〔研究ノート〕

Jetson TK1を用いた物理レンダリング

Physical Rendering by using Jetson TK1

箕　原　辰　夫

1．はじめに

3次元CGにおける物理レンダリングという2次元画像生成技術は，レイ・トレーシングなどの技術から始められたもので，現実世界の実写とほぼ同等の品質を持つ画面を生成することを目的としている。そのために，画面の1画素ごとに，光源まで光線を追う必要がある。加えて，画面に映されるカメラと光源までの間の，物質の反射や屈折などの物理的な光に対する現象も反映させる必要がある。また，ラジオシティなどの間接光などの影響を受けた大域照明技術や煙や水蒸気のような微細粒子状の物質によって引き起こされる光学現象などを計算し，シャボン玉やガラスなどによって生成される集光模様（caustic）を再現することが可能となるフォトンマッピングなどの技術もその中に含まれる。

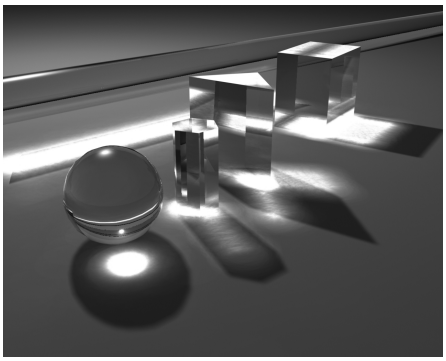

図1　フォトンマッピングによる集光模様（caustic）[(1)]

3次元CGにおける物理レンダリングは，Unreal[(2)]やUnity[(3)]といったゲーム開発環境にも積極的に採り入れられている。ただし，今までのゲームの実行環境は，OpenGL[(4)]やDirect3D[(5)]といったシェーダー（shader）を基本としたレンダリングが中心であり，物理レンダリングが行なう光の屈折や反射などの投影は，飽くまでも追加の機能であり，疑似

(1)　http://img05.deviantart.net/a1be/i/2002/26/3/4/caustics.jpg
(2)　https://www.unrealengine.com/ja/
(3)　https://unity3d.com/jp/
(4)　https://www.opengl.org
(5)　https://msdn.microsoft.com/ja-jp/library/cc324500.aspx

的に実現している状況である。これは，ゲームの実行環境が，実時間で30fps（1秒間に30画面・画面はフレームとも呼ばれる）の高速レンダリングを必要とするからであり，光の複雑な挙動の投影を緻密に計算する物理レンダリングをしていたのでは，30fpsの性能を出すことができないからである。そのため，ある程度の軽い物理レンダリングを行ない，シェーダーの作り出す画面に追加としてエフェクトを掛けて実現している。アニメーションとして人間が認識できるのは，12fpsぐらいからであり，画面のすべての画素に対して物理レンダリングをしていたのでは，12fpsさえも下回る状況がある。しかしながら，このところのCPUおよびGPUの性能向上によって，ゲーム開発環境においても，ある程度の物理レンダリングを行なうことが可能となってきた。そのため，GPGPU（General Purpose Graphic Processing Unit）として，それまでのOpenGLやDirect3Dなどのシェーダー用のGPUの計算パイプライン機構を用いて，レイ・トレーシングや物理計算のための補助計算をさせるようになってきている。これは，Compute Shader（あるいはComputed Shader）と呼ばれて，Unreal，Unity，あるいはCryEngine[6]といったゲームエンジンでも利用可能になってきている。

図2　Unreal 4における環境光反射エフェクト[7]

本研究では，NVIDIA社がリリースしたボード型コンピュータJetson TK1[8]を複数台用いて物理レンダリングを分散させて行なうことにより，より高速な物理レンダリングを行なわせることを試みるものである。これまでも，3次元モデリングソフトウェアでは，1台では非力であったために，複数のコンピュータに分散させてレンダリングを行なうことによって，高速にレイ・トレーシングを基本とした物理レンダリングを行なわせてきた。それでも，1秒間に30画面のような性能は得られてこなかった。しかしながら，Jetson TK1は，1台にCPUコアを4つもつコンピュータであり，強力なGPUも付随している。これらを用いれば，以前は不可能であった30fps以上の性能を持つ物理レンダリングが可能であると考えられる。

(6)　https://www.cryengine.com
(7)　https://docs.unrealengine.com/latest/images/Resources/Showcases/Reflections/Subway.jpg
(8)　http://www.nvidia.co.jp/object/jetson-tk1-embedded-dev-kit-jp.html

２．システム構成

１つのJetson TK1のTerga K1プロセッサ[9]は以下のような仕様になっている。CUDA[10]は，NIVIDA社のGPUのアーキテクチャや開発言語のことを指している。

NVIDIA Terga K1 SoCプロセッサ
・NVIDIA Kepler GPU[11]：192 CUDA Coresを持つ
・NVIDIA ARM Cortex-A15[12] CPU：４つのARM11[13] Coreを持つ32bit CPU

４つのCPUコアおよびGPUが共有メモリで連携しているJetson TK1を９台連携させることにより，８あるいは９分割して物理レンダリングを行なう。９台の通信には，1GBit/10GBitのEthernetおよびスイッチングHUBを用いて，通信遅延が少なくなるようにする。この構成は，一般には，NUMA（Non Uniform Memory Architecture）と呼ばれており，スーパーコンピュータでも複数のCPUを連携させる基本的な構成法になっている。

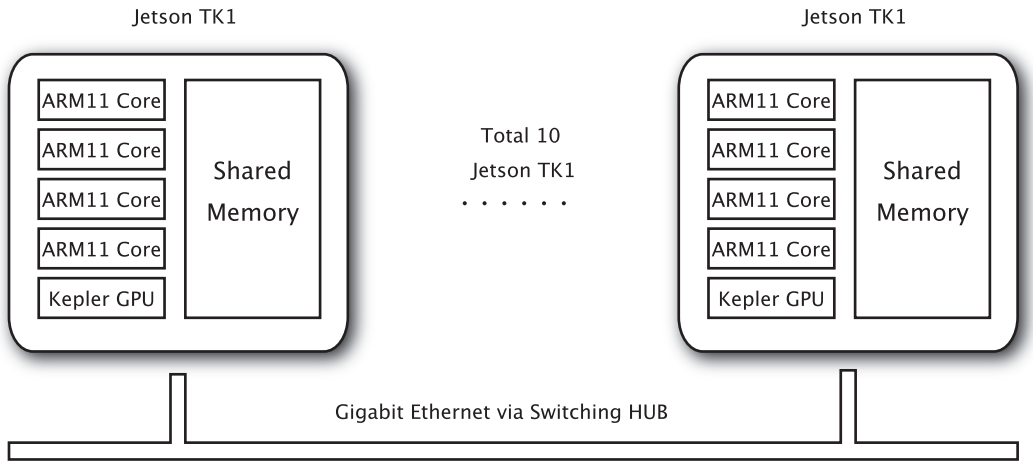

図３　システム構成

(9)　http://www.nvidia.co.jp/object/tegra-k1-processor-jp.html
(10) http://www.nvidia.co.jp/object/cuda-parallel-computing-platform-jp.html
(11) http://www.nvidia.co.jp/object/nvidia-kepler-jp.html
(12) https://www.arm.com/ja/products/processors/cortex-a/cortex-a15.php
(13) https://www.arm.com/ja/products/processors/classic/arm11/index.php

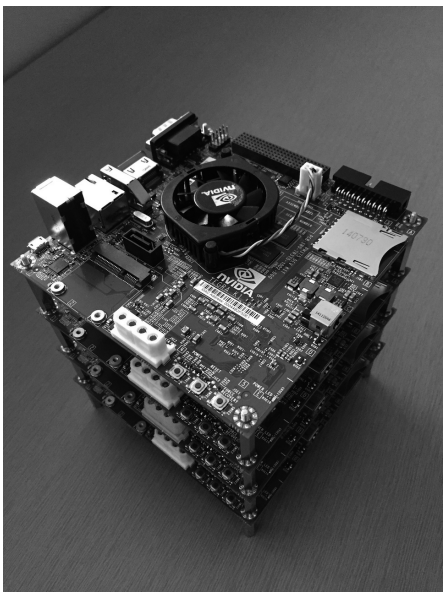

図4　4枚のJetson TK1をクラスタリングしたもの

それぞれのJetson TK1では，オペレーティングシステムとして次のようにLinux Ubuntu[14]が稼働している。開発環境なども合わせて記述すると以下のようになる。

・Operating System: Linux Ubuntu L4T R19 ARM版
・Python: Python 3.3 ARM版
・CUDA SDK: CUDA 6.0 Toolkit for L4T Rel-19.2 ARM版

2017年現在は，それぞれの版元では，もう少しアップグレードしている（L4T R19がR21になっており，Python[15]は3.4版を使うほうが良い，またCUDA Toolkit[16]はR21上では6.5版が稼働している）が，実際にJetson TK1上でアップグレードを掛けるとハングアップしてしまうので，Jetson TK1がリリースされた2014年当時の環境での構築をする必要に迫られた。

3．物理レンダリングのためのライブラリ

本稿では，物理レンダリングを行なうライブラリとして，いくつかのフリーのライブラリを用いて，その性能などについて比較検討する。最終的な研究では，ライブラリを1つ選び，スクリプト言語Pythonによって開発を行ない，Pythonから，ネットワークで繋がった複数のサーバでレンダリングを行なわせ，1秒間に複数のフレームをレンダリングさせ

（14）https://www.ubuntu.com
（15）https://www.python.org
（16）https://developer.nvidia.com/cuda-toolkit

るように構築していく。ライブラリの中には，ネットワーク経由のレンダリングを可能とするものもある。また，Jetson TK1の持っているKepler GPUをレンダリング計算に使えるものもある。以下に比較検討を行なったライブラリ・アプリケーションの詳細を記す。

3－1．PovRay

PovRay[17]（Persistence of Vision Ray tracer）は，独自のシーン記述言語（SDL: Scene Description Language）を用いてシーンを記述し，レイ・トレーシングを行なっていく。たとえば，次のようにシーンを記述していく（PovRay Examplesから引用[18]）。

```
//カメラの配置
camera {
  sky <0,0,1>
  direction <-1,0,0>
  right <-4/3,0,0>
  location <30,10,1.5> //カメラの位置
  look_at <0,0,0>      //カメラの焦点位置
  angle 15     //カメラの画角
}
//環境光の設定
global_settings { ambient_light White }
//光源の設定
light_source {
 <10,-10,20>  //光源の位置
 color White*2        //光源の明るさ
}
//背景色の設定
background { color White }
//床として使うオブジェクト
plane {
  <0,0,1>, 0          //平面の設定 0x+0y+z=0
  texture { T_Silver_3A }      //テキスチャの設定
}
//球体の位置と半径の設定およびテキスチャの設定
sphere { <0,0,1.5>, 1 texture {T_Stone1} }
```

PovRayは，古くからの処理系であり，プラットフォームを選ばないが，問題は，OpenCLやCUDAといったようなGPUを利用する環境には対応していない点にある。そのため，非常にレンダリングされるまで時間が掛かってしまう。負荷軽減のためにクライ

（17）http://www.povray.org
（18）http://www.ms.uky.edu/~lee/visual05/povray/povray.html

アント・サーバ型のネットワーク・レンダリングを行なうための拡張としてHTTPov[19]（A distributed POV-Ray rendering system）が用意されている。しかしながら，すべてCPUで行なうレンダリングなので，ある程度速くすることはできても，1秒間に複数のフレームを作成するまで高速化は難しいと考えられる。

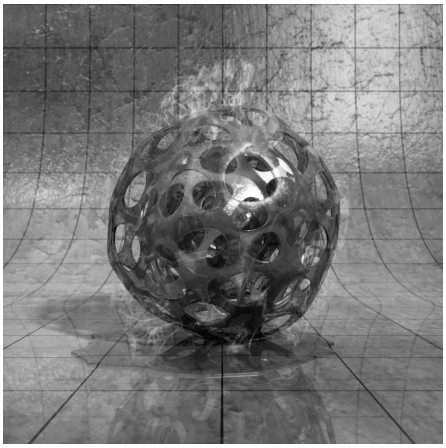

図5　PovRayによるレンダリング例[20]

3－2．LuxRender

LuxRender[21]も，プラットフォームを選ばずにレイ・トレーシングを行なえる処理系である。ただし，Ubuntu 32bit ARM版が稼働しているJetson TK1用には，いくつかのフリーのライブラリをインストールして，ソースファイルからmakeする必要がある。LuxRenderには，3次元モデリング用のソフトウェアBlender[22]用のプラグインがあり，Blenderのリンダリング・エンジンとして用いることができる。このプラグインの中にpylux[23]という，PythonからLuxRenderのシーンを定義して，レンダリングさせるためのライブラリが含まれている。もちろん，LuxRender自身でも独自のシーン記述言語を持っているが，それがPythonでも同じような記述が可能になっている。pyluxでは，例えば，次のようにシーンを記述してレンダリングを行なわせる。

```
import pylux
context = pylux.Context ( "sample context" )
context.worldBegin ()
context.lightSource ( "infinite", [] ) # 光の設定
context.lookAt ( 0,10,100, 0,-1,0, 0,1,0 )
```

(19) https://columbiegg.com/httpov/
(20) Robert McGregor, http://hof.povray.org/rwmcgsphere2_final.html, 2008
(21) http://www.luxrender.net/en_GB/index
(22) https://www.blender.org
(23) http://www.luxrender.net/wiki/LuxBlend_2.5_installation_instructions

```
context.camera ( "perspective", [ "fov", 30.0 ] ) # カメラの画角の設定
context.attributeBegin ( )
context.shape ( "disk", [ "radius", 20.0, "height", 5.0 ] ) # 円筒形の配置
context.attributeEnd ()
context.worldEnd ()
context.updateStatisticsWindow ()
context.exit ()
context.cleanup ()
del context
```

LuxRenderでは，標準でクライアント・サーバ型のネットワーク・レンダリングが可能となっているが，その機能をプログラムからアクセスするためにLuxFire[24]というプラグインも用意されている。LuxFireでは，サーバのコンピュータの種類は異なっていても構わない。LuxRenderは，GPUを使ってレンダリングをするためにOpenCL[25]のライブラリを利用する。OpenCLを用いたライブラリでは，45倍程度高速にレンダリングが可能となる[26]。しかしながら，現在のところOpenCL自体をJetson TK1にインストールできない。OpenCL用のドライバが開発されていないからである。そのため，この機能を用いることはできない。

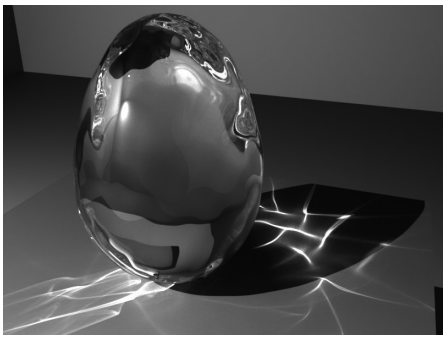

図6　LuxRenderによるレンダリング例[27]

3－3．Cycles

Cycles[28]も，プラットフォームを選ばないが，LuxRenderと同じようにBlenderのためのプラグインとして位置づけされており，主にBlenderのレンダリング・エンジンとして用いられている。また，OSL[29]（Open Shading Language）というシーン記述言語を用い

(24) http://www.luxrender.net/wiki/LuxFire
(25) https://www.khronos.org/opencl/
(26) http://www.luxrender.net/wiki/Luxrender_and_OpenCL
(27) http://www.luxrender.net/wiki/Creating_Beautiful_Caustics
(28) https://wiki.blender.org/index.php/Doc:JA/2.6/Manual/Render/Cycles
(29) https://docs.blender.org/manual/en/dev/render/cycles/nodes/osl.html

ることも可能であるが，GPUレンダリングのみでは使用することができない。Cyclesは，CUDAおよびOpenCLのいずれかを用いて，GPUを使った高速化レンダリングを可能としている。しかしながら，ネットワーク・レンダリングには対応していない。

図7　Cyclesによるレンダリング例[30]

3－4．CUDA RayTracer

CUDA RayTracer[31]（以下RayTracer）は，CUDAをGPGPUとして用いるレイ・トレーシング用のライブラリになっている。RayTracerは，線形代数の数値計算にGLM[32]（OpenGL Math library）を用いている。性能として，60fpsに耐えられるレイ・トレーシングを謳っている。しかしながら，ネットワーク・レンダリングなどの分散レンダリングについては，組込まれていない。シーン記述言語には，TAKUAscene[33]と呼ばれる独自の言語を用いている。TAKUAsceneの記述例は以下のようになる。

```
MATERIAL 0                  //white diffuse
  RGB         1 1 1
  SPECEX      0
  SPECRGB     1 1 1
  REFL        0
  REFR        0
  REFRIOR     0
```

(30) Luis Voronov, https://www.blender.org/features/cycles/, http://studioblender.com
(31) https://github.com/tiansijie/CUDA-RayTracer/blob/master/README.md
(32) http://glm.g-truc.net
(33) https://github.com/tiansijie/CUDA-RayTracer/blob/master/README.md

```
  SCATTER     0
  ABSCOEFF    0 0 0
  RSCTCOEFF   0
  EMITTANCE   0

CAMERA
  RES         800 800
  FOVY        25
  ITERATIONS  300
  FILE        test.bmp
  frame 0
  EYE         0 4.5 12
  VIEW        0 0 -1
  UP          0 1 0

OBJECT 0
  cube
  material 0
  frame 0
  TRANS       0 0 0
  ROTAT       0 0 90
  SCALE       .01 10 10
```

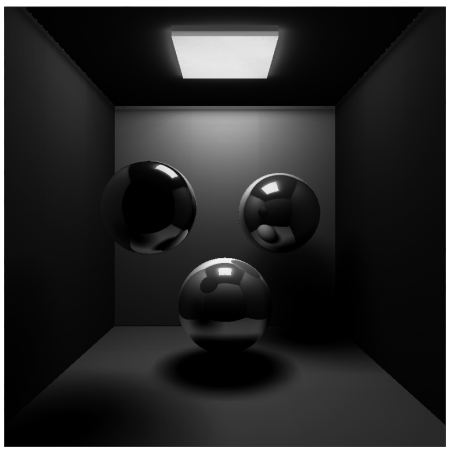

図8　CUDA RayTracerによるレンダリング例[34]

(34) Ricky Arietta, https://github.com/rarietta/CUDA-Raytracer

３－５．NVIDIA® OptiX™ Ray Tracing Engine

OptiX[35]は，NVIDIA社がアナウンスした高速のレイ・トレーシングのエンジンとなっている。これは，CUDAを利用して，高速な物理レンダリングを可能にしており，ネットワークを介したクライアント・サーバ型の分散レンダリングも可能となっている。ただし，稼働環境として64bitのCPUなどを仮定しており，32bitのCPUを持つJetson TK1では利用することができない。CUDAおよびC++を用いて，シーンも記述していく。シーンのC++側における記述例は次のようになる。プログラム全体ではなく，シーンの記述をしている一部分だけを抜き出した。

```
rtContextCreate ( &context ) ;
rtContextSetRayTypeCount ( context, 1 ) ;
rtContextSetEntryPointCount ( context, 1 ) ;

rtBufferCreate ( context, RT_BUFFER_OUTPUT, &buffer ) ;
rtBufferSetFormat ( buffer, RT_FORMAT_FLOAT4 ) ;
rtBufferSetSize2D ( buffer, width, height ) ;
rtContextDeclareVariable ( context, "result_buffer", &result_buffer ) ;
rtVariableSetObject ( result_buffer, buffer ) ;

sprintf ( path_to_ptx, "%s/%s", sutil::samplesPTXDir () ,
"optixHello_generated_draw_color.cu.ptx" ) ;
rtProgramCreateFromPTXFile ( context, path_to_ptx, "draw_solid_color",
&ray_gen_program ) ;
rtProgramDeclareVariable ( ray_gen_program, "draw_color", &draw_color ) ;
rtVariableSet3f ( draw_color, 0.462f, 0.725f, 0.0f ) ;
rtContextSetRayGenerationProgram ( context, 0, ray_gen_program ) ;
```

これと連動するCUDAの記述は以下の通りになる。これも，必要な部分だけを抜き出した。

```
rtDeclareVariable (uint2, launch_index, rtLaunchIndex, ) ;
rtBuffer<float4, 2>   result_buffer;
rtDeclareVariable (float3, draw_color, , ) ;

RT_PROGRAM void draw_solid_color ()
{
 result_buffer [launch_index] = make_float4 (draw_color, 0.f) ;
}
```

(35) http://www.nvidia.co.jp/object/optix_jp.html

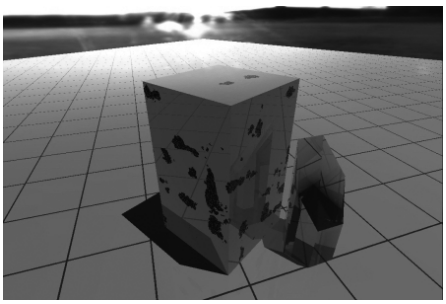

図9　OptiXによるレンダリング例[36]

ここで紹介したもの以外に，GPUを用いたリアルタイム・レンダリングを行なう商用での製品は，Arion[37]，FurryBall[38]，octanerender[39]，V-Ray[40]，Indigo[41]，Redshift[42]などがある。ただし，それはすべてオープンソースではないため，Jetson TK1では稼働しない可能性が高く，調査対象から外した。また，オープンソースのものとしてもYafaRay[43]（Yet another free ray tracer）があり，シーン記述言語として汎用のマークアップ言語XMLを用いているが，これについては，GPUレンダリングもネットワーク・レンダリングもできないため，比較対象から外した。

４．稼働環境の比較

本稿では，稼働環境の比較要素として，GPUを用いたレンダリングが可能であることと，ネットワークを用いた分散レンダリングが可能であることに焦点を当てて表1のように比較結果をまとめた。GPUを用いたレンダリングは，1秒間に複数の画面をレイ・トレーシングすることができる性能を持っている。CPUだけでレンダリングするのとは，桁違いの高速なレンダリングを行なうことが可能となる。また，ネットワークを用いた分散レンダリングが可能であるということは，複数のコンピュータ（本稿ではJetson TK1）を用いたスケール化が可能であることを意味する。コンピュータの台数を増やすことにより，最大で台数分の1の時間でレンダリングを終了させることができる。なお，総じて，どのライブラリでもネットワークを用いて複数のコンピュータで分散レンダリングするライブラリにおいては，その中1台は，並列の物理レンダリングを行なうのと同時に，タスクの配布（Distributor）および結果の回収と表示を行なうようにさせている。

（36）https://docs.nvidia.com/gameworks/content/gameworkslibrary/optix/optix_quickstart.htm
（37）http://www.randomcontrol.com/arion
（38）http://furryball.aaa-studio.eu
（39）https://render.otoy.com/forum/
（40）https://www.chaosgroup.com/vray/application-sdk
（41）https://www.indigorenderer.com
（42）https://www.redshift3d.com/products/redshift
（43）http://www.yafaray.org

表1　各ライブラリにおけるGPU・Network Renderingの可否

	PovRay	LuxRender	Cycles	RayTracer	OptiX
GPU Rendering	×	△	○	○	△
Network Rendering	○	○	×	×	△

△は，GPU Renderingの機能はあるものの，Jetson TK1では使用不可能なものを指す。

5．今後について

Jetson TK1は，CUDAを用いることができるボード型のコンピュータということで，購入したが，リアルタイム・レイ・トレーシングについては，比較で見たように他の進化し続けている一般のパーソナル・コンピュータに比べると見劣りのする結果となった。これまでは，LuxRenderを移植する作業を続けてきたが，結局GPUを用いたレンダリングができず，Jetson TK1上では非常に時間が掛かっている。ここで比較した結果，特にOptiXをJetson TK1用にNVIDIAが提供していないのは，大きな問題があると思われる。物理レンダリングをリアルタイムに行なうためには，CUDAの強みを活かしてRayTracerを中心に開発をする必要があるが，これをOpenMPI[44]などのネットワーク通信を行なうライブラリを用いて，並行計算ができるような形に書き直す必要がある。CUDAについてもPyCUDA[45]，OpenMPIについてもPyMPI[46]などのPythonへの移植版があり，Pythonを用いて記述することが可能になっている。これらのライブラリを用いて，Pythonで分散レンダリングを記述するところから始めていきたいと考えている。

この研究は，本学の個人研究助成によって助成された。ここに感謝の辞を述べる。

（2017.2.6受稿，2017.2.28受理）

（44）https://www.open-mpi.org
（45）https://mathema.tician.de/software/pycuda/
（46）http://pympi.sourceforge.net

〔抄　録〕

Jetson TK1という，汎用計算も可能なGPU（Graphic Processing Unit）が組み込まれた組込み型のボードコンピュータがNVIDIAからリリースされたことを受け，これを複数台組み合わせて，リアルタイム・レンダリングを可能とするようなシステムを制作することを研究の目標としている。この研究ノートでは，そのシステム制作に必要な物理レンダリングを行なうライブラリの選定・評価を行なった結果について記した。オープンソースであり，Jetson TK1上でも稼働でき，GPUによる高速レンダリングが可能で，かつネットワークを介した分散レンダリングが可能なものについて調査した。この研究は，本学の個人研究助成によって助成された。