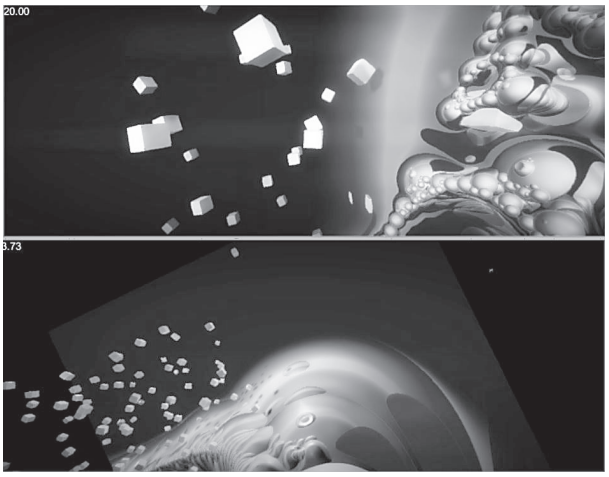

図5　フラクタルオブジェクトに影を落としながら移動するBoids

3－5　最適化処理について

Raymarchingでは、シェーダーに距離関数を記述して、それらをオブジェクト単位で定義するため、オブジェクトの数だけシェーダーが必要になる。プログラムでスイッチすることも可能であるが、今回は、オブジェクトスペース内にレンダリング範囲を限定して描画する方法を以下の理由から採った。

- フルスリーンクアッドスペース全体にレンダリングを行うよりも、計算負荷が低い
- 空間座標を任意に設定できるのでレイを飛ばす開始点が近い
- 数種類の形状を配置できるので管理がしやすい

そこで、オブジェクトスペースとして単純な立方体を選定した（図6、7）。

次にピクセルシェーダーの負荷を低減するために、アダプティブサンプリングというレンダリング解像度を減らしてコンパクト化とオプティマイズ処理を行う方法を採った。まずは低解像度でレンダリングしておき、高解像度でリレンダリングする際、低解像度レンダリングバッファの近傍ピクセルの状態を判定し、それが相似性の閾値以内であれば、その結果を流用し、乖離していれば新規に計算を行って結果を得るというものである。

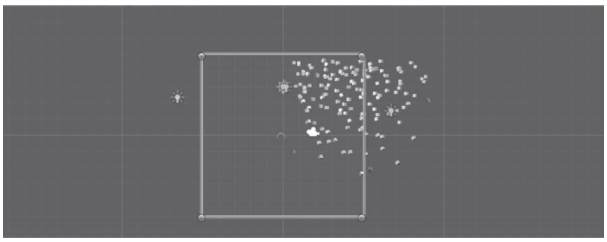

図6　オブジェクトスペース（白色の矩形）とBoids

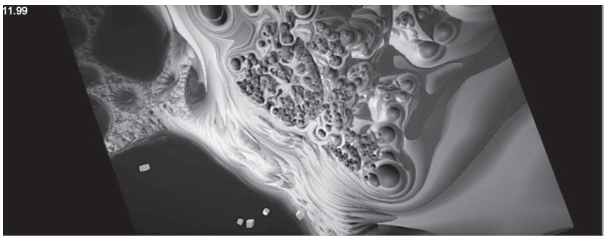

図7　オブジェクトスペースによる計算対象境界

4．プロシージャル生成による聴覚表現

4－1　デジタルコンテンツにおけるオーディオ処理

次に聴覚要素について述べる。アンビエントミュージックや環境音楽といったコンセプトにしたがって、以下の要素や形式を備えた音響トラックを作成することを意図した。

- 12音階といった西洋音楽的な要素を排し、明確なメロディーやリズムを持たない曖昧かつ輪郭のないトラック
- 雑音でも楽音でもない新規的な音色
- 乱数的に連続変化するピッチ感を伴う音程
- フィルターを乱数的に連続変化させて、構成が動的変化する倍音

このような音響トラックを作成するための技術的な前提条件となるのは波形を直接生成し、倍音構成と共振効果を動的に変化させるための一貫したプロシージャルなオーディオ信号制御である。このように、波形そのものをコーディングによって生成し、それらの波形を更にポスト処理を加えて動的に制御するためには、一連のオーディオ処理を一括してプログラム側で行う必要がある。

ゲームやアプリをはじめとする現代のデジタルコンテンツにおけるサウンドは、多くの場合、外部環境、例え

ば音響スタジオでシンセサイザーなどのデバイスで合成編集処理した音をレコーディングし、WAV波形化して、ストレージに格納、プログラムで呼び出してメモリに展開、ストリーミングなどの方法でプレイバックする方式を採用している。

その理由はサウンド作成業務をアウトソースできる、複雑かつ計画的にサウンドを作成できる、プログラムが実行するハードウェア構成に余裕があるなど、経済的、技術的理由によって、こうした方法が主流化している。

かつて、8bitCPUによるコンソールゲーム機器が主流であった1980年代前後においては、サウンドは全てプログラマブルなものであり、プロシージャルな方法によって、動的に生成をしていた。コンソールゲーム機器本体内部に、PSG（Programmable Sound Generator）を備え、それらのチップには、正弦波、矩形波、三角波、雑音などの単純な基本波形が備えられており、専用命令で音色の合成からシーケンスの制御までを行っていた。それらのサウンドは、今現在も、NESやC64などのチップチューンサウンドとして、今現在も聴くことができるが、こうしたプロシージャルなレガシーサウンドは、技術的制約から要請された過渡的な技術と見做され、現在では技術的に積極活用される機会は減じた。

デジタルコンテンツが、携帯デバイスに移行するにつれ、メモリの節約などのコスト低減や、プログラムとの密接なリンケージによる演出上の効果、パイプラインに柔軟性をもたらすものとして、一部のタイトルでは導入される例もあるが、その場合であっても懐古的な趣向を回帰することを意図するものであるに留まっている。

本作が指向する動的にリデザインされ、リレンダリングされるプロシージャルなサウンドは、現代音楽といった領域では主に正弦波やノイズをモチーフとして様々な試みが実験されている例はある。しかし、MAX/MSPやその他のメディアアートに特化した環境における実践が大多数であり、Unityのようなゲームエンジン上、かつ同環境において視覚効果を伴いながら展開する事例は多くはない。

その理由としては、そうした作品の着想や動機がそもそも発現、発露されていないなどの原因は様々考えられるが、技術的な理由として挙げられるのは、従来のUnityにおいてはサウンド関連の機能は基本的に音声再生を主とした機能が中心あったという点である。

本来Unityは、ゲームエンジンであるために、当然ゲームタイトル開発に特化している。前述したように、ゲームタイトルをはじめとする昨今のデジタルコンテンツは、プロシージャルなサウンドを実装するだけの利点や効果が薄い。しかしながら、Unityは3.5以降からOnAudioFilterReadという手続き型オーディオをサポートしている。このことによって、コーディングによって、波形を生成し、そのデータをオーディオソースとして用いることが可能となった。

本来の用途としては、メモリテーブル上に展開する波形ファイルを当該機能によって参照し、その波形ファイルをフィルターに通したり、リバーブやディレイなどの一連の音響エフェクトをポスト処理として加えるものである。文字通り、OnAudioFilterReadである。しかし、技術的には、この仕組みを直接的にオシレーターとして用いることもできる。よって、本作においては、この機能に着目し、プロシージャルなサウンド生成を試みた。

４－２　OnAudioFilterReadを用いたオーディオ信号生成

プロシージャルなサウンド生成が、もたらす大きな可能性の１つは、様々な視覚要素とサウンドと緊密に関連付け、プレイヤーの挙動や空間オブジェクトの変化に同期し、有機的な視聴覚空間を演出・構築することである。

前述したように、プロシージャルなサウンド生成には多くの利点がある。プリレコーディングされた波形データとしてではなくコーディングによって波形を動的に生成するので、メモリの節約に貢献する。サウンドはコンテンツ内におけるどのようなパラメーターにも直接的にリンクされて相互的にワイヤリングされるために、柔軟かつ動的に変化させることが可能である。

しかしながら、欠点も有する。プリレコーディングされたような複雑で自然な音や楽音や楽曲などはコーディングによって直接生成はできない。これは欠点というよりは、トレードオフの関係であり、両方の方法が互いに補間することによってこうした問題は、コンテンツ開発においては解消する。

onAudioFilterReadメソッドは、次のようなフローで実装される。まず、サウンドバッファに何もデータがない状態、つまりエンプティの状態から、サウンドを生成できるようにするために、コーディングによってオーディオデータを生成する。前述したように、本来はメモリ上にあるWAVファイルを参照してバッファを満たすのであるが、その代わりに、サウンドバッファに直接波形データを書き込んでUnityシステムに搭載されているオーディオエンジンに渡し、サウンドカードやDSP上でプレイバックするオーディオストリームとして解釈させるというフローになる。

大きな注意点としては、コーディングによって生成するオーディオデータは可聴域、つまり20Hzから22000Hzの信号として生成しなければならないということである。プリレコーディングされた波形データは、その時点でトリートメントされているが、コーディングによる波形デー

タは、RAWな波形であるために、発振する周波数だけでなく、サンプリング周波数などにも配慮する必要がある。

このような処理のためのUnity上のオペレーションとしては、具体的に次のような手順となる。まず、宣言のようなものとして、１つ以上のAudioListenerをカメラオブジェクトなどにアタッチする。これは、プロジェクト内で、音源から発音される音を聞く存在のようなものと考える。

次いで、１つ以上のAudioSourceをオブジェクトにアタッチする。これは、実際に発音する音源のようなものと考える。キャラクターや、ステージ、あるいはアイテムのようなものがその対象となる。このような関係で、要素が構築されることにより、例えば次のような処理が可能となる。AudioListenerがアタッチされているカメラオブジェクトを、プレイヤーのオブジェクトにリンクし、ブザーの音を発するように設定されたAudioSourceを、ステージ上のブザーオブジェクトにアタッチして配置すれば、プレイモード時に、プレイヤーが、音源であるブザーオブジェクトに接近して遠ざかると、ドップラー効果を伴いながら、音量が距離に比例して増減してプレイヤーにブザー音を聞こえるように出来るのである。このようにして、全体の配置を整えた上で、onAudioFilterReadメソッドを使って、スクリプトにて特定の波形を発振するオシレーターを生成した。

４－３　波形の生成

まずは、試験として440Hzの鋸歯状波を再生するプログラムを作成した。鋸歯状波を選定した理由は、倍音が多く含まれているためフィルター処理効果などを確認しやすいからである。正弦波には倍音が含まれず、基音しかない。その点、鋸歯状波は、振幅は1/2、1/3…と減じる正弦波の合成であり、そのため、倍音構成も、振幅に比例し、基音＋倍音＋２倍音＋３倍音…となる。よって、三角波、鋸歯状波の順に倍音を豊かに含む波形と言えるのである。

デフォルトのサンプリング周波数は、プラットフォームに依存しており、44100や48000の場合もある。もちろん、96000によるオーバーサンプリングなどもプログラム次第では実装可能であるが、ここでは、単純な正弦波であるため、デフォルト設定のままとした。buffersizeのデフォルト値は、2048であるが、同様に可変させることも可能である。コーディングによって生成された信号がバッファに満たされると、データがオーディオデバイスに転送され、DAコンバータによって、スピーカーから音声が再生される。基準周波数として、A（ラ）の音である440Hzを選定した。ラジアンによって、１秒間に440の周波を作り、可聴域にまで到達させた。本作においては、源波形としてパーリンノイズを用いて可聴域にまで増幅した（図８）。

パーリンノイズの詳細については後述する。

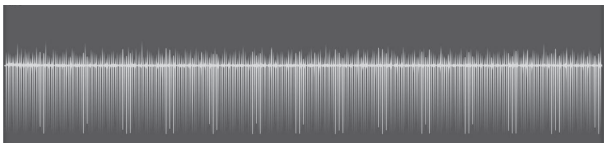

図８　可聴域の帯域まで高周波化されたパーリンノイズ波形

４－４　フィルター処理

第二段階として、フィルター処理を施した。源波形が含む周波数成分を指定した周波数帯域であるカットオフ周波数の高域と低域で濾過して抑制し、倍音の構成を変化させ、音に質感と変化を与える処理である。

ローパス・フィルターはカットオフ周波数よりも高い周波数成分のレベルを下げて低い周波数成分を通すフィルターである。ハイパス・フィルターはその逆効果である。他にも特定の周波数の周辺を通して他の成分をカットするバンドパスフィルタなどがあるが、今回はハイ・ローパスフィルターのみを用いた。

当然ながら、周波数成分を広域から低域まで豊かに含んでいる波形の方が、フィルタリング効果が高い。基音しか持たない純音である正弦波の場合は効果がないが、前述したように鋸歯状波は、倍音を多く含んでいるため、効果が顕著に現れる。更にはノイズは全ての帯域で同質のエネルギーを持っている。ホワイトノイズをローパス・フィルターに通すことにより、ホワイトノイズの音色はカラードノイズといわれるピンクノイズやブラウンノイズのような低音感のある音質に変化する。そのため、顕著にフィルター効果はあるものの、音程感がないために聴感上は魅力的な変化とは言い難い。

そこで、源波形としてパーリンノイズを生成し、実装した。その理由については後述する。

次にフィルター後段の処理としてレゾナンス処理を加えた。レゾナンスは「共鳴」のことである。指定したカットオフ周波数付近のハイもしくはローの帯域の周波数を強調し、倍音の構成やバランスを変化させて、音色に特徴をつける機能である。今回この機能自体はUnityのオーディオコンポーネントとして標準装備されているためコーディングは行わなかったが、フィルター同様に、ゲットコンポーネント命令によりインスペクター上で外部操作によるパラメーター可変動作するように改造を行った。

４－５　LFOモジュレーション

LFOは、「低周波発振器」の意味であり、最低可聴域以下の波形を生成する仕組みである。LFOモジュレーションは、当該波形によって値を変化させ、効果を変調

させるものである。通常は、正弦波や、ランダム効果を狙ってノイズ、あるいは、ノイズを用いたＳ＆Ｈ波形が用いられることが多い［６］。アナログシンセサイザーにおいて、古典的な方法として今日でも用いられる方法であるが、今回はパーリンノイズを用いた。

パーリンノイズの概要や実装については後述するが、当該波形を用いた理由は、不規則かつ有機的な変化の効果を得るためである。

このようにして、いわば減算方式の１DCOのアナログシンセサイザーをUnity上に実装したことになる（リスト２）。パーリンノイズを用いた音響効果に関する詳細と、実際に作成した波形の評価については後述する。

リスト２　可聴域の鋸歯状波を出力する処理

```
public class SawtoothWave_DCO :
MonoBehaviour{

        const double frequency =
        440;
        public double gain = 0.05;
        private double increment;
        private double
        time;
        const double sampling_frequency =
        44100;
        private float
        noise;

        void Update () {
                noise = Mathf.PerlinNoise(Time.time, Time.time);
                gameObject.GetComponent<AudioLowPassFilter>().cutoffFrequency =
                noise*7000;
        }

void OnAudioFilterRead(float[] data, int
channels) {
                increment = frequency * 2 * Mathf.PI / sampling_frequency;
        for (var i = 0; i < data.Length; i = i + channels)
        {
                time = time + increment;

                        data [i] = (float)(gain*((time + Mathf.PI) % Mathf.PI*2) / Mathf.PI - 1.0);
                if (channels == 2)
                        data [i + 1] = data [i];
                        if (time > 2 * Mathf.PI)
                        time = 0;
        }
}
```

５．パーリンノイズの積極的活用

５－１　パーリンノイズについて

ホワイトノイズなどに由来する乱数値は、完全に不規則であるため、隣接する変化の度合いに連続性がない。局所的に補間処理を加えたとしても、全体としては、離散的な出力となる。そのため、自然で滑らかな値の列の出力結果を得たい場合には適切ではない。そこで、生成結果が自然であるパーリンノイズを選定して用いた。

パーリンノイズは、疑似乱数アルゴリズムの一種である。不規則性がありながらも、フラクタルの性質を持ち、隣接する変化の度合いが連続的で自然な連続性を持った値を得ることができる。

本来は、コンピュータグラフィックス分野において大理石や木目、地表面などの自然造形物のテクスチャ作成法として、パーリン氏によって1985年にSIGGRAPHで発表された技術である（図９）。

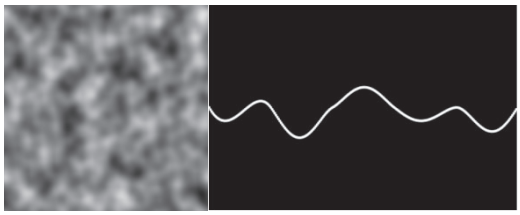

図９　パーリンノイズによって生成されたテクスチャと波形

本作においては、これを以下の２点に用いた

- フラクタルの形状を変化させる変数
- オシレータに読み込む波形

通常、こうした変調処理であるモジュレーション機能は、低周波発振器、いわゆるLFO（Low Frequency Oscillator）を用いる事が多い［６］。その波形としては、用途によって様々あるが、正弦波や雑音をその変化量として取得して用いる場合が多いが、本作では、パーリンノイズをLFOとして用いることを試みた。

５－２　パーリンノイズによる動的形状変化

パーリンノイズには、改造版や改良版など様々な実装方法が公開されているが、Unityには標準でパーリンノイズを発生させるための
Mathf.PerlinNoise（）;関数が備えられている。

今回は、この関数機能を用いたが、目的に応じた期待結果を得るには、多少の修正や改修が必要となる。

本作では、フラクタルの形状を変化させる変数を外部から制御できるように独自の関数を定義した。

シェーダー内のmap関数内に、本来ならばcgincで定義されている距離関数の内容を直接記述し、インクルードしなくてすむようにした上で、独自にset関数を定義した。

このset関数は、フラクタル図形を定義する距離関数内にある処理対象となるフラグメントの座標位置の値を動的に変化させるものである。

通常、最も見映えのよい景勝地点として、この値は決め打ちとして固定化されている。この値を、外部から動的変化させることにより、プロシージャル生成による視覚表現の可能性を探求するという意図である。

このset関数経由によってスクリプトから値をシェーダーに渡し、Inspector上の値をマウスで変更することで、距離関数の値を動的に変化させることや、関数式によって時間遷移によって変化させることができる。そのスクリプトの一部を抜粋する（リスト３）。

リスト３　動的変化をさせるための処理

```
float map(float3 p)
{
        float3 CSize =
        float3(0.92436,0.90756,0.92436);
        float Size = 1.0;
        float3 C = float3(0.0,0.0,0.0);
        float DEfactor=1.0;
        float3 Offset = float3(0,0,0);
        float3 ap=p+1.;
        for(int i=0;i<10 ;i++){
        ap=p;
        p=2.*clamp(p, -CSize, CSize)-p;
        float r2 = dot(p,p);
        float k = max(Size/r2,1.);
        p *= k;
        DEfactor *= k + 0.05;
        p += C+_set;
        }
```

リスト４　パーリンノイズによるLFO変調処理

```
        void Update(){

        m_material.SetFloat("_set",m_set);

        m_set =
Mathf.PerlinNoise(0,Time.time/100);
        m_set = m_set * 15-5;

        GetComponent<Renderer>().material.SetFl
oat("_set",m_set);
```

５－３　パーリンノイズによる形状のLFO

次に変化量を動的変動させる変調方法として、同様にパーリンノイズを用いた。

前述したset関数にパーリンノイズが生成するノイズ値を、どのタイミングで取得するかtimeで指定し、その値に対して、調整を施した。

Mathf.PerlinNoise関数が、Unity実装の他の関数同様、０～１までの値をとるため、５をoffset値として設定した。５という数値は試行錯誤の上で求めた結果である。

値を変化させて、最も変化の度合いが、表象表現として主観的に面白く見える値として選定した。

加えて、このoffset値をさらに変数化し、パーリンノイズの値を入れて入れ子のような動的変化を試みたが、変化の度合いと複雑に変化するフラクタル図形の親和性を得ることができなかった。つまり、映像として混雑かつ混沌な印象になってしまったのである。

よって、模様の変化は複雑にし、変化の動きは有機的な変動を持ちながらも推移は単純にすることとした。疑似乱数であるために、周期性や繰り返し感が感じることなく変化の勾配に連続性があるために自然な挙動となった（リスト４）。

このことにより、距離関数によって描画されたフラクタルオブジェクトが、有機的な物体が息づいているかのように見映えを変化させることができた（図10）。

結果的にパーリンノイズをLFOとして用いるという今回の処理は期待した効果を得ることができた。

図10　変調処理に伴い形状が動的変化するフラクタル画像

５－４　パーリンノイズの先行応用例

シンセサイズのためのオシレーター源波形としてホワイトノイズノイズジェネレーターをアナログ回路で組む場合は、ノイズ源とし、定電圧ダイオードやトランジスタを用いることが一般的な基本の方法である。これらの電子部品に一定以上の逆電圧をかけたときに発生するノイズを広帯域アンプで増幅して、ノイズを信号として取得する。

実際のシンセサイザーに搭載されるノイズジェネレーターは、よりコンパクトなものであったり、近年ではソフトウェアで生成したノイズ波形をアナログ回路に送って用いる機器もあるが、いずれにせよホワイトノイズジェネレーターは比較的簡便にアナログ回路上にて実装出来得るものである。

対して、パーリンノイズは、そのアルゴリズムの特性上、アナログ回路で組むことが難しい。ホワイトノイズは、実際に発生している物理現象を信号として直接利用しているが、パーリンノイズは、そのような方法で信号を得る手段はない。よって、従来型の減算方式のアナログシンセサイザーに搭載されることはなかった。そして、CPUを搭載する機器が一般的になり、組み込み技術が進んだ現在において、パーリンノイズジェネレータをソフトウェアで実装することは、比較的容易であるにもかかわらず、市場で流通する一般的なDTMソフトや、VST音源、ハードウェアシンセサイザーに、パーリンノイズジェネレータが搭載されている例はほとんどみられない。

例外的に、sonic-potions社のMalaclypseというユーロラック規格のパーリンノイズライクなフラクタルノイズモジュールと、Andesと呼ばれるPerlinオシレータや、LFOにパーリンノイズに基づく機能を持ったVST音源などが挙げられるのみである。

Malaclypseは、基本的には変調器である。LFOの波形として、従来型の正弦波や、矩形波、S&Hに加え、フラクタルノイズ波形を持っている。そのため、音程やフィルターに、フラクタルノイズ波形を信号として割り当てると、自然ながらも一定の変化に富む有機的な結果を得ることができる。

そして、サイクル速度を可聴域にまで上げることによって、ノイズオシレータとしても用いることができる。

ユーロ規格のアナログモジュールではあるが、基板の大きさや、機能内容から推測すると、内部信号の生成にはデジタル回路が使用されていると思われる。

Andesは、VST音源であるため、完全なデジタルソフトウェアシンセサイザーである。非整数ブラウン運動（fBM）の方法を用いて、パーリンノイズを解像度の高い緻密なフラクタルノイズとして生成し、それを源波形としてオシレーションをしている。

本作の着想は、Andesに近いものであるが、Andesがあくまでを音源として音程感のある楽音としてパーリンノイズを用いるのに対し、本作はあくまでパーリンノイズが持つノイズ性を表現として用いる事が相違点である。

５－５　パーリンノイズによるDCO

パーリンノイズを源波形として生成し、オーディオ信号化するスクリプトを作成した（リスト５）。

パーリンノイズは様々な帯域のエネルギーを満遍なく、自然な勾配をもって分布しているため、三角波や、鋸歯状波にさらに高周波成分を足してメタライズ化したようなジリジリ、ギリギリとした尖りと、正弦波のようなしっかりとした音程感が併存する不思議な聴感上の印象となった。

しかしながら、元がノイズであるために、楽音というよりは、やはり雑音であり、パーリンノイズを源波形として、様々な変調や効果を加えてシンセサイズ処理を行ったとしても一般的に用いられるような楽音的な音色になるかと言えば、それは難しいと言えるだろう。換言すれば今まで聞いたことのない既知感の希薄な新しい音を模索するうえでは、可能性が開かれているとも言える。

そこで、オシレーションする音源を二つに増やし、それぞれの基本周波数を262Hzと392Hzに設定した。つまり、Cメジャーのドとソであり、周波数比は２：３の完全五度の響きである。

この関係をオフセットの値にして、基本周波数とフィルターのフレケンシーの値をさらにパーリンノイズ波形をLFOとして変調させた。

こうしたポスト処理によって、幻想的な雰囲気を持つ音響空間を、コンパクトなコーディングで得ることができた。

リスト５　パーリンノイズによる発振処理

```
1 public class perlin_noise_DCO : MonoBehaviour
2{
3       private System.Random RandomNumber = new System.Random();
4public float offset =
0;
5       private AudioLowPassFilter filter;
6       void Update () {
7               filter = this.gameObject.GetComponent<AudioLowPassFilter>();
8               filter.cutoffFrequency = 1500f*Mathf.Sin(Time.frameCount/30f)* 1000f;
9       }
10
11      void OnAudioFilterRead(float[] data, int channels)
12      {
13              for (int i = 0; i < data.Length;
                i++)
14              {
15                      data[i] = offset *1.0f + Mathf.PerlinNoise(0, i*1000000000000) *2f;
16
17              }
18      }
19}
```

５－６　評価

次にポスト処理による効果の分析を行った。まず、パーリンノイズ源波形にさらにパーリンノイズによるLFO変調を加えたものをFFT処理にて比較したところ、パワースペクトルの構成が変化し、聴感上の耳あたりが改善され、過剰なメタリック感が低減した（図11、12）。そして、不規則かつ自然に倍音構成が変化し、不規則でありながら、自然な揺らぎのような一定の

規則ともいえるような1/f揺らぎ成分が顕著に増加した（図13）。

このようにポスト処理効果によって源波形の質感が変化し、揺らぎ成分が増聴感上増加し、鑑賞者が何かの効果を持った音響として認識できる表現効果があることが検証できた。

今回は、パーリンノイズをオシレーターの源波形として用いるという実験に留まり、楽音としての探求に至っていないが、新規性のある音響効果としては、意図した結果となった。

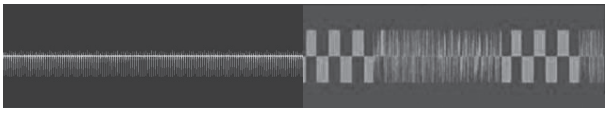

図11　処理前の波形（左）と処理後の波形（右）

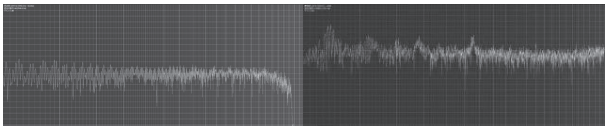

図12　処理前のスペクトル（左）と処理後のスペクトル（右）

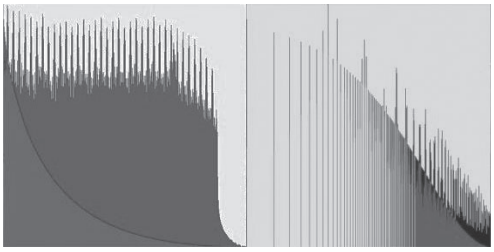

図13　処理前の1/f成分（左）と処理後の1/f成分（右）

6．VRデバイスへの実装

6－1　VRデバイスの選定

VRデバイスの実装については、コンソールゲーム機のPlayStation上で動作するPSVRは開発環境の問題から候補から除外し、PC環境上で動作するOculus Rift DK2、ハイエンドな高機能スマートフォンや、それらと同等の性能を有するCPUなどを搭載した環境で動作するプラットフォームのDaydreamを候補として検討した。他にも独自のプラットフォームで動作するOculus GOや、その他海外でリリースされているVRデバイスもあるが、今回のコンセプトを満たすには、以下の要件を出来るだけ備えているデバイスでなければならない。

- プログラムをコンパクトに最適化せずに冗長性の高い状態でビルドし、安定動作させるためにメインメモリが32GB以上であること
- 今回用いるRaymarchingによる距離関数ファイルのレンダリング処理の計算負荷は非常に高いため、遅延を生じさせないように最低でも2GHz以上のプロセッサーであること
- 自由な動作を可能にし、空間や事物の存在感を向上させるため、VRデバイスのローカル座標だけなく、VRデバイス自体のグローバル座標を取得できる6DOF（six degrees of freedom）ポジショントラッキングセンサーを搭載していること
- 使用する場所の制限をなくし、運用を簡便にするためにケーブルレスであること

以上の条件をバランスよく満たすVRデバイスとして、2018年5月時点において、前述したDaydreamプラットフォームで動作するLMSを採用した（図14、表2）。

図14　Lenovo Mirage Soloの装着図と分解図

表2　LMS製品仕様一覧表

プロセッサー		Qualcomm® APQ8098オクタコア プロセッサー（最大2.45GHz）
OS		Daydream 2.0
メインメモリー		メインメモリー4GBフラッシュメモリー64GB
表示機能	ディスプレイ	※ 5.5型IPSパネル（2560x1440ドット）
	レンズ	非球面フレネルレンズ、110° FOV
センサー類		WorldSense™対応デュアル・モーション・トラッキング・カメラ、ジャイロセンサー、加速度センサー、電子コンパス、近接センサー

6－2　開発環境

開発環境は、以下のソフトウェアとアドイン機能によって構築した。

- Unity：2018.1.4
- GoogleVRForUnity_1.130.1
- JDK：jdk-8u172-windows-x64.exe
- AndroidSDK：Android 8.1

安定的にビルドを行うためには、上記環境の組み合わせが重要である。JDKやASDK側にリビジョンの相性があるために構成が異なるとエラーが頻出することになる。今回は、複数回の試行錯誤を経ることによって、安定的

に動作する実行ファイルをビルドすることが出来たため、その実行ファイルをUSB Type-Cと接続したVRデバイスとPC間で転送し、VRデバイスのメインメニュー上に登録した。そこから先は、通常のアプリとして起動させることができる。

６－３　動作の調整

Raymarchingによるレンダリングは、基本的に高負荷な類の処理である。よって、前述したようにオブジェクトスペースを用いて描画空間を制限しているが、その他にもライティングやシャドーイング、後処理効果など、見映えを向上させる付加的な処理がさらに負荷をかけてしまう。そのような複雑な動作が何重にもスタックされると、非線形に遅延が生じてしまう。フレームレートを超える速度で即座に視点移動すると、視界の描画に遅延が生じ、未描画の黒い領域が視野の両端に視認できてしまう。PCVRの場合は、GPUのアクセラレーション機能をもったグラフィックカードを本体に増設することによって、物理的に解消できるが、今回の環境ではそのような方法を採ることができないため、VRデバイスから逐次転送し、負荷と見映えのバランスを手動で調整した。

６－４　VR空間内の視点

前述したUnityの開発環境下においては、仮想的に画面上にVR画面を左右ステレオ表示させることが可能である。そこで、仮想カメラのキャリブレーションやチューニングを行って調整を行った（図15）。

１mのものが１mの“見え”で、感覚的にとらえられるように、ステレオベースの距離等を微調整しながら、最適な視差と視点位置を割り出した。LMSは、視野角であるFOVが110度である。人間の視野角よりは狭いが、既存デバイスの中では最も広い。

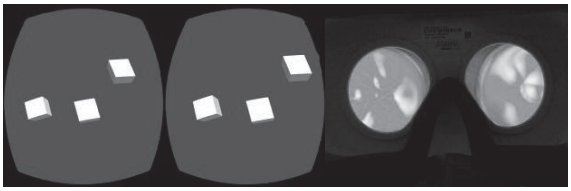

図15　開発用仮想画面（左）とVRデバイス上の画面（右）

６－５　VR空間内の移動

大きな利点である6DOFポジショントラッキングを活用するために、自分自身が移動することによって空間内を文字通り移動できるように設定した。開発者モードからSafety areaの移動制限を解除し、Safety areaが無効化されると、障害物がない限り、プレイヤーは無制限に移動ができる。

通常のコンテンツでは、ポリゴンメッシュベースのアセットであるために、一定の距離を移動し、そのアセットの最大値を超えてしまえば、その先には何も表示されない。しかし、本作はプロシージャルによって空間が生成されているために、行き止まりが生じることなく無制限に空間移動ができる。これはプロシージャル方式の最大の特徴である。しかしながら、こうしたVR空間での移動は、いわゆる3D酔いを誘発しやすい［７］。

そこで没入感や現実感は、やや低減するもののユーザービリティの観点から、Daydrame規格のコントローラーを用いて、テレポート移動できるようにプログラムを加えた。テレポート移動とは、コントローラーを傾けることによって、空間内に仮想の放物線を描き、目的地を指定して、そこへ直接移動して空間移動すべき距離をショートカットする空間移動方式である。画面の上に横切る三角屋根型がポインターであり、円形の枠が、移動指定された目的点である。

６－６　VR空間内の衝突判定

Raymarchingによるレンダリングは、あくまで“見え”だけを画像として出力しているため、ボリュームなどはもたない。いわば張りぼてであるために、視点が境界内以内に入ると距離関数によって生成された空間を突き抜けてしまう。スタティックで変化しない静的な形状であれば、距離関数によるオブジェクト形状とUnityが標準装備する衝突検知のためのプロキシオブジェクトであるコライダーなどと一致させ、衝突判定を行えるが、今回はダイナミックに動的変化するために、この方法は今回には適さないと判断し、衝突判定の実装を断念した。考えられる解決方法としては、同じ距離関数で、衝突検知のためのプロキシオブジェクトとしてUniry上の標準制御パイプラインの経路上に組み込むことが考えられるが、実装には困難が伴うと予想される。Raymarchingによるレンダリング画像処理の場合は、フラグメントシェーダーに色情報を渡す形で視覚化されていたが、衝突検知となると、当然ながらXYZのデカルト空間内に、離散的な解像度を持った何らかの幾何形状的なものに、それらを変換して再定義する必要がある。仮にそのような情報を生成することができたとしても、それを距離関数によって定義されたオブジェクトと同期連動させて並行処理を行うには、現状のUnityパイプラインでは仕様上、不可能であると考えられる。よって、衝突検知機能は今後の課題としたい。

６－７　処理パフォーマンスの調整

Unityには、プログラムの実行時における負荷状態を監視する機能であるStatistics（Stats）とProfilerが搭載されている。今回のプロジェクトの負荷状態を、まず

はStatisticsで確認したところ、１秒間に描画しているフレームが18.5、全体的なDrawCall数が53、マテリアル処理に対してのDrawCallであるSetPass callsが53、影の処理に対してのDrawCallであるShadow castersが313という結果であった（図16）。

DrawCallとは、１フレーム描画するのに、CPUからGPUに対して何回描画命令を送っているかを示す値であり、この数が少ないほど負荷が低く、高い描画レートを維持できているということである。標準的な描画レートとして30FPS、DrawCall数100がUnity上での目安となっている。

今回の評価としては、基本的な負荷としては標準以内であるが、影の処理に３倍以上の高負荷がかかっていることが判明した。影の処理を省略すれば、負荷は低減するが、見映えは低品質にならざるを得ない。

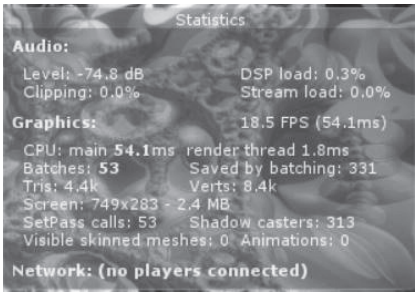

図16　試作作品プロジェクトのStats表示画面

そこで、Profilerを使って、フレーム毎の処理を時系列で再検証を行った。負荷がスパイク上になっていることから、フラクタル距離関数の値が動的変化する際に、複雑な結果が出現する頻度に応じて高負荷であることがわかった（図17）。

複雑なプログラムではないので、ボトルネックやタスクマネジメントに由来する原因はないと考えられる。VRの場合は、左右２画面分計算して出力することになるが、単純に２倍とはならない。ステレオ出力自体はフレーム毎の負荷よりも、ヘッドセットの動きに追従し、円滑に画像をレンダリングさせることが、体験要素としては重要である。

そこで、今回は負荷低減のために、計算すべきレンダリング解像度を大幅に下げて、補間処理によって2560x1440ドットの5.5型IPSパネルの解像度を満たした。

フラクタル図像の場合、自己相似形ということもあり、画面に占める類似部分が多いため、前述したオーバーサンプリングによる計算のコストダウンに適している。若干、ピントの甘い写真用語でいうところの「眠い画質」になってしまったが、そもそもそのような曖昧なイメージが効果として作用しているため、むしろポスト処理を1段減らすことにもなり、二重にコンパクト化に寄与するとも言える。その結果、遅延のない描画速度を維持することができた。

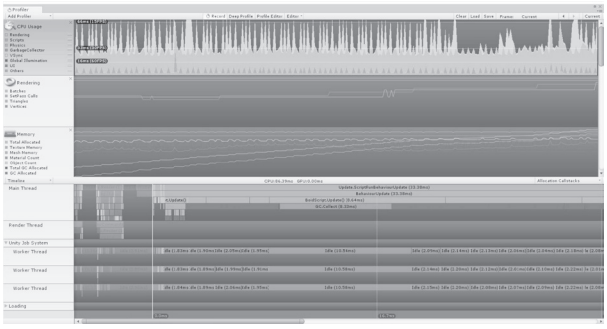

図17　試作作品プロジェクトのProfiler画面

７．まとめと今後の課題

今回の試作を通して得られたものが２点ある。

１点目は、視聴覚情報を生成して出力出来るフローを構築しプロシージャルの可能性を提示できたことである。ただし、前例のない実験作であるため、冗長性だけが高くなり、ゲームなどの他分野の類型コンテンツに比べ、パブリック向けタイトルとしての訴求力を有するところまでは至っていない。

しかし、今回の試作成果を既存のジャンルに適応させるとした場合、例えば、メディテーション、ヒーリングといった、かつてのビデオドラッグや、瞑想ガジェットのような位置づけにアレンジすることも可能であろう。そのような点では、単なる実験にとどまらずパブリック向けタイトルとして萌芽的な可能性を潜在的に有していると言える。

２点目は、実行デバイスとしてVRを用いて未知の存在感を鑑賞者に現前化させることが、一つの表現として成立することが確認できたことである。例えば、近年話題のチームラボによる一連のニューメディアアート作品群も、そうした効果を自覚的に援用し、制作されていると思われる。これらの作品群はVRではないが、壁や床一面にプロジェクションされたUnityベースのセミインタラクティブCG映像は、来場者を圧倒する迫力がある。しかも、物理空間であるから、その体験は共有することが可能である。

VRシステムは当然のことながら、そのデバイスの特性上、どうしても空間共有による共同性や協同性に限界がある。しかし、Facebook社が、Oculusデバイスをプラットフォームにした Facebook Spacesというバーチャルコミュニケーションスペースを構築しようと企図している。

仮想空間による体験共有が可能になれば、「そこに一緒にいるだけで楽しめる」という新しいコンテンツ活用スタイルが促進され、新しいジャンルの契機が生まれる

ことが考えらえる。そのような条件が揃ったとき、今回の試作の成果が適用できることを期待したい。

そして、今回において試作した知見を用いてSTYLY上にてVRコンテンツを制作した。

STYLYとは、VR空間を駆使した新たな表現・体験を生み出せるクリエイティブプラットフォームを謳ったWeb上で動作するVRコンテンツのオーサリング環境である。STYLYは、ショップ空間やインスタレーション、ギャラリーなど現実世界の拡張空間としてのVR展開を想定しているが、このSTYLY上にて今回の試作で試行した実験を展開してみた。

STYLYは、視聴覚情報の両方ともプロシージャル生成による逐次レンダリング機能は有していない。よって外部DCCTなどを用いて制作したアセットを読み込んで配置する。よって、今回の試作における距離関数のオブジェクトは転用することは全くできないが、プロシージャル生成のアルゴリズムによって形状化したモデルをポリゴンメッシュに変換して、STYLYに読み込んだ。同様にして音声ファイルも、スタティックなファイルへとベイクしてSTYLYのアセットとして読み込んだ。

加えて、アクセントオブジェクトとしてBoidsを読み込もうとしたところ、原因不明のエラーが頻出したため、これを断念し、フォトグラメトリによる人間オブジェクトを数体配置した［8］。そして、アンビエントなサイクリック音が、延々と流れる中、極彩色に彩られた仮想的な聖堂空間の中に、何かに見立てることはできるが実在する何かではないという抽象的なVR空間を構築した（図18）。これらの発想や指向性は、今回の試作で試みようとしたことである。

このようにして当初コンセプトを反映し制作した作品を、PARCO、STYLY、loftworkの三者が主催するNEWVIEW2018コンテストに応募した。その結果、世界7ヶ国、応募総数219作品の中から一次審査を通過したファイナリスト19作品ファイナリストの一つとして選出された［9］。今回のシステム成果の直接応用ではないが、VRコンテンツにおける作品指向性と、そのような作品における客観的評価を確認できた。

今後の課題としては、試作作品におけるプログラムの調整と定量評価の実施である。今回のプログラムでは多くのドローコールが発生し、即応性が低くなってしまい、一部では遅延も生じた。プログラムのチューニングが必要なだけでなく、まだ表現として模索状態であるので、形式が確立されておらず、コンテンツとしての作り込みが足りないことが反省点である。そして、被験者による心理的な定量評価を今回は行えていない。ＳＤ法尺度等の印象評価手法による測定結果に基づき、より客観的な評価を得たいと考えている。

プログラムの調整不足による高負荷は、VRデバイスの更なる性能向上で解消できる余地もあるため、それ以外ついて今後の課題として積極的に取り組んできたい。

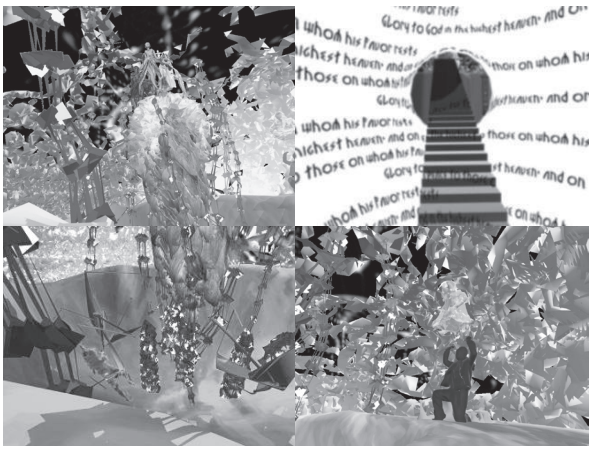

図18　COMPRESSION ARTIFACT　-LK.2:14-

参考文献

［1］熊谷武洋，“フラクタルを芸術作品に応用する手法について ―心象風景としてのフラクタル図像”，山口大学教育学部論叢，第55巻第３部pp.105–118,2005.

［2］wgld.org，https://wgld.org/d/glsl/g009.html

［3］HYPERでんち，https://dench.flatlib.jp/opengl/glsl_hlsl

［4］i-saint　primitive: blog，http://i-saint.hatenablog.com/

［5］TechBlogh,http://developer.wonderpla.net/entry/blog/engineer/Unity_BoidsSimulation/

［6］熊谷武洋，“アナログシンセサイザーの変調機能によるリズミックフレーズ音の合成手法について一低周波発振器を活用した事例分析一”，山口大学教育学部研究論叢、第62巻第３部pp.91–103,2013.

［7］西川善司，古林克臣，野生の男，izm，比留間和也，“VRコンテンツ開発ガイド 2017”，エムディエヌコーポレーション，2017.

［8］熊谷武洋，“アート系コンテンツ制作のための3Dスキャナー活用方法の研究”，山口大学教育学部研究論叢、第68巻第３部2017

［9］Newview design awards2018，https://newview.design/awards/2018/en