

並列可視化システム HIVEのチュートリアル

2015-09-01

チュートリアル内容

- ・ HIVE について
 - ・ HIVEの動作モード
 - ・ HIVE のビルドとインストール
 - ・ HIVE UI,SceneNodeEditor の起動と使いかた
- ・ hrenderについて
 - ・ hrender概要
 - ・ データローダについて
 - ・ フィルタ機能について
 - ・ 可視化フロー例
 - ・ MPI並列レンダリングについて
- ・ シェーダについて
- ・ SceneNodeEditorについて
 - ・ SceneNodeEditorによる簡単な可視化例
 - ・ NodeEditorで作成したシーンのhrenderでのレンダリング例
- ・ HIVE UIについて
 - ・ ボリュームレンダリングの伝達関数について
 - ・ マルチカメラの設定について
 - ・ HIVE UIでのムービー作成方法
 - ・ タイムステップデータのムービー作成方法
 - ・ HIVEUIへの独自シェーダ登録方法

HIVEについて

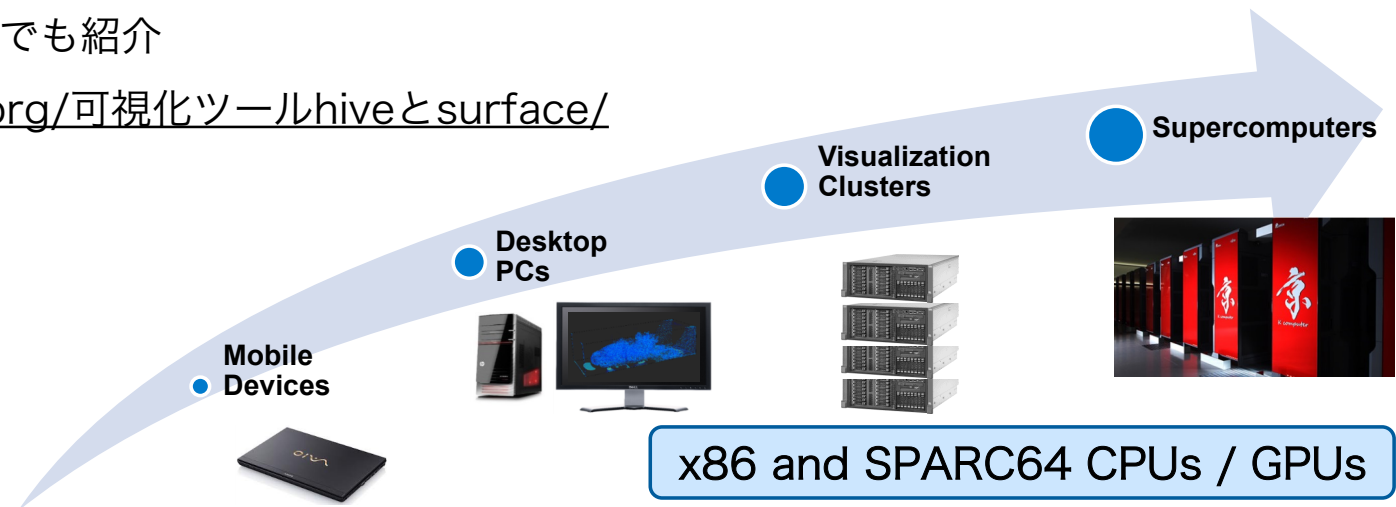
HIVE概要

- **Heterogeneously Integrated Visual analytic Environment**

- 大規模データの効率的な可視化を支援するシステム
 - 大規模計算結果を移動することなく、HPC環境下でも動作
- webインターフェイスを採用し、様々なプラットフォームで動作
- スケーラブルな性能
- カスタマイズ可能
 - シェーダーコードを書き、質感や表現を制御可能

- HIVEは以下のサイトでも紹介

<http://www.cenav.org/可視化ツールhiveとsurface/>



HIVE概要

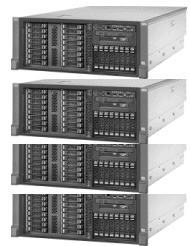
- HIVEは以下のサイトでも紹介

<http://www.cenav.org/可視化ツールhiveとsurface/>



京, Linux環境など MacOSX, Linux環境など

SURFACE, hrender



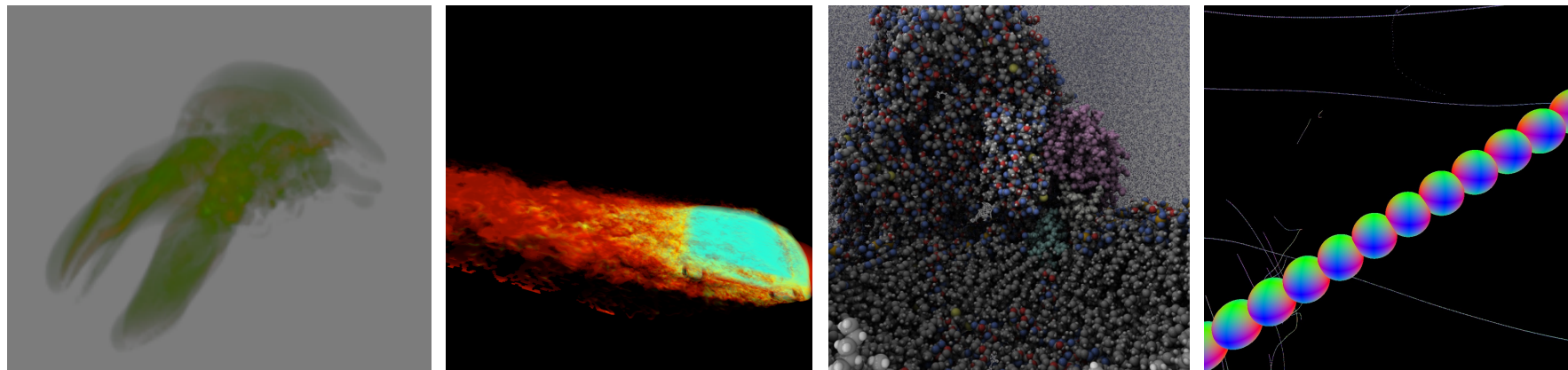
ターミナル

ブラウザ

SURFACE

Scalable and Ubiquitous Rendering Framework
for Advanced Computing Environment

アーキテクチャ非依存の並列レンダリングフレームワーク




**ULTRASCALE
VISUALIZATION WORKSHOP**

November 16, 2014 | New Orleans, LA



Scientific Discovery through Advanced Computing




Computer simulations create the future

SURFACE
A Visualization Framework
for Large-Scale Parallel Simulations


$$I_j = \int_{\Omega} I_j(x) d\mu(x)$$

Jorji Nonaka, Kenji Ono
RIKEN AICS
Kobe, Japan

Masahiro Fujita
LTE Inc.
Tokyo, Japan



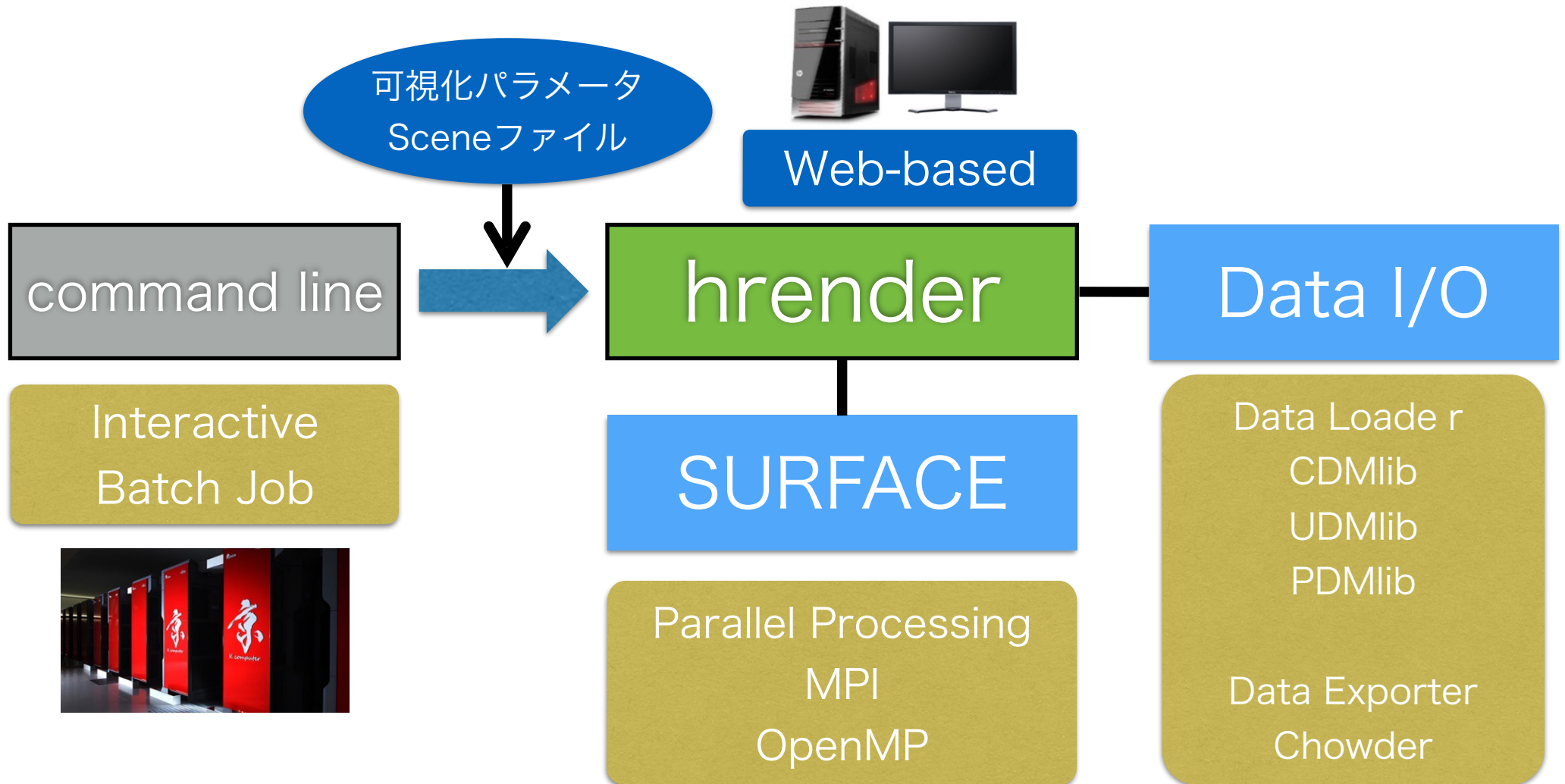
November 16th, 2014



RIKEN ADVANCED INSTITUTE FOR COMPUTATIONAL SCIENCE

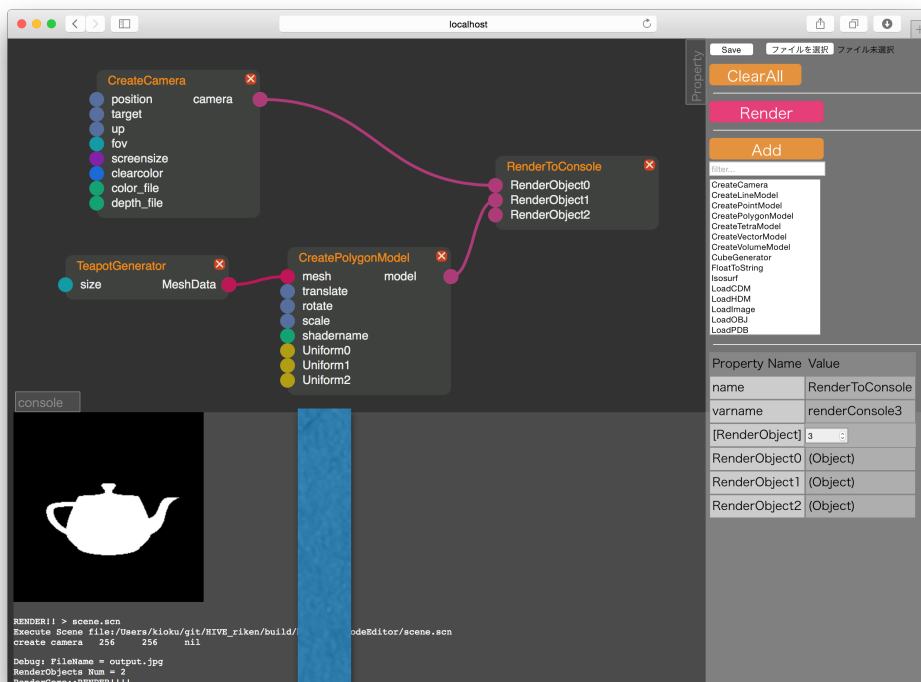
hrender

SURFACEが提供するレンダリング機能をコールし一連の可視化処理を行う単体アプリケーション

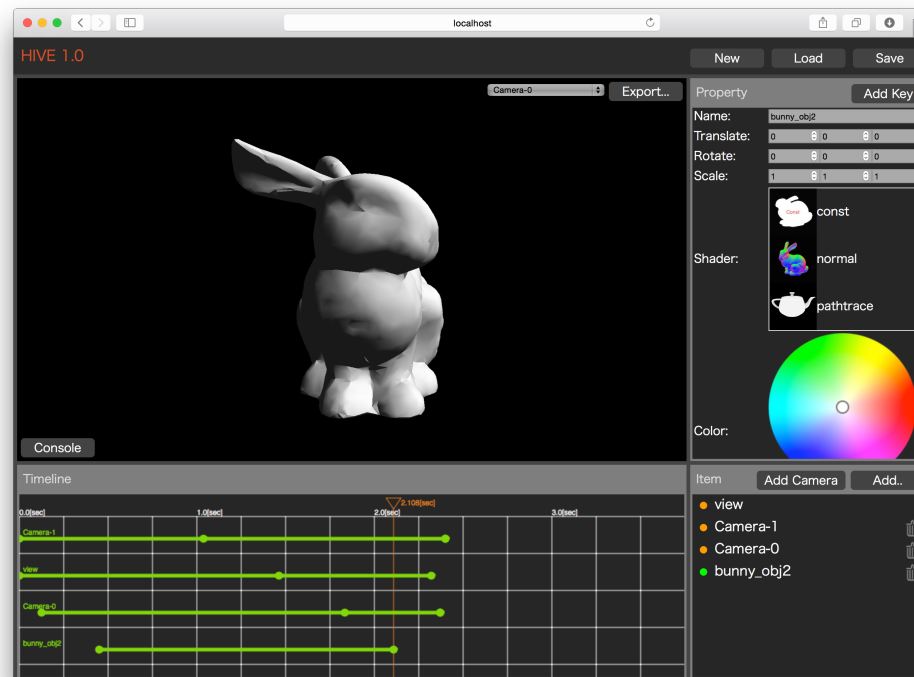


hendererを用いたWebベースアプリケーション

SceneNodeEditor



HIVE UI



アニメーション作成支援ツール

Sceneファイル
作成支援ツール

```
print('Render Obj')
local camera = Camera()
camera:SetScreenSize(1024, 1024)
camera:SetFilename('render_obj.jpg')

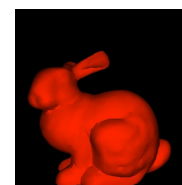
camera:LookAt(
    0,0,100,
    0,0,0,
    0,1,0,
    60
)

local obj = OBJ.Loader()
obj:Load('bunny.obj')

local model = PolygonModel()
local meshdata = obj.MeshData()
model:Create(meshdata)
model:SetShader('normal.frag')

local analy = PolygonAnalyzer()
analy:Execute(model)
print(analy.MinX())
print(analy.MaxX())
print(analy.MinY())
print(analy.MaxY())
print(analy.MinZ())
print(analy.MaxZ())

render {camera, model}
```



ffmpeg

output.mp4

HIVE Applications

HIVEは以下の3つのアプリケーションから構成されています。

```
# コマンドラインでレンダリング
```

```
$ cd $HIVE/hrender/test  
$ mpirun -np 1 ../../build/bin/hrender  
render_obj.scn
```

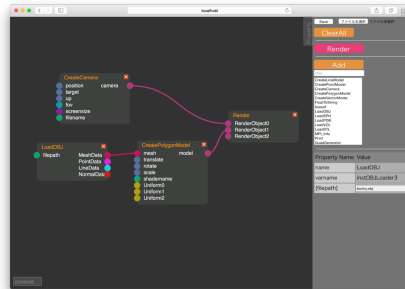
hrender

SURFACE

standalone renderer

raytracing library

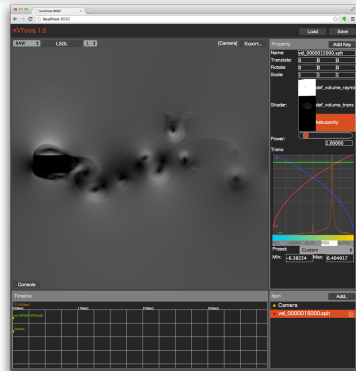
コマンドラインで動作し、並列処理（スレッド並列、プロセス並列）に対応



SceneNodeEditor

web interface node editor

webインターフェイスを用いた可視化ユーザインターフェイスで、hrenderをコールして処理を実施



HIVE UI

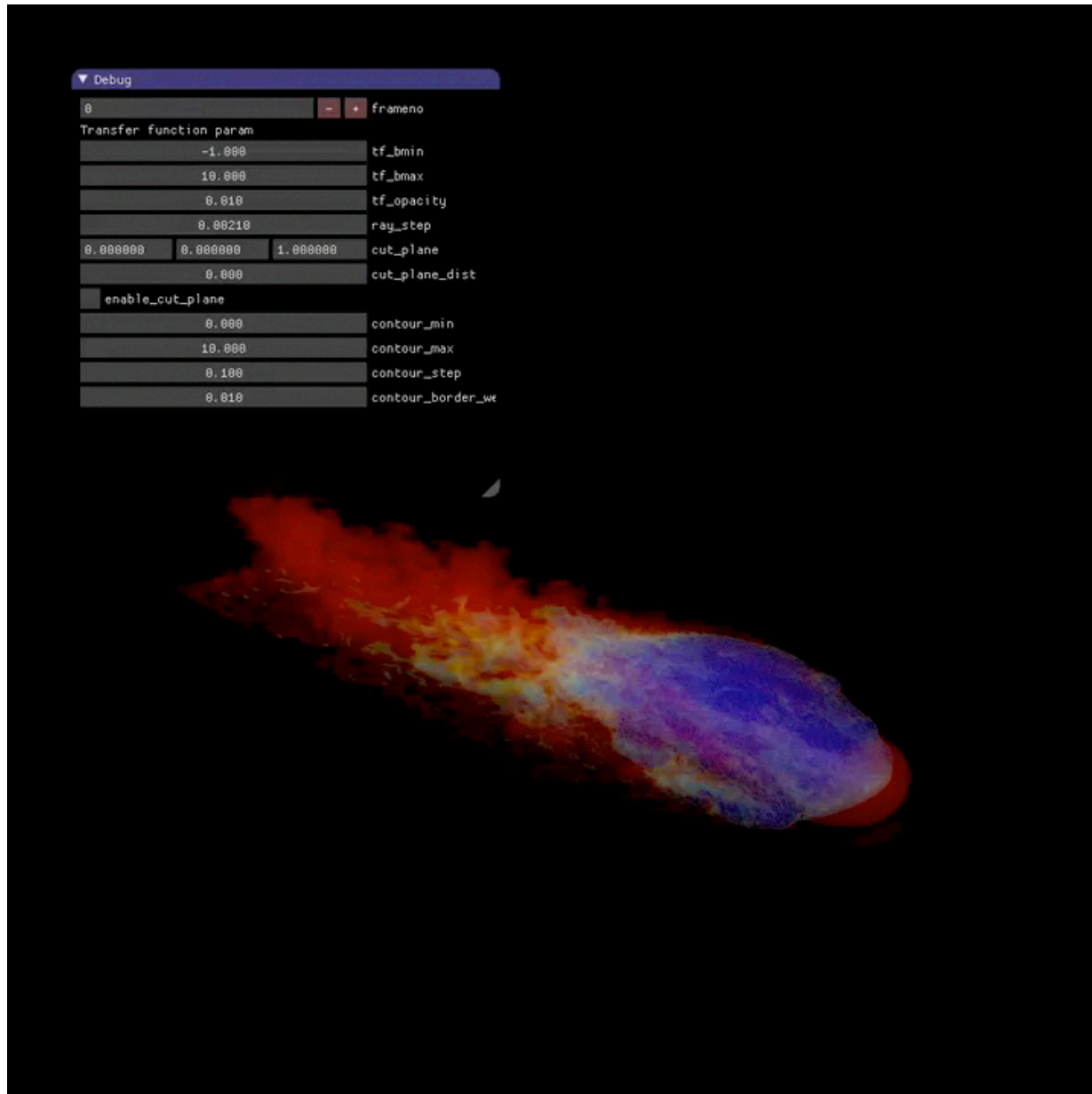
web interface animation editor

webインターフェイスを用いたアニメーション作成インターフェイスで、hrenderをコールして処理を実施

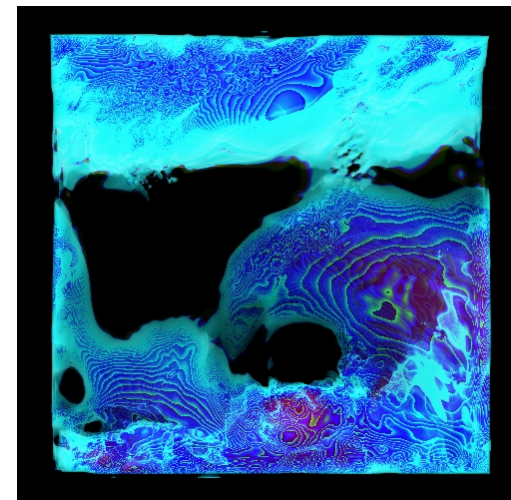
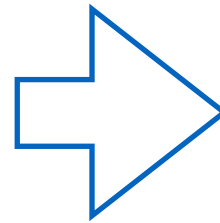
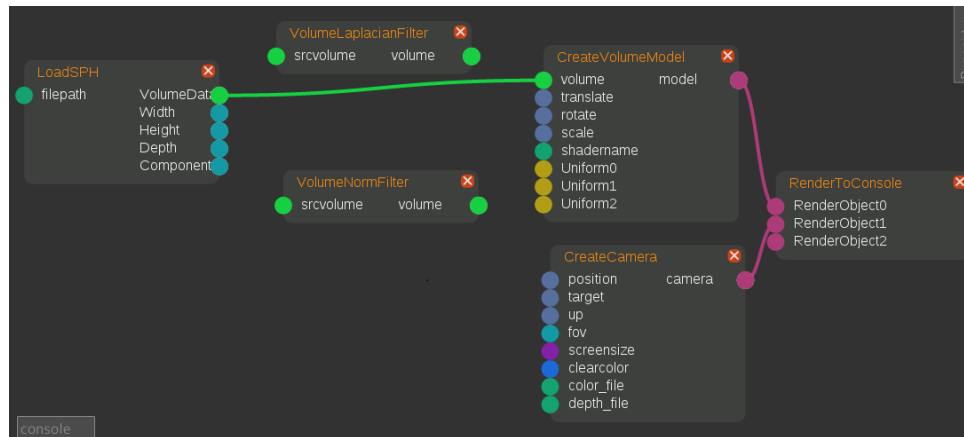
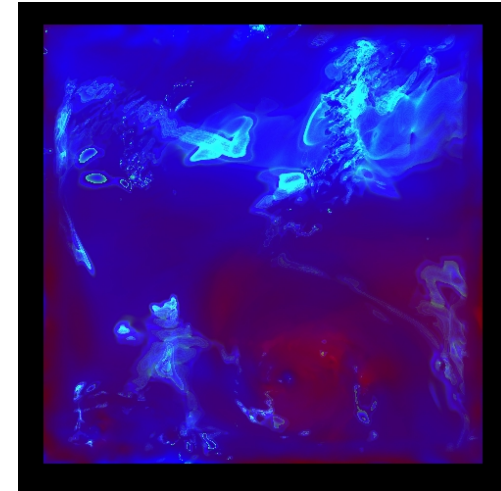
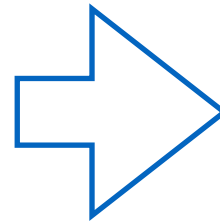
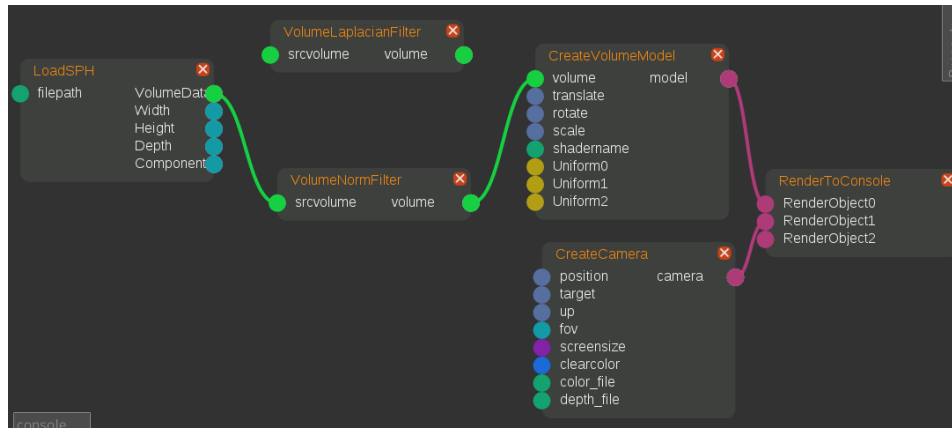
HIVE動作環境

- ・ ユーザー環境
 - ・ Linux 64bit : CentOS6, 7 で動作確認
 - ・ Mac : MPI/OpenMP 利用時は brew などで gcc 4.8 以上をインストールしてください。
HPC on MacOSX: <http://hpc.sourceforge.net>
 - ・ Windows(予定)
- ・ HPC環境
 - ・ 京コンピュータ(演算ノード, ポスト処理ノード)
 - ・ FOCUS

車のなにか

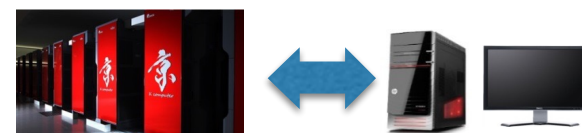
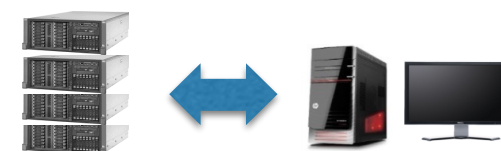


台風のかなにか



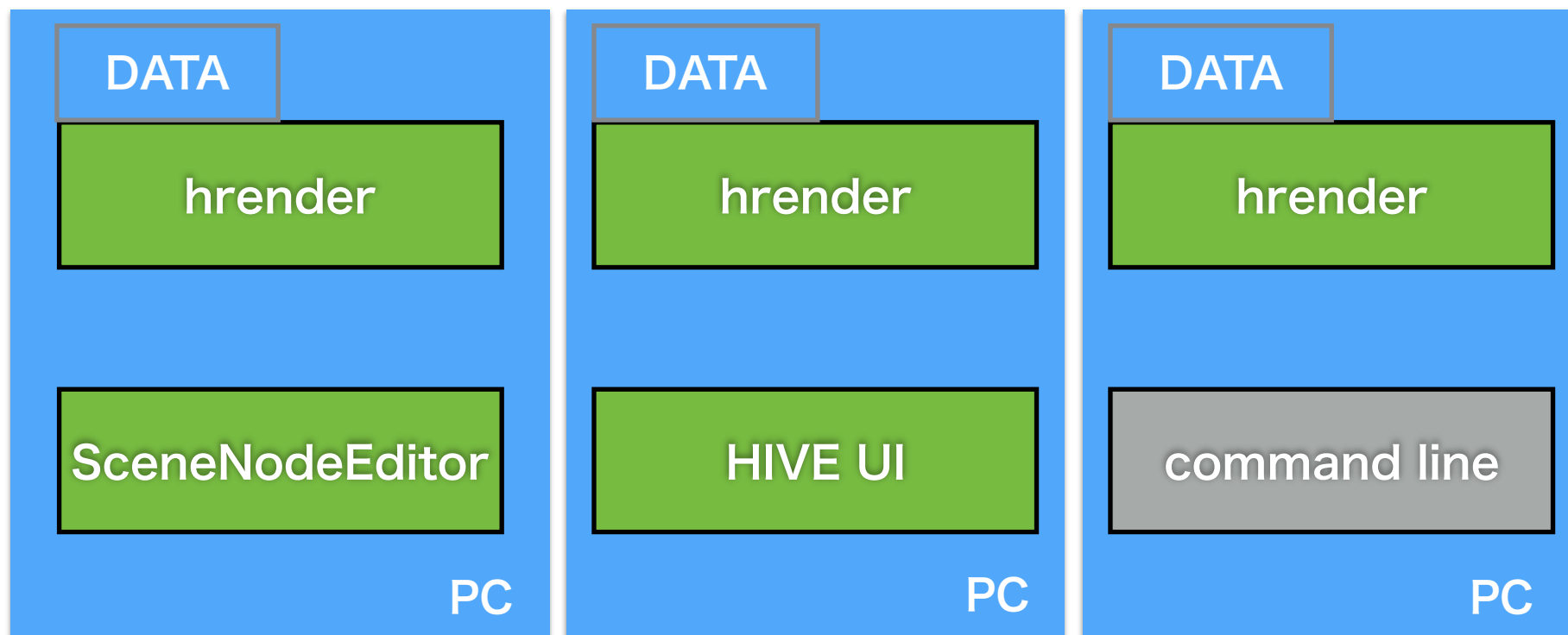
HIVEの動作モードについて(1)

- HIVEは以下の構成で動作することが可能です。
 - ローカル可視化（単体）
 - リモート可視化（単体、クラ・サーバ）
 - コンカレント可視化（単体、クラ・サーバ）
- 現在研究開発中 >> 将来機能として取り入れる
 - In-Situ可視化
 - データ分析機能



HIVEの動作モードについて(2)

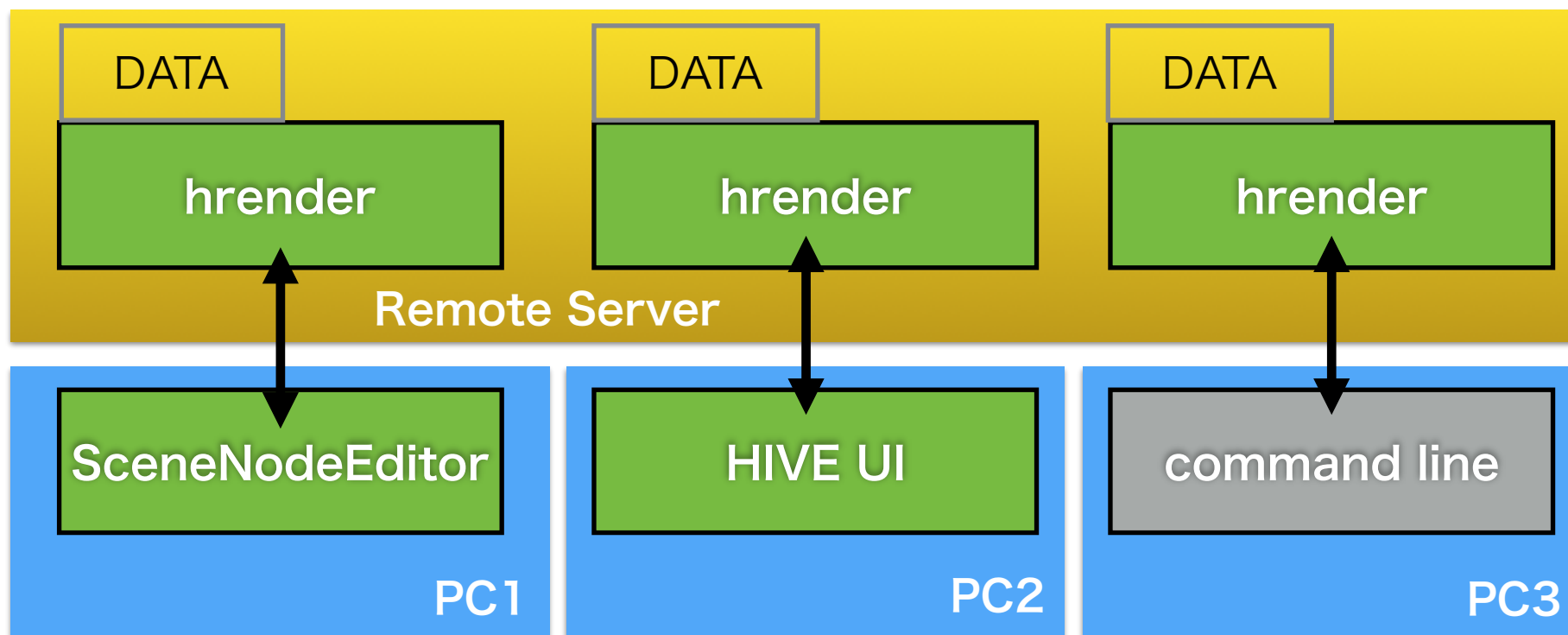
- ローカルPC上にあるデータの可視化が可能です。



ローカル可視化

HIVEの動作モードについて(2)

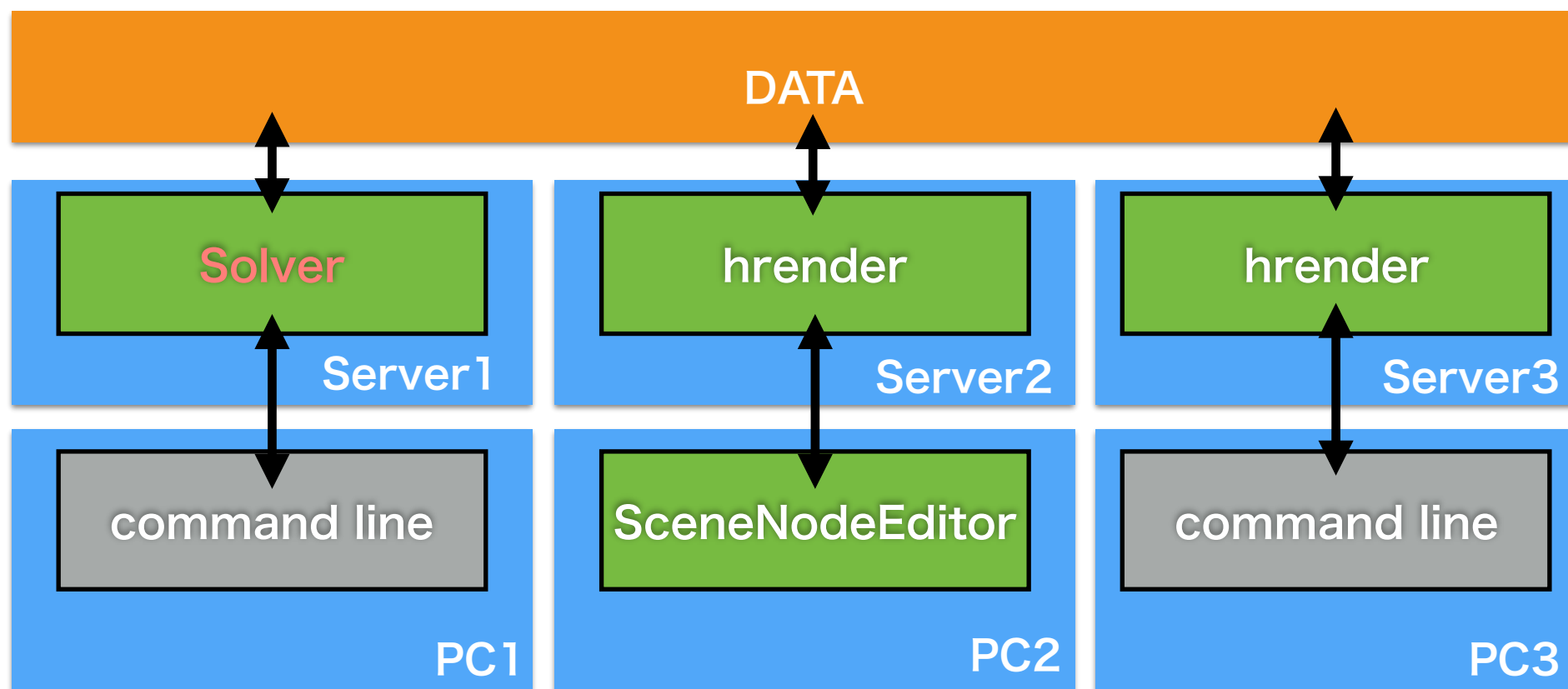
- リモート計算機上にあるデータの可視化が可能です。



リモート可視化

HIVEの動作モードについて(3)

- シミュレータが出力するデータを共用ストレージを経由して、計算と同時に可視化



コンカレント可視化

HIVEの ビルドとインストール

必要なツールとライブラリ

- ・ **cmake 2.8** 以降 (2.8.12 以降推奨)
- ・ **gcc 4.4** 以降 (gcc 4.8 以降推奨)
 - ・ Intel C Compilerの場合はVer.13 以降推奨
- ・ **MPI** コンパイラ環境
- ・ **autotools** (autoconf, automake, libtools)
- ・ **zlib**
- ・ **node.js v0.12.6** (<http://nodejs.org>)
 - ・ HTML5 UI ツールで利用

hrender を動かす環境にて適切にパスが設定されているものとします

その他ライブラリ（データローダ向け）

・**Zoltan 3.81**

HDMLib, PDMLib, UDMLIB

・<http://www.cs.sandia.gov/Zoltan/>

・**fpzip 1.0.1**

PDMLib

・<https://computation.llnl.gov/casc/fpzip/>

・**cgnslib 3.2.1**

HDMLib

・<http://cgns.github.io>

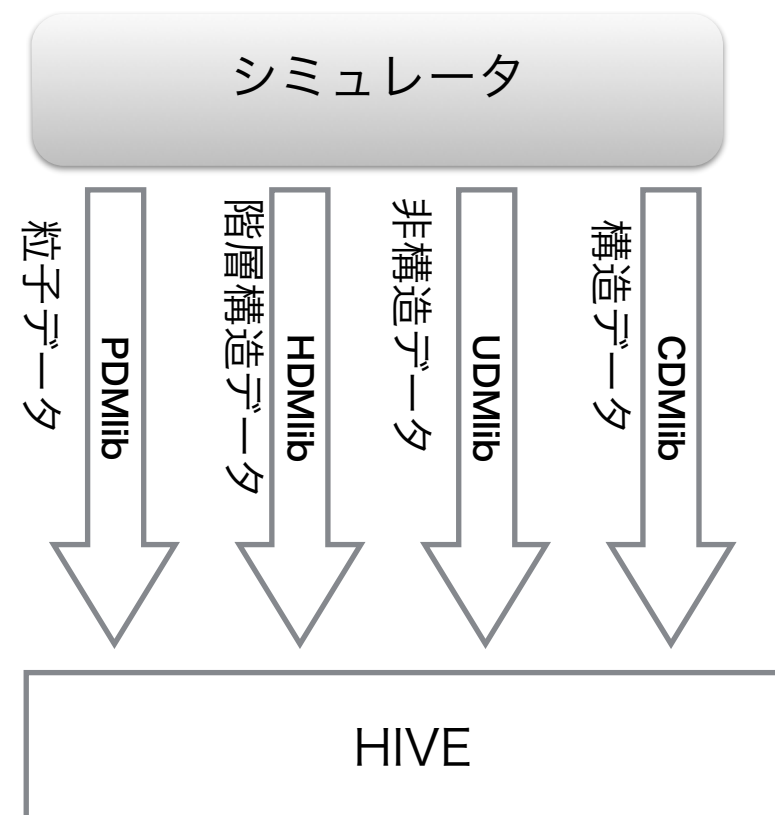
・ユーザ登録は必須ではないが、推奨されている

これらのライブラリは、権利およびライセンスの都合上、HIVE に含めて配布できないため、ユーザごとに登録とダウンロードを事前にお願ひします。

データローダーライブラリ xDMLib

- ・ CDMlib <http://avr-aics-riken.github.io/CDMLib/>
- ・ HDMLib <https://github.com/avr-aics-riken/HDMLib>
- ・ UDMLib <http://avr-aics-riken.github.io/UDMLib/>
- ・ PDMLib <http://avr-aics-riken.github.io/PDMLib/>

分散ファイルをメタファイルにより管理し、
入出力機能を提供するライブラリ群



注意事項

- **Intel C Compiler 13**

- ICC の不都合(?)により, 長いパスの環境下では cgns のビルドに失敗します.
 - 対処法: ~/work/HIVE 程度の短いパスでビルドします.

- **npm (node.js package manager)**

- LSF の npm とコマンド名がかぶるため, LSF がインストールされている環境では, 明示的なパスで node.js npm コマンドを指定するのを推奨します.

HIVEの取得

- ・ ~/work/HIVE ディレクトリで作業すると仮定します.
- ・ 以降, \$HIVE と記載します.
- ・ Gitで取得(clone)するか、Zipファイルをダウンロードします。

```
$ cd ~/work
$ git clone https://github.com/avr-aics-riken/HIVE.git
$ cd HIVE
$ git submodule update --init

# fpzip, zoltan, cgns ライブラリを third_party ディレクトリにコピーします
# HIVE を UDMlib, CDMlib ローダライブラリとリンクしてビルドしない場合は不要です。

$ cp /path/to/fpzip-1.0.1.tar.gz third_party/
$ cp /path/to/zoltan_distrib_v3.81.tar.gz third_party/
$ cp /path/to/cgnslib_3.2.1.tar.gz third_party/
```

必要に応じてあらかじめダウンロードしておいたライブラリーをコピー

HIVE cmake オプション

- ビルドスクリプトを利用しない場合は, 以下のオプションを cmake に指定してカスタムのビルドを作ることができます.

HIVE_BUILD_K_CROSS_COMPILE	On/Off	K/FX10 クロスコンパイルを行います.
----------------------------	--------	-----------------------

HIVE_BUILD_WITH_MPI	On/Off	MPI ビルドを行うかを指定します. ローダライブラリ, Compositor を利用する場合は On が必須になります.
---------------------	--------	---

HIVE_BUILD_WITH_OPENMP	On/Off	OpenMP の利用を指定します.
------------------------	--------	-------------------

HIVE_BUILD_WITH_CDMLIB	On/Off	CDMLib とリンクするかを指定します.
------------------------	--------	-----------------------

HIVE_BUILD_WITH_PDMLIB	On/Off	PDMLib とリンクするかを指定します.
------------------------	--------	-----------------------

HIVE_BUILD_WITH_HDMLIB	On/Off	HDMLib とリンクするかを指定します.
------------------------	--------	-----------------------

HIVE_BUILD_WITH_UDMLIB	On/Off	UDMLib とリンクするかを指定します.
------------------------	--------	-----------------------

HIVE_BUILD_WITH_COMPOSITOR	On/Off	234 Compositor とリンクするかを指定します.
----------------------------	--------	-------------------------------

推奨構成のビルドスクリプト

- ・ HIVE/scripts に推奨する構成でビルドを行うビルドスクリプトを用意してあります.
 - ・ Linux + GNU Compiler
 - ・ Linux + Intel Compiler
 - ・ K / FX10 環境
 - ・ Mac OSX 環境
- MPI
OPENMP
HDMLIB
PDMLIB
UDMLIB
CDMLIB
COMPOSITOR
- ・ 必要に応じてビルドスクリプトの編集を行ってください。

Linux + Intel compiler 環境でのビルド

```
build_loader_libs_linux-x64-icc.sh  
cmake_linux-x64-all-icc.sh
```

```
# HIVE が利用するローダライブラリを一式ビルドします。  
$ ./scripts/build_loader_libs_linux-x64-icc.sh  
  
# 全ローダライブラリとリンクする設定で HIVE の configure を行います。  
# (必要であれば事前に .sh を編集し, cmake のパスなどを設定します)  
$ rm -rf build # 以前の build 物があれば削除  
$ ./scripts/cmake_linux-x64-all-icc.sh  
$ ./scripts/cmake_linux-x64-all-icc.sh # cmake の設定を確実にを行うため, 2 回繰り返します。  
  
# build ディレクトリで HIVE のコンパイルを行います。  
$ cd build  
$ make
```

Linux + gcc 環境でのビルド

```
build_loader_libs_linux-x64.sh  
cmake_linux-x64-all.sh
```

```
# HIVE が利用するローダライブラリを一式ビルドします。  
$ ./scripts/build_loader_libs_linux-x64.sh  
  
# 全ローダライブラリとリンクする設定で HIVE の configure を行います。  
# (必要であれば事前に .sh を編集し, cmake のパスなどを設定します)  
$ rm -rf build # 以前の build 物があれば削除  
$ ./scripts/cmake_linux-x64-all.sh  
$ ./scripts/cmake_linux-x64-all.sh # cmake の設定を確実にを行うため, 2 回繰り返します。  
  
# build ディレクトリで HIVE のコンパイルを行います。  
$ cd build  
$ make
```

K/FX10 環境でのビルド

```
build_loader_libs_k_cross.sh  
cmake_k_cross.sh
```

```
# HIVE が利用するローダライブラリを一式ビルドします。  
$ ./scripts/build_loader_libs_k_cross.sh  
  
# 全ローダライブラリとリンクする設定で HIVE の configure を行います。  
# (必要であれば事前に .sh を編集し, cmake のパスなどを設定します)  
$ rm -rf build # 以前の build 物があれば削除  
$ ./scripts/cmake_k_cross.sh  
$ ./scripts/cmake_k_cross.sh # cmake の設定を確実にを行うため, 2 回繰り返します。  
  
# build ディレクトリで HIVE のコンパイルを行います。  
$ cd build  
$ make
```

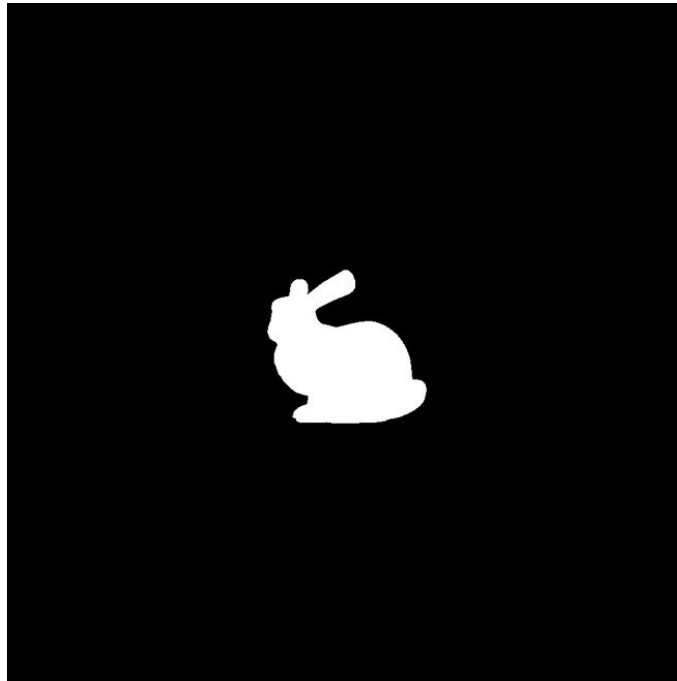
MacOSX 環境でのビルド

```
build_loader_libs_macosx.sh  
cmake_macosx.sh
```

```
# HIVE が利用するローダライブラリを一式ビルドします。  
$ ./scripts/build_loader_libs_macosx.sh  
  
# 全ローダライブラリとリンクする設定で HIVE の configure を行います。  
# (必要であれば事前に .sh を編集し, cmake のパスなどを設定します)  
$ rm -rf build # 以前の build 物があれば削除  
$ ./scripts/cmake_macosx.sh  
$ ./scripts/cmake_macosx.sh # cmake の設定を確実にを行うため, 2 回繰り返します。  
  
# build ディレクトリで HIVE のコンパイルを行います。  
$ cd build  
$ make
```

hrender 動作確認

```
# hrender を使い, コマンドラインでレンダリング(可視化)できるのを確認します.  
  
$ cd $HIVE/hrender/test  
$ mpirun -np 1 ../../build/bin/hrender render_obj.scn  
  
# render_obj.jpg が生成され, 以下のように白い bunny がレンダリングされれば成功です.
```



SceneNodeEditor, HIVE UI

```
# SceneNodeEditor, HIVE_UI の実行に必要な
# node.js モジュールの取得とビルドを行います。
# node_modules を展開します。
$ cd $HIVE/build/bin/SceneNodeEditor
$ npm install

$ cd $HIVE/build/bin/HIVE_UI
$ npm install

# server.js, hrender_server.lua を編集し、
# ポート番号(デフォルト 8080) を空いている番号へ変更します。
$ vi $HIVE/build/bin/HIVE_UI/server.js
$ vi $HIVE/build/bin/HIVE_UI/hrender_server.lua
$ vi $HIVE/build/bin/SceneNodeEditor/server.js

# ポート番号は起動時にも設定できます。
```

HIVE UI

```
# HIVE UI の起動を確認します。
```

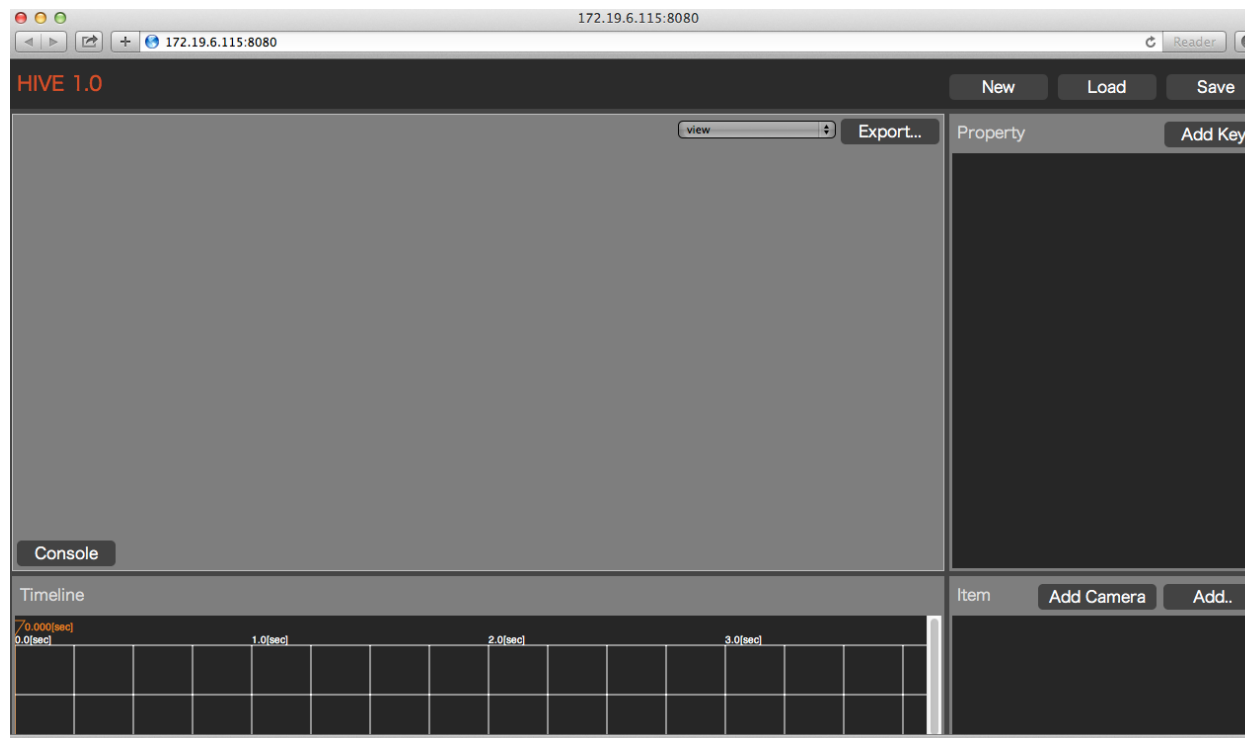
```
$ cd $HIVE/build/bin/HIVE_UI
```

```
$ node server.js
```

```
# またはポート番号[portnum]を指定して起動
```

```
$ node server.js [portnum]
```

```
# http://localhost:[portnum] を手元 PC のブラウザで開き、以下の画面が表示されれば成功です。
```



SceneNodeEditor

```
# SceneNodeEditor の起動を確認します。
```

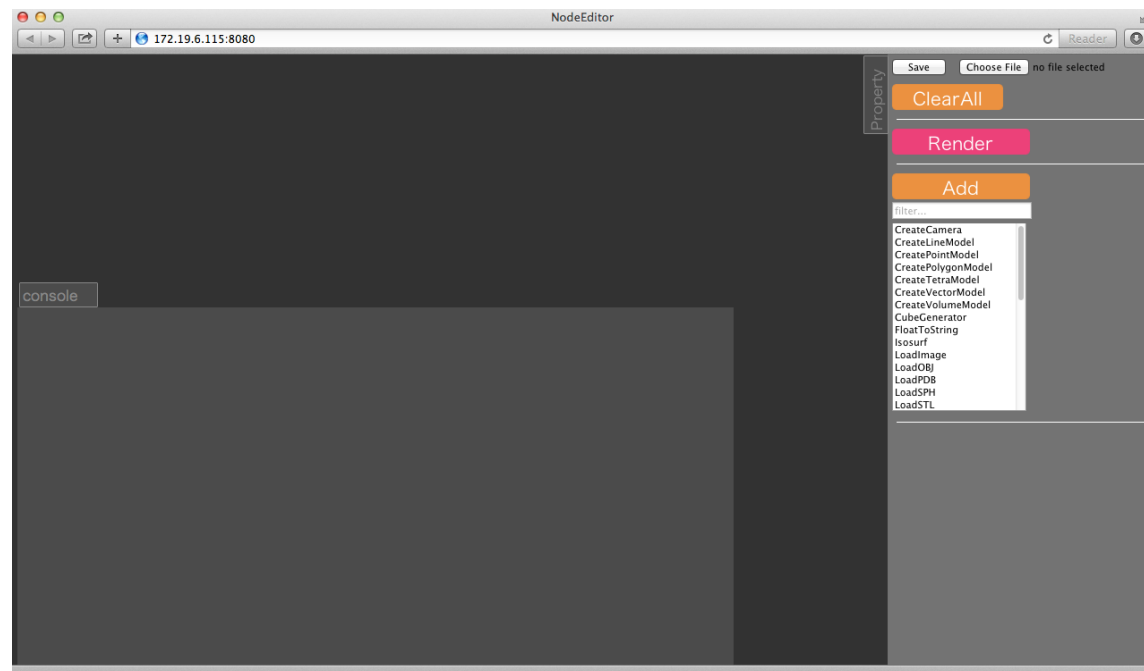
```
$ cd $HIVE/build/bin/SceneNodeEditor
```

```
$ node server.js
```

```
# またはポート番号[portnum]を指定して起動
```

```
$ node server.js [portnum]
```

```
# http://localhost:[portnum] を手元 PC のブラウザで開き, 以下の画面が表示されれば成功です。
```



SSH ポートフォワーディング

- ・ サーバがリモートにあり, クライアントからは直接見えない(ssh でアクセスできる)場合は, SSH ポートフォワーディング機能を使います.

```
# クライアント側で行います.
```

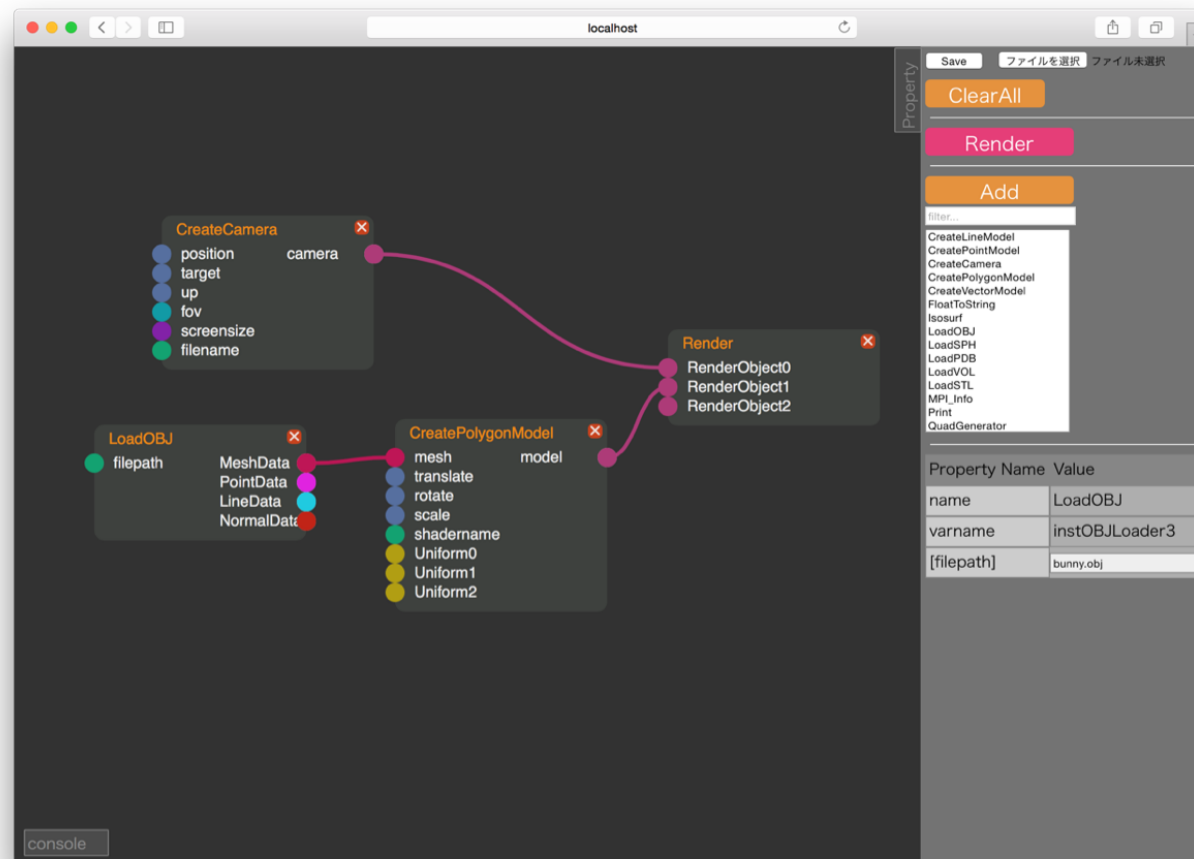
```
$ ssh -L 8080:localhost:8080 serveraddr
```

```
# これにより, http://localhost:8080 にアクセスすると, serveraddr:8080 につながります.
```

SceneNodeEditor

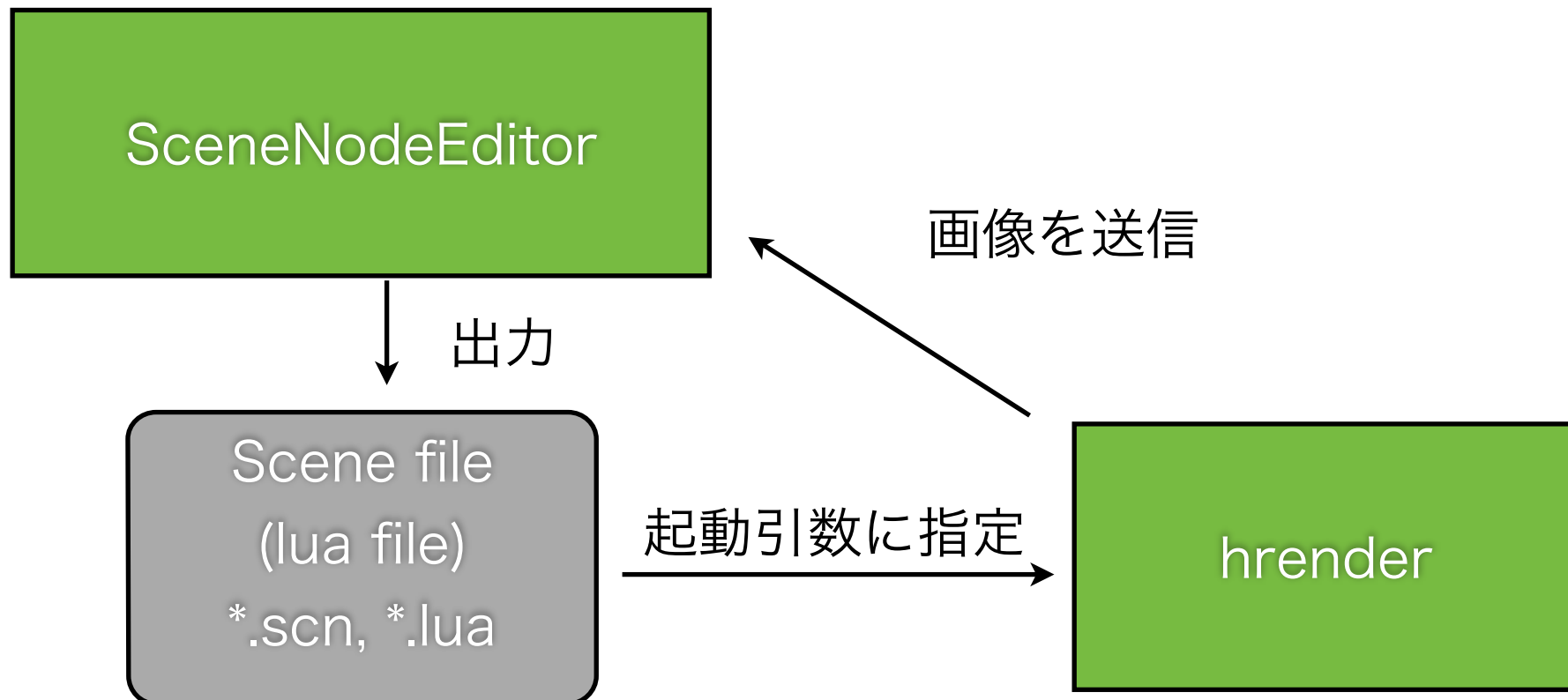
SceneNodeEditor外観

SceneNodeEditorはシーンファイル作成のための
Webブラウザベースアプリケーション



SceneNodeEditor概要(1)

SceneNodeEditorはLuaを出力する



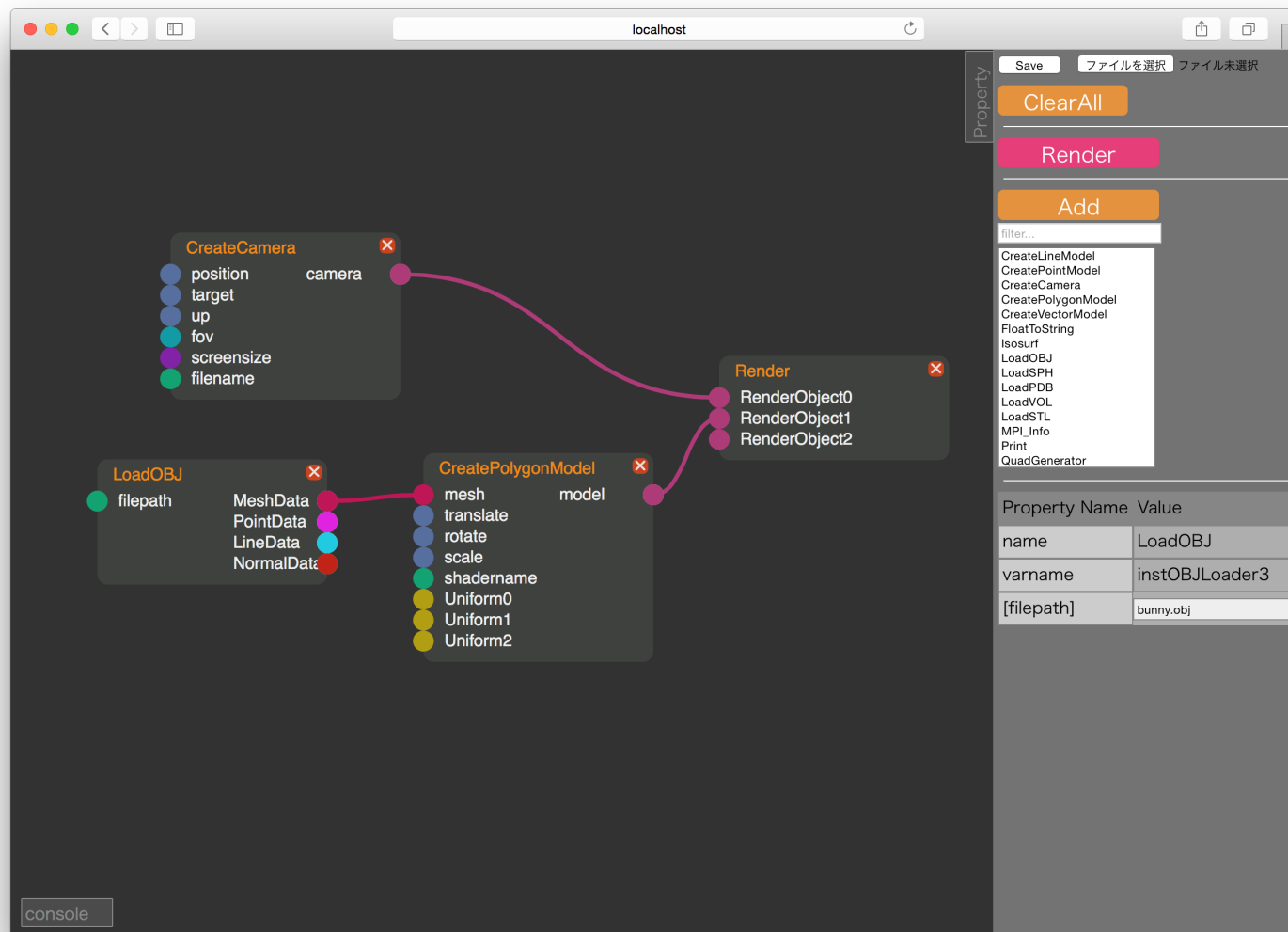
SceneNodeEditor概要(2)

- ・ SceneNodeEditor(以下NodeEditor)はScene fileの作成を補助するためのツールです。
- ・ script interfaceを習得することなく、直感的なScene fileの作成が可能となります（**初学者向け**）。
- ・ 作成したScene fileを、hrenderの入力ファイルとしてレンダリングが可能です。
- ・ NodeEditorで作成したScene fileをベースにしたシーン編集も可能です。

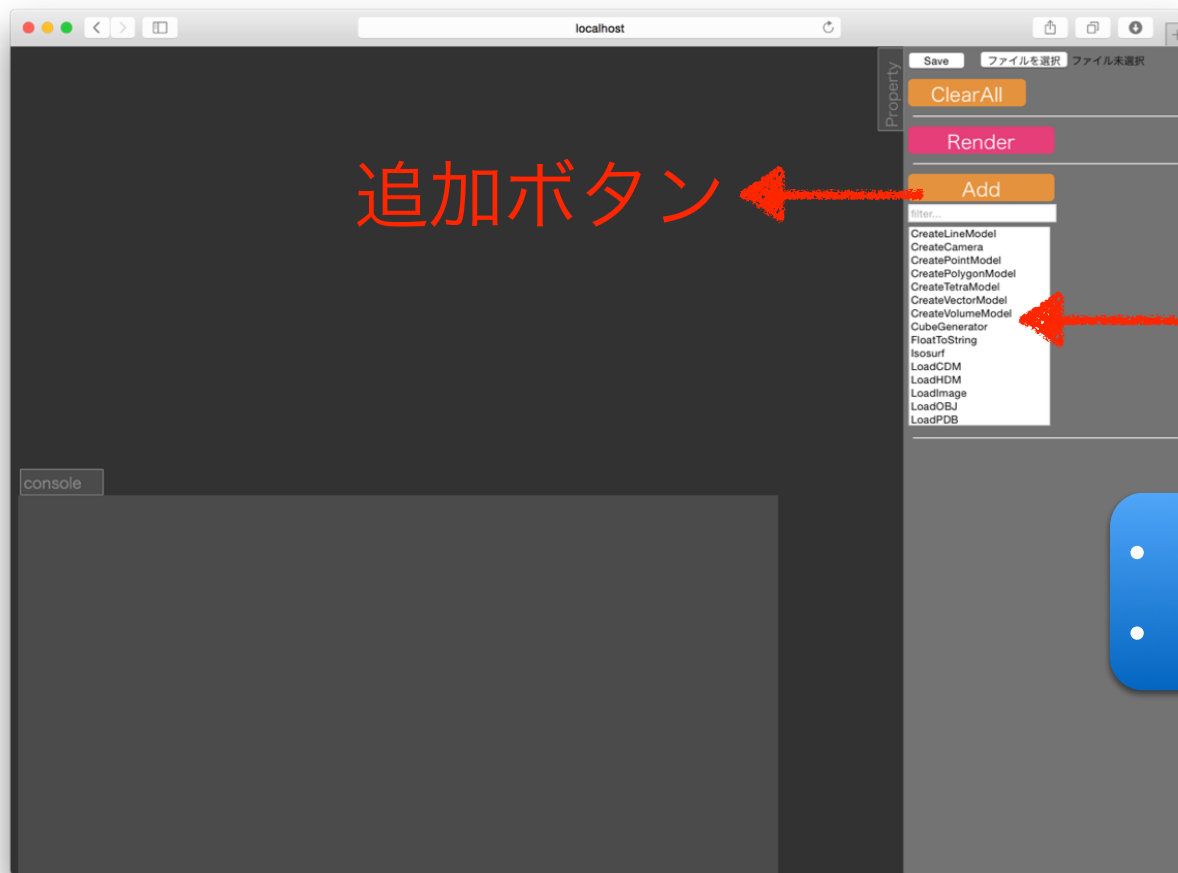
SceneNodeEditorによる 簡単な可視化例

SceneNodeEditor外観

SceneNodeEditorはシーンファイル作成のための
Webブラウザベースアプリケーション



NodeEditorによる可視化例(1)



追加ボタン

ノードを
一覧から選択

- CreateCamera
- RenderToConsole

何も無いシーンをレンダリングし、結果をNodeEditorで表示

SceneNodeEditorによる可視化例(1)

① ノード同士をつなげる

② hrender 実行

③ 結果画像

The screenshot shows the SceneNodeEditor interface. On the left, a 'CreateCamera' node is connected to 'RenderObject0', 'RenderObject1', and 'RenderObject2' nodes. A blue callout bubble points to these connections with the text 'ノード同士をつなげる' (Connect nodes). On the right, a 'Property' panel is visible with a 'Render' button highlighted by a red arrow. Below the 'Render' button is an 'Add' button and a list of available nodes. At the bottom, a 'console' window displays the following output:

```
RENDER!! > scene.scn
Execute Scene file:/Users/kioku/git/HIVE_riken/build/bin/SceneNodeEditor/scene.scn
create camera 256 256 nil

ERR: cannot find program object for handle: 0
ERR: cannot find program object for handle: 0
ERR: cannot find program object for handle: 0

Debug: FileName = output.jpg
RenderObjects Num = 1
RenderCore::RENDER!!!!
Save:output.jpg
[HIVE] Resize=0.000 DrawCall=0.000 Readback=0.001 Save=0.002
RenderObject0 CAMERA

Send: ws://localhost:58432
connect > ws://localhost:58432
```

Property Name	Value
name	CreateCamera
varname	instCreateCamera4
[position]	0 0 300
[target]	0 0 0
[up]	0 1 0
[fov]	60
[screensize]	256 256
[clearcolor]	0 0 0
[color_file]	output.jpg
[depth_file]	

SceneNodeEditorによる可視化例(1)



- **RenderToConsole**ノード

- レンダリングを実行し、結果をSceneNodeEditorに送る

- **CreateCamera**ノード

- カメラオブジェクトを作成し、パラメータを設定する
- レンダリング解像度、背景色、視点位置、結果画像ファイル名が指定可能

Property Name	Value
name	CreateCamera
varname	instCreateCamera4
[position]	0 2 8
[target]	0 2 0
[up]	0 1 0
[fov]	60
[screensize]	256 256
[clearcolor]	0 0 0 1
[color_file]	output.jpg
[depth_file]	

SceneNodeEditorによる可視化例(2)

teapotモデルをレンダリングし、結果をNodeEditorで表示

結果
画像



The screenshot shows the SceneNodeEditor interface with the following components:

- CreateCamera** node: position, target, up, fov, screensize, clearcolor, color_file, depth_file.
- TeapotGenerator** node: size.
- CreatePolygonModel** node: mesh, translate, rotate, scale, shadername, Uniform0, Uniform1, Uniform2.
- RenderToConsole** node: RenderObject0, RenderObject1, RenderObject2.

The rendered teapot is shown in the console window. The console output is as follows:

```
RENDER!! > scene.scn
Execute Scene file:/Users/kioku/git/HIVE...
create camera 256 256 nil

Debug: FileName = output.jpg
RenderObjects Num = 2
RenderCore::RENDER!!!
```

The **Property** panel on the right shows the following table:

Property Name	Value
name	RenderToConsole
varname	renderConsole3
[RenderObject]	3
RenderObject0	(Object)
RenderObject1	(Object)
RenderObject2	(Object)

A blue callout box contains the following list of nodes:

- CreateCamera
- RenderToConsole
- TeapotGenerator
- CreatePolygonModel

SceneNodeEditorによる可視化例(2)

- **TeapotGenerator**ノード

- ・ 組み込みのTeapotモデルのメッシュデータを作成するノード

- **CreatePolygonModel**ノード

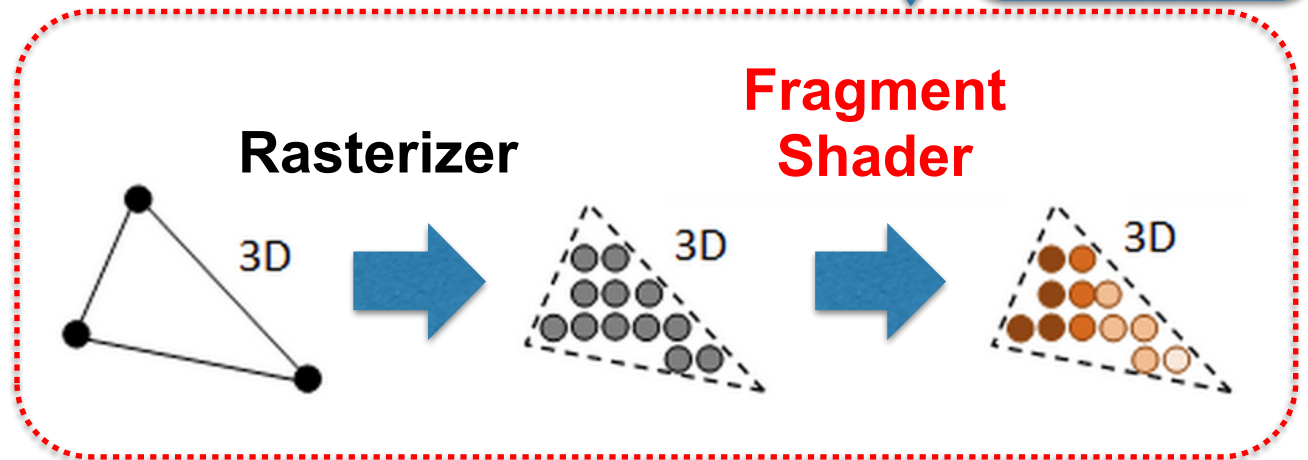
- ・ メッシュデータ表示用オブジェクトを作成するノード
- ・ Teapotのメッシュデータを接続し、表示する
- ・ 空間上の位置、シェーダファイルなどが指定可能

SceneNodeEditorによる可視化例(2)

- シェーダー(Shader)ファイルの設定
- CreatePolygonModelのノードで設定可能

最終画像を作成する処理部分を直接調整することが可能です。

Property Name	Value
name	CreatePolygonModel
varname	instPolygonModel6
mesh	(Object)
[translate]	0 0 0
[rotate]	0 0 0
[scale]	1 1 1
[shadername]	HIVE/hrender/test/white.frag
[Uniform]	3
Uniform0	(Object)
Uniform1	(Object)
Uniform2	(Object)



シェーダーファイルへのフルパス

または、NodeEditorからの相対パスを指

\$HIVE/hrender/test/
サンプルファイル

SceneNodeEditorによる可視化例(2)

- ・ HIVEのシェーダー
 - ・ **GLSL** (OpenGL Shading Language) の **Fragment Shader** (フラグメントシェーダー) に準拠

```
#ifdef GL_ES
precision mediump float;
#endif

void main(void)
{
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

white.frag



白 : (1, 1, 1, 1)

SceneNodeEditorによる可視化例(2)

- ・ \$HIVE/hrender/test/*.frag (用意されているシェーダー)
- ・ ユーザーが用途に合わせて作成したシェーダーを読み込むことが可能です。

```
#ifdef GL_ES
precision mediump float;
#endif

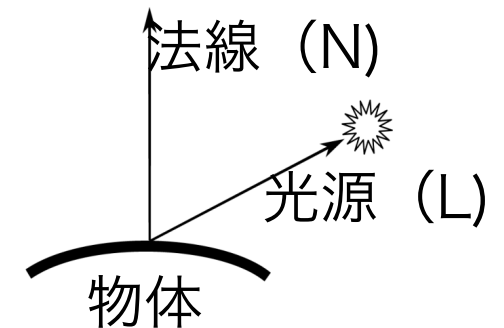
varying vec3  mnormal;

void main(void)
{
    vec3 lightdir = vec3(0.5, 1.0, 0.5);
    float shade = max(0.0, dot(lightdir, mnormal));
    gl_FragColor = vec4(shade, shade, shade, 1.0);
    return;
}
```

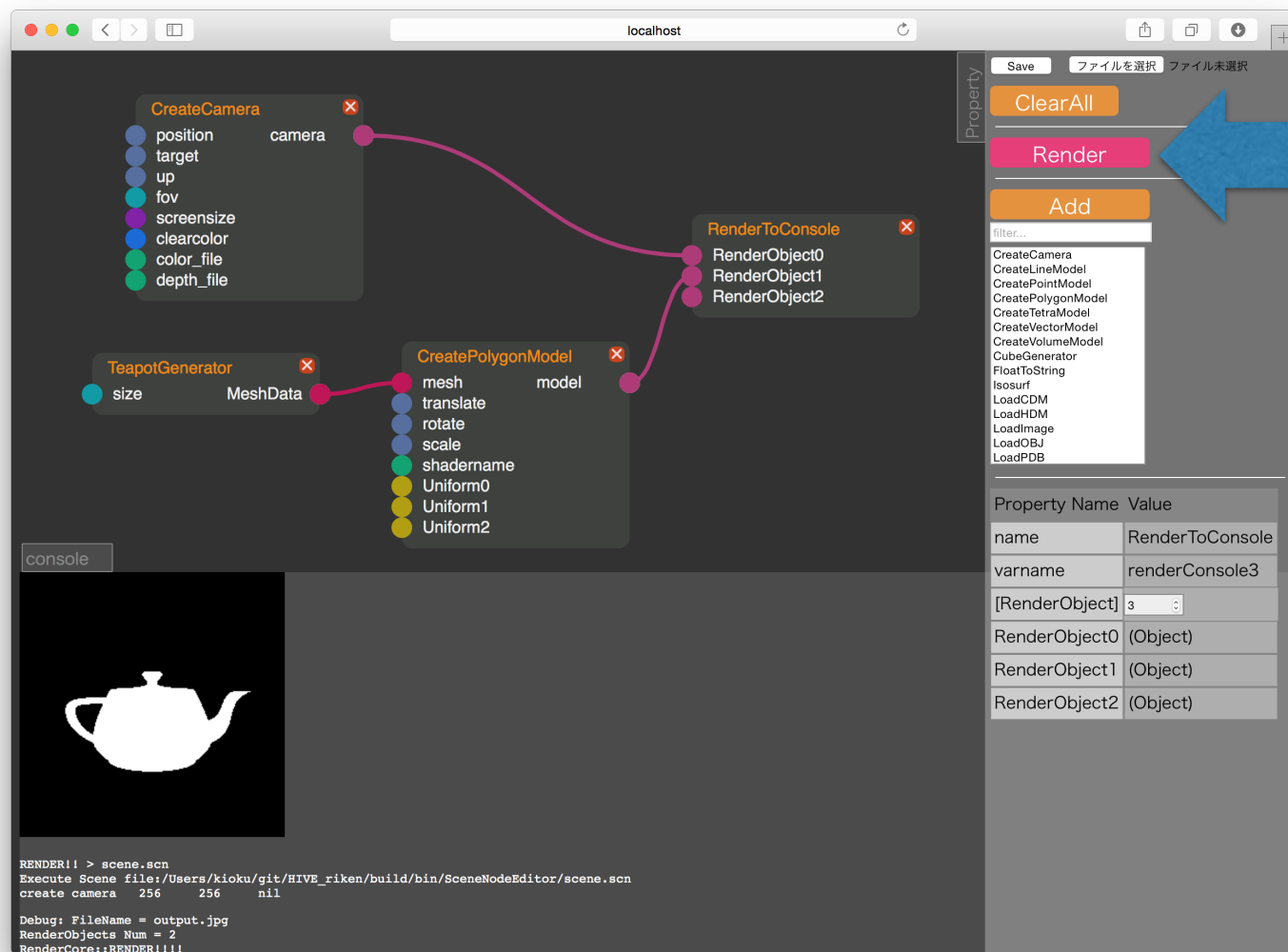
光源 (L)

法線 (N)

$N \cdot L$



シーンファイルの出力

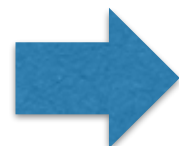


\$HIVE/build/bin/SceneNodeEditor
scene.scn

Lua言語の
スクリプトファイル



\$ hrender scene.scn

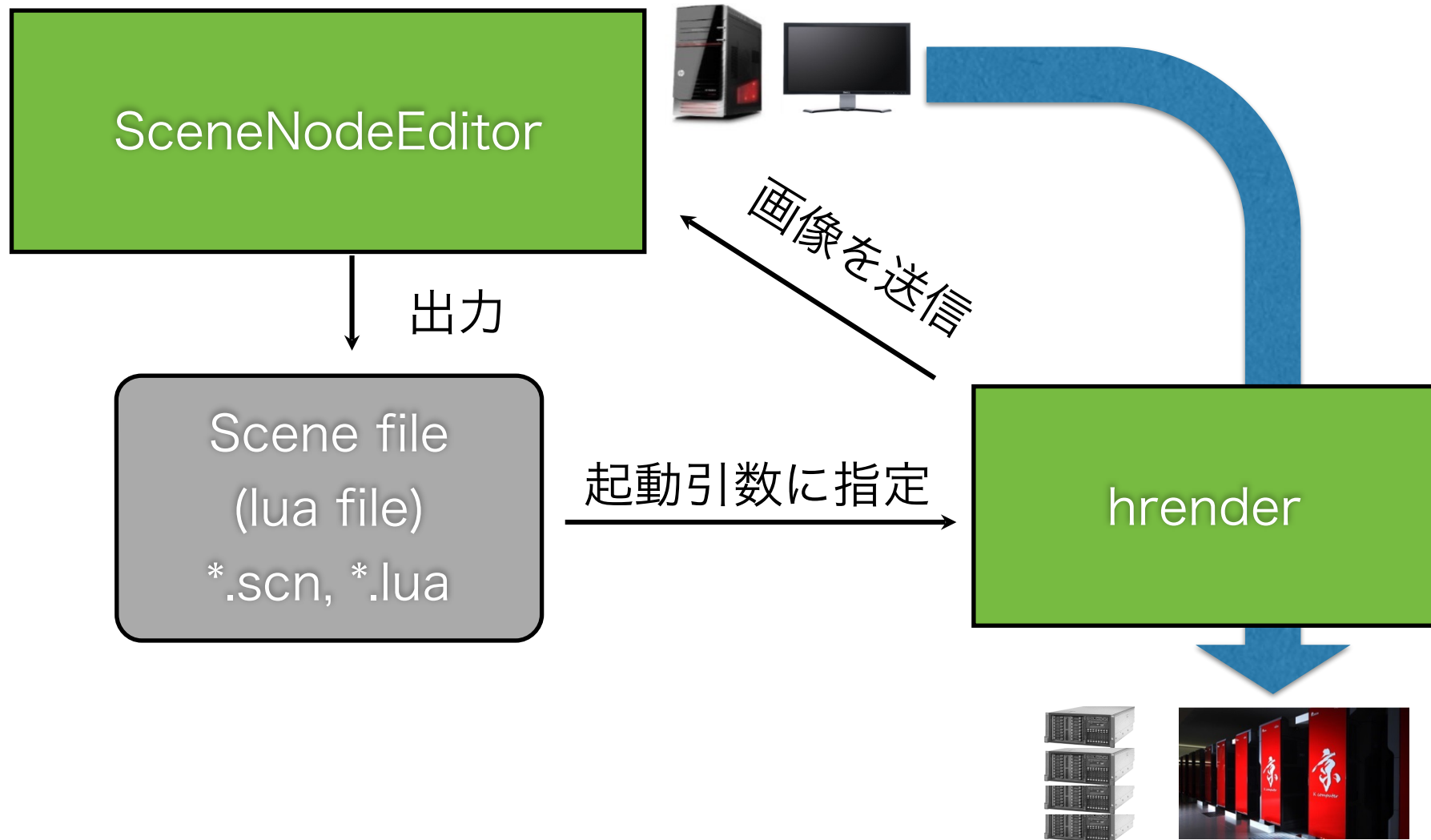


output.jpg



シーンファイル (Scene file)

Scene fileはポータブルな仕様のため、他の環境にコピーし、**hrender**で実行することが可能である



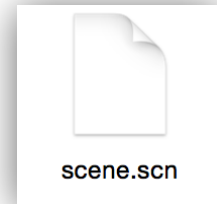
SceneNodeEditorのまとめ

- ・ SceneNodeEditorはScene fileの作成を補助するためのツールです。
- ・ script interfaceを習得することなく、直感的なScene fileの作成が可能となります。
- ・ 作成したScene fileはhrenderでのレンダリングが可能です。
- ・ NodeEditorで作成したScene fileをベースにしたシーン編集も可能です。

SceneNodeEditorで
作成したシーンの
hrenderでのレンダリング例

hrenderによるレンダリング(1)

- ・ 前述の可視化例実行後に、SceneNodeEditor配下に scene.scnというシーンファイルが作成されている



- ・ hrenderプログラムの起動引数に設定することでレンダリングの実行が可能である

- ・ **\$ hrender /path/to/scene.scn**

- ・ scene.scnのフォルダにoutput.jpgが作成されているはずで
す。

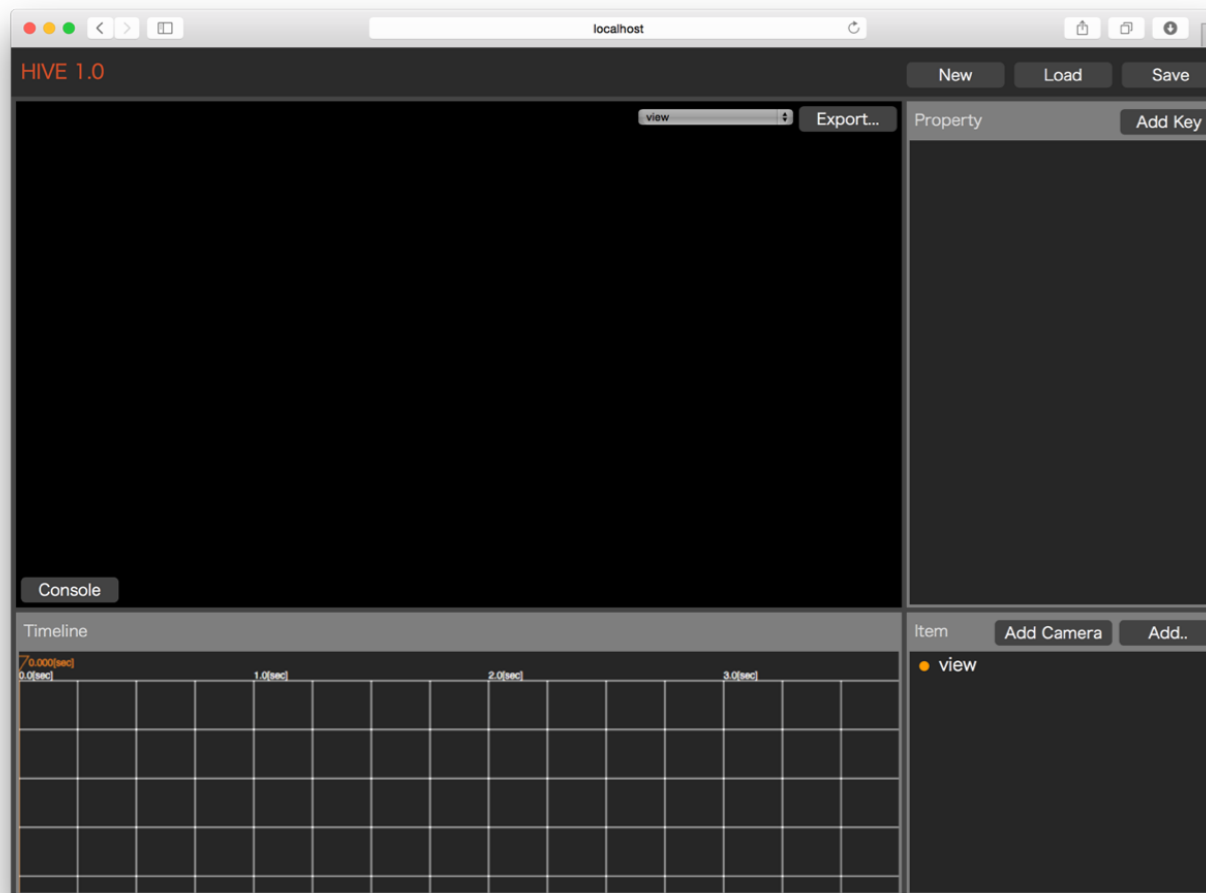
hrenderによるレンダリング(2)

- ・ 他の実行環境でのレンダリング
- ・ Scene fileはポータブルな仕様のため、他の環境にコピーし、hrenderで実行することが可能である。
- ・ 移行の際は、データファイルパスや、シェーダファイルパスをフルパスではなく、`scene.scn`ファイルからの相対パスになるように変更しておくのが望ましい
- ・ パスの設定はNodeEditorのプロパティ設定からでも、`scene.scn`ファイルの内容を直接変更してもどちらでもよい
- ・ 可視化のテスト時は、リダクションデータで行い、本番用に別データでレンダリングする際も、この方法が利用できる

HIVE UIについて

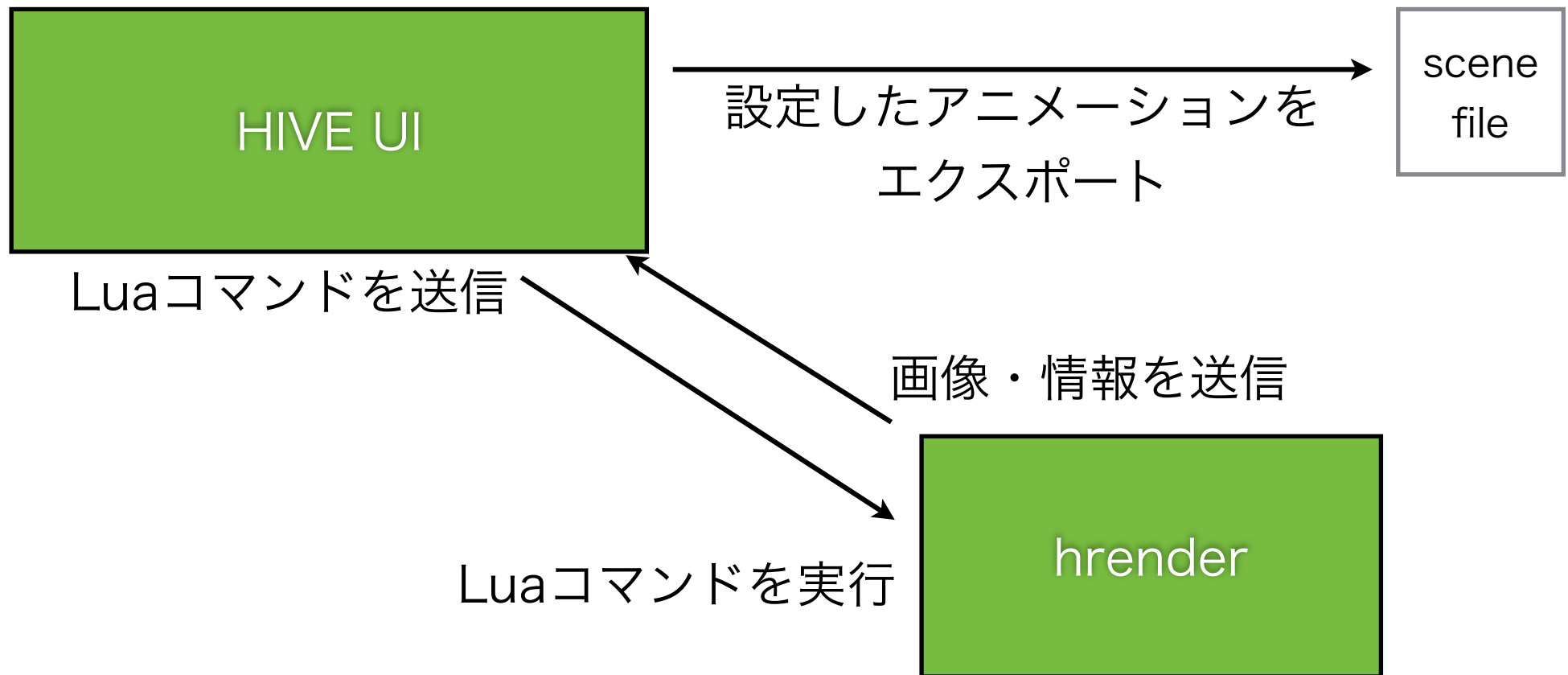
HIVE UI外観

HIVE UIはアニメーションシーン作成のための
Webブラウザベースアプリケーション



HIVE UI概要(1)

HIVE UIはScene Interfaceのコマンド(Lua)を送信するプログラム



HIVE UI概要(2)

- ・ HIVE UIはカメラアニメーション作成ツール
 - ・ キーフレームベースのアニメーションが作成可能
 - ・ 複数視点によるアニメーション作成が可能
 - ・ シェーダのパラメータを変更したアニメーションが作成可能
- ・ 作成したアニメーションをシーンファイルに出力し、hrenderでレンダリングが可能

HIVE UI利用説明

- ・ HIVE UIの利用方法については別資料を参照のこと
- ・ HIVE UI利用説明書

HIVE UI機能紹介

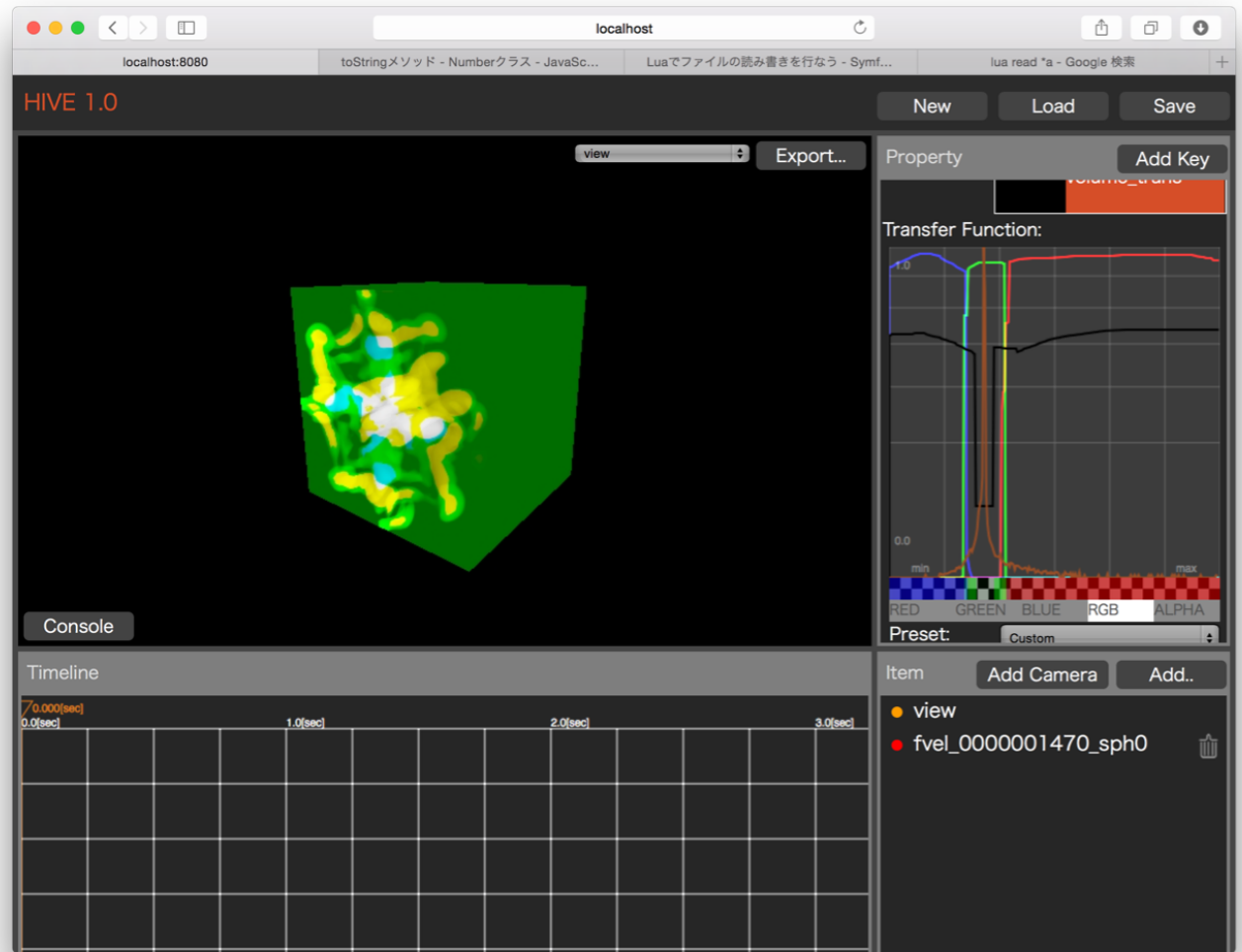
- ・ HIVE UIに実装されている便利な機能について紹介する。
 - ・ ボリュームレンダリングの伝達関数について
 - ・ マルチカメラの設定について
 - ・ HIVE UIでのムービー作成方法
 - ・ Scene Node EditorからHIVE UIへのシェーダの登録方法

ボリュームレンダリング
の伝達関数について

ボリュームレンダリングの伝達関数(1)

ボリュームシェーダの
パラメータとして、
任意の伝達関数
(Transfer function)が
設定可能。

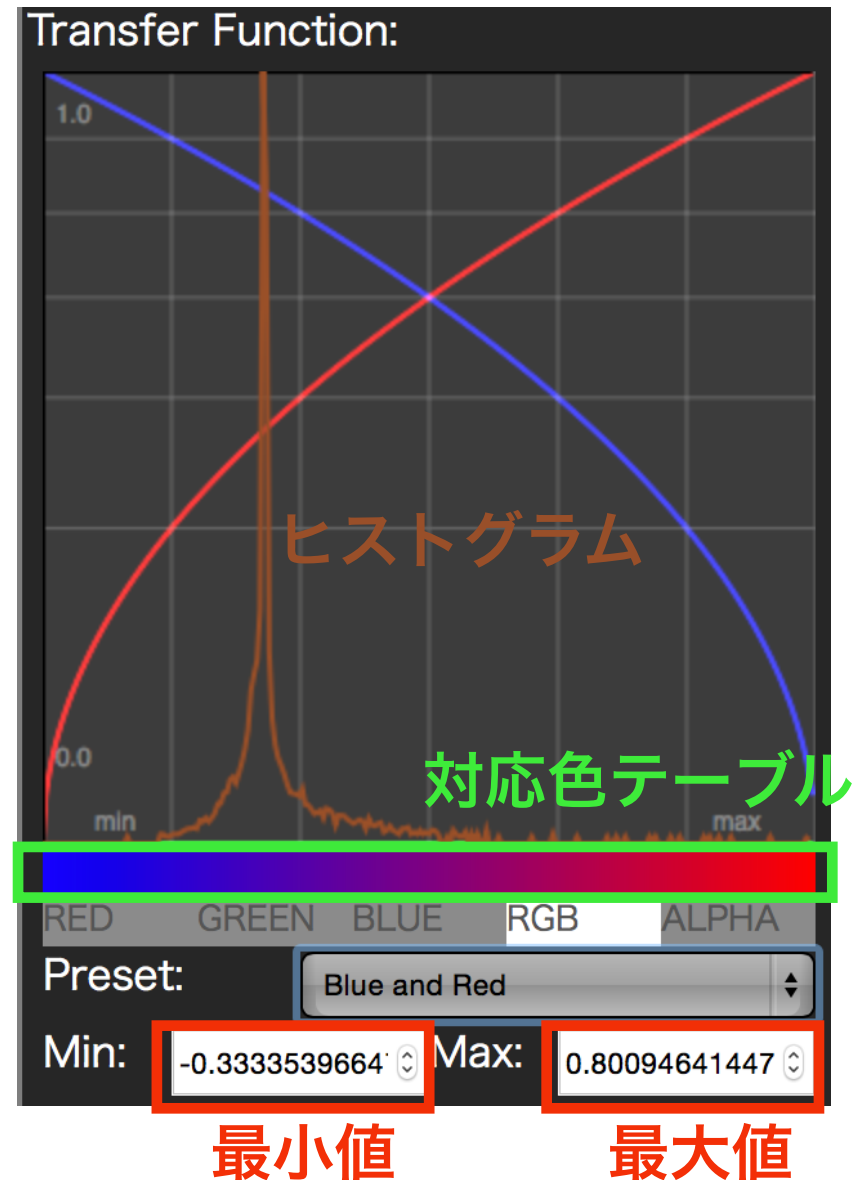
インタラクティブに
変更結果が確認可能



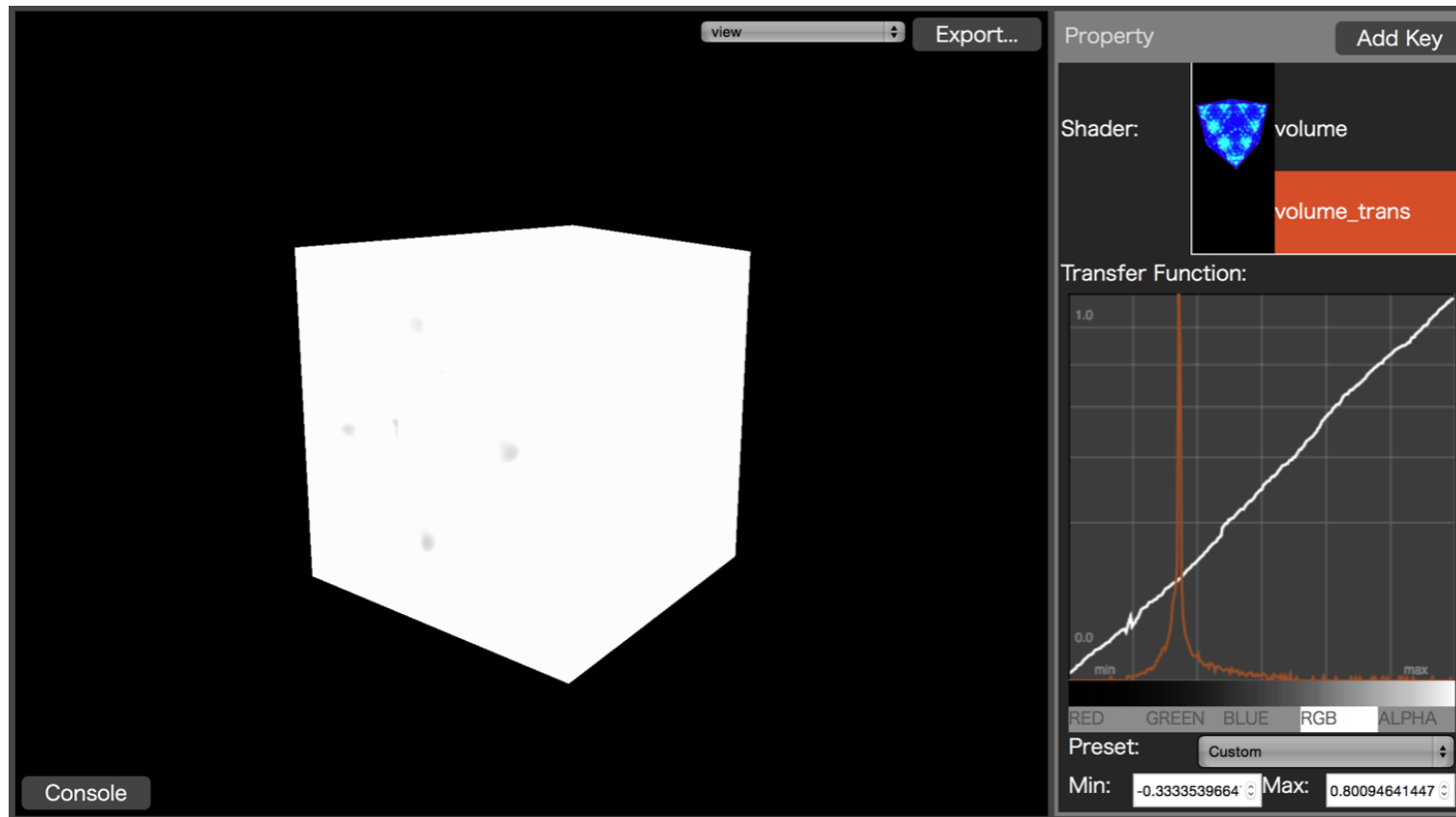
ボリュームレンダリングの伝達関数(2)

伝達関数(Transfer Function)のUIはボリュームデータの最小値、最大値を0.0 - 1.0として、その値を256段階に量子化し、対応する色テーブルを設定するものである。

また、UI中にボリュームデータのヒストグラムが表示してある。同じ値が多く含まれる場合はグラフの値が大きくなる

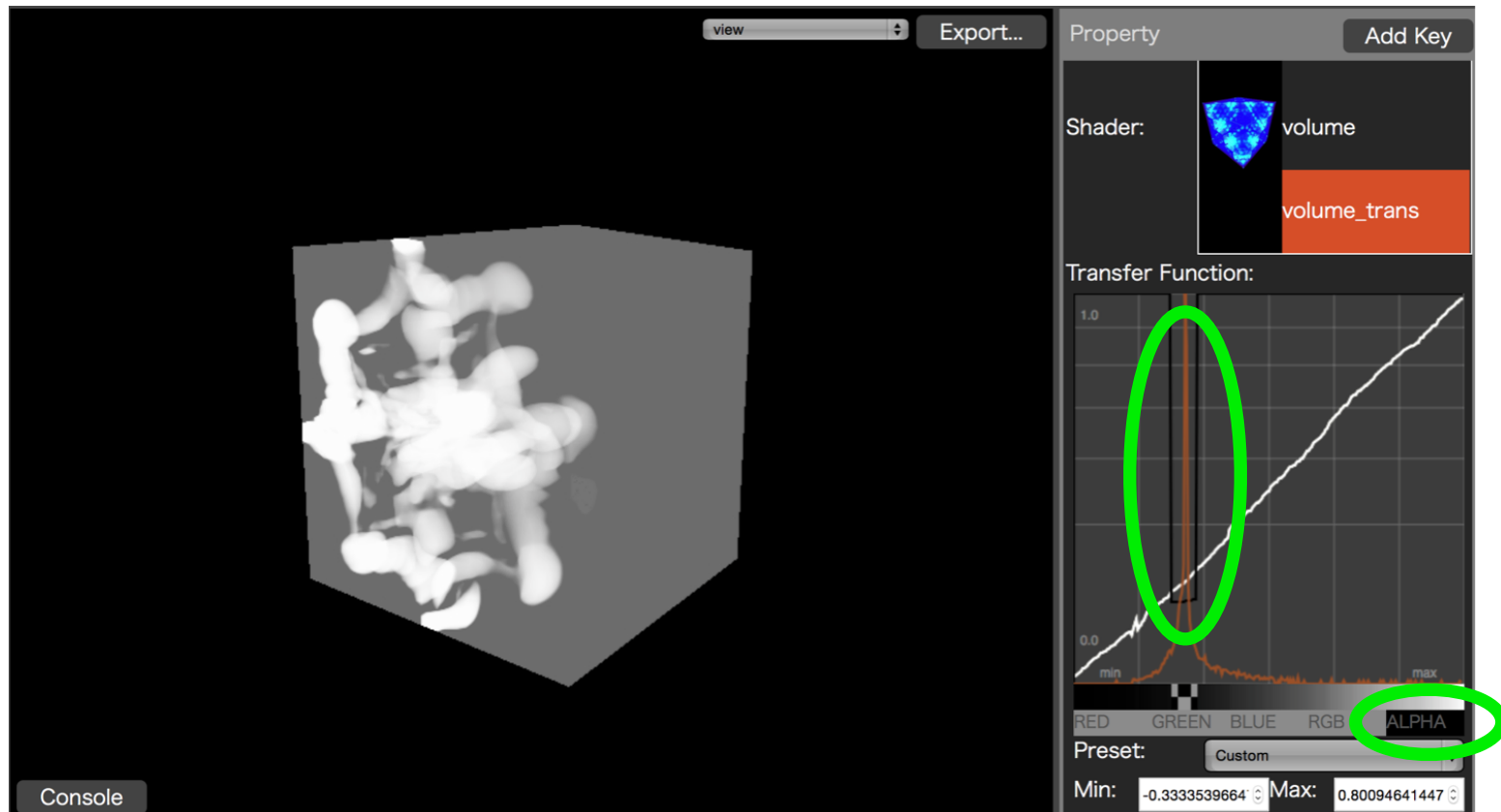


ボリュームレンダリングの伝達関数(3)



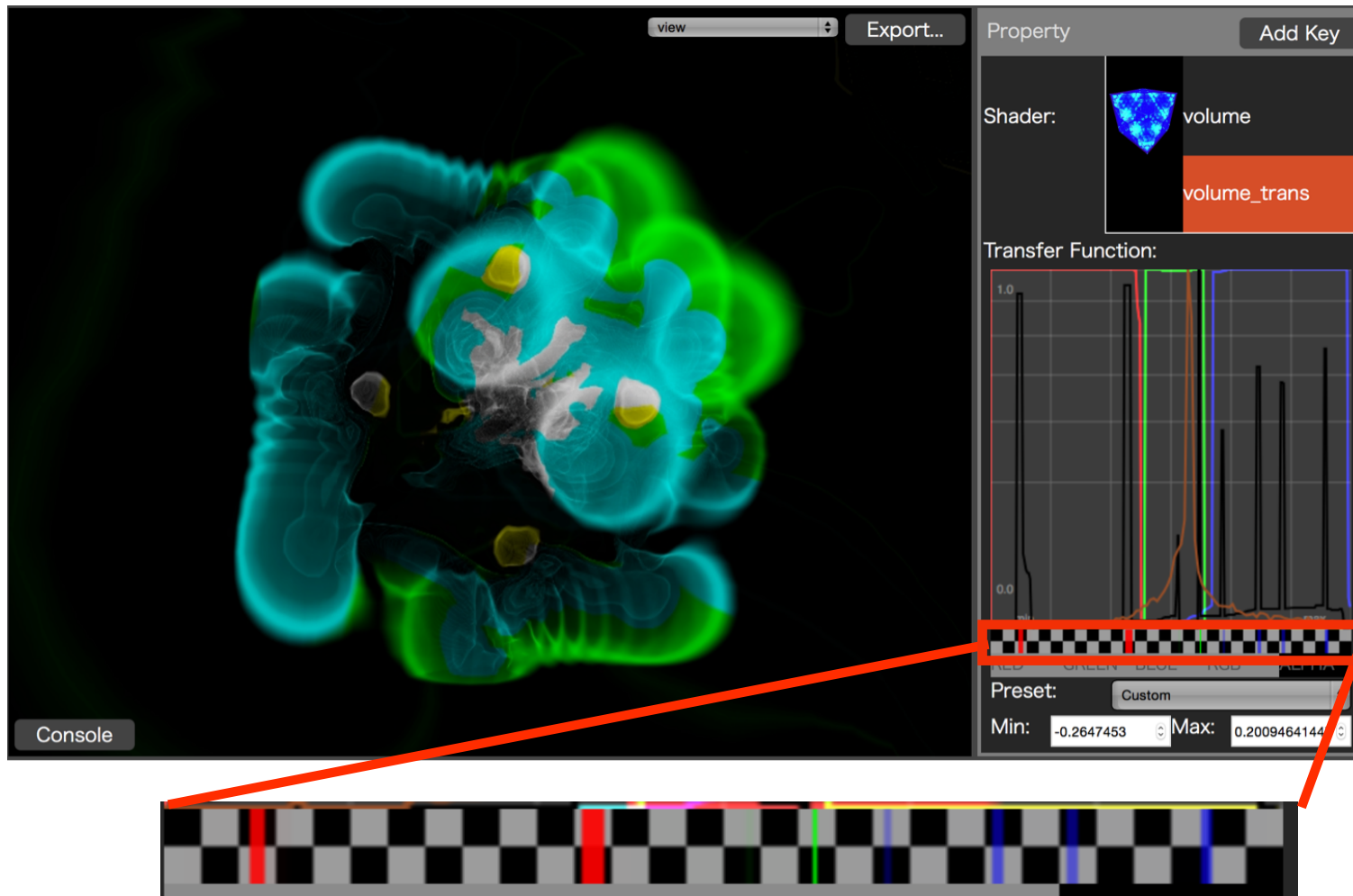
黒→白のグラデーションで表示しようとしているが
全体的に白飛びしている例

ボリュームレンダリングの伝達関数(4)



ヒストグラムのデータが多い部分の
アルファ値を低めに設定し、特徴を抜き出した例

ボリュームレンダリングの伝達関数(5)

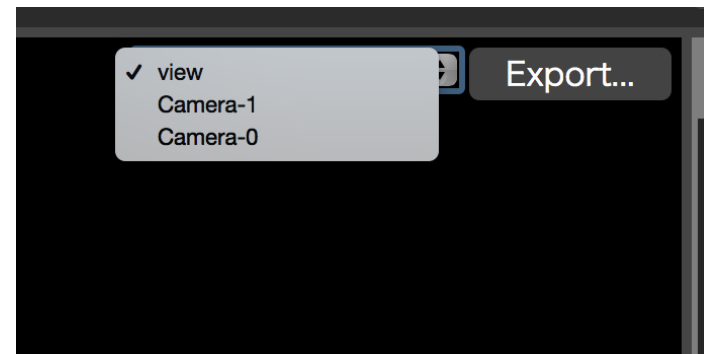
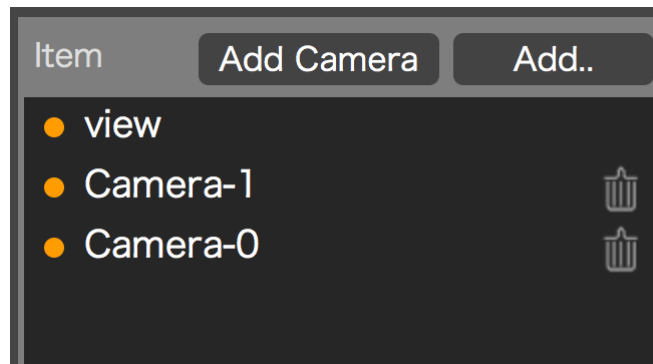


特定のデータ値を離散的に色付けしたレンダリング例

マルチカメラの 設定について

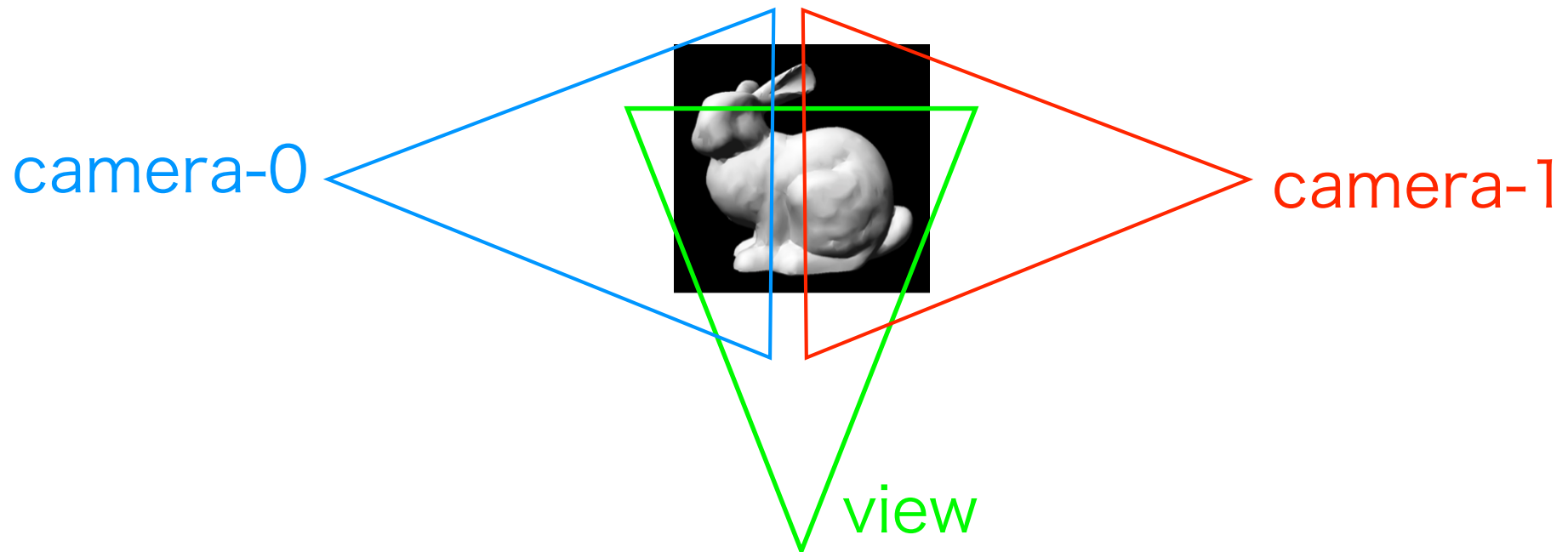
マルチカメラの設定(1)

- ・ HIVE UIでは複数のカメラ設定が可能である
- ・ 右下の「Add Camera」ボタンより複数のカメラを追加後、メインビューの右上よりカメラの選択(表示切り替え)が可能になる



マルチカメラの設定(2)

- ・ カメラの選択を行った後、ビューの操作をすることで、任意視点にカメラを設定できる。
- ・ この操作で以下のような配置を行った

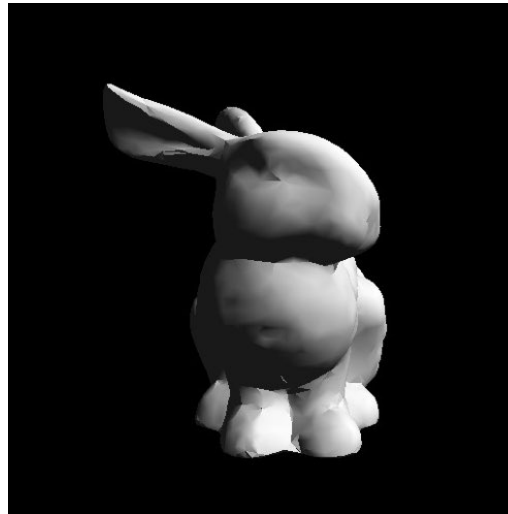


マルチカメラの設定(3)

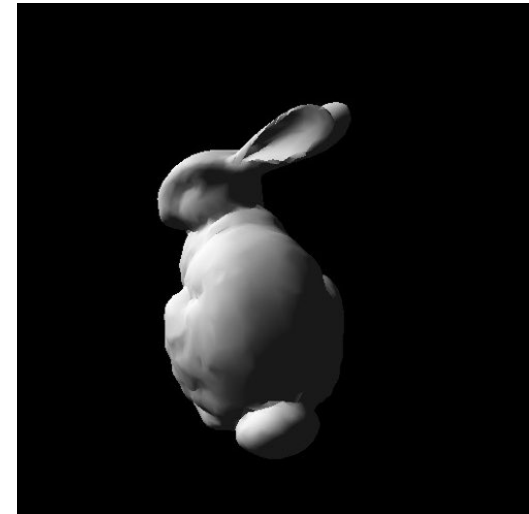
- 設定後、Export sceneでシーンのエクスポートを行い、hrenderでレンダリングすると以下の3つの画像が出力される



view.jpg



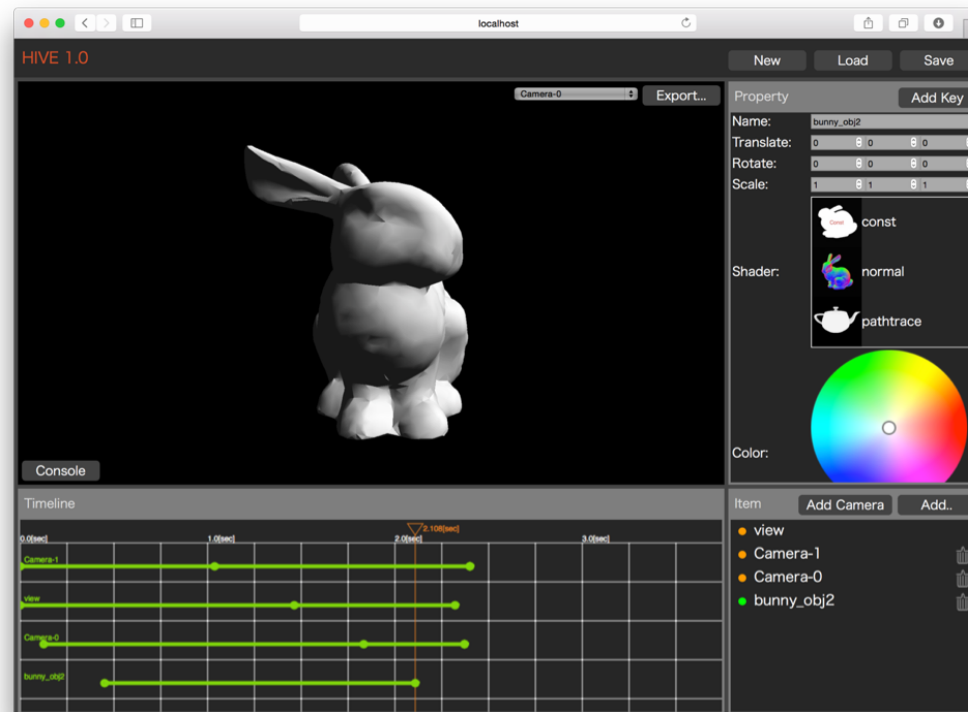
camera-0.jpg



camera-1.jpg

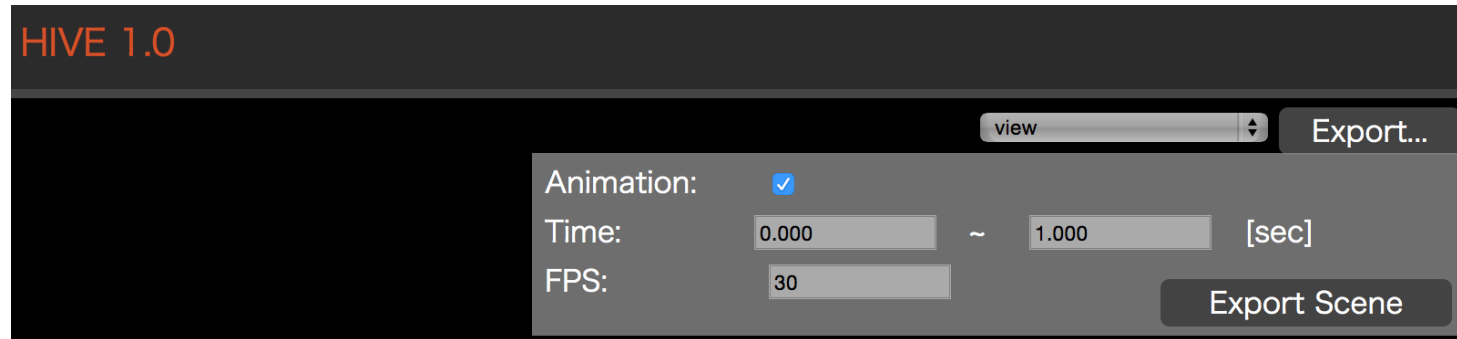
マルチカメラの設定(4)

- ・ それぞれのカメラは、カメラアニメーションも個別に設定できるため、同一データでの別アングルでのカメラアニメーション作成が可能となる。
- ・ 個別のカメラアニメーション設定については別資料を参照



HIVEUIでの ムービー作成方法

HIVE UIでのムービーの作成方法(1)



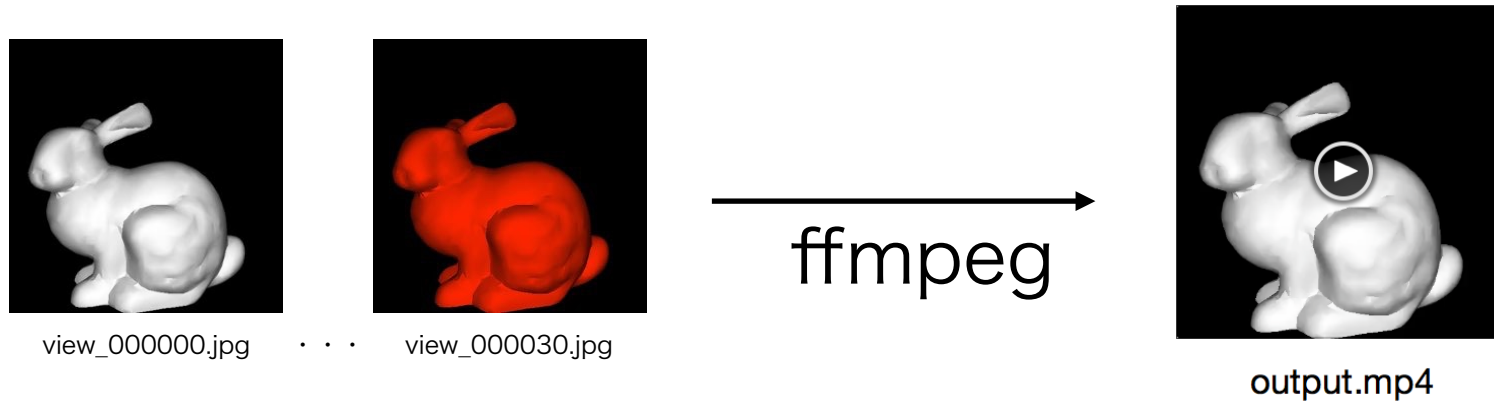
- HIVE UIではキーフレームアニメーションによる連番画像アニメーションのシーンファイルがエクスポート可能である。
- ここでは連番画像を出力し、ムービーを作成する方法を紹介する。

HIVE UIでのムービーの作成方法(2)



- ・ キーフレームアニメーションにシーンを作成し、連番アニメーションファイルを作成する
- ・ HIVE UIの操作方法についてはHIVE UIマニュアルを参照のこと

HIVE UIでのムービーの作成方法(3)



- ・ ffmpegによるムービーファイル作成
- ・ 以下URLからffmpegをダウンロード
- ・ <https://www.ffmpeg.org/download.html>

HIVE UIでのムービーの作成方法(4)

- ・ 以下のコマンドで変換
 - ・ `$ ffmpeg -r 30 -i view_%06d.jpg output.mp4`
- ・ FPSを指定 ここでは30fpsをつくるため“-r 30”と指定
- ・ 対象連番ファイルは view_000012.jpgのような数字が6桁のファイル名なので入力ファイルとして“-i view_%06d.jpg”という書式でファイル名を指定
- ・ 出力するムービーファイル名として最後に”output.mp4”を指定

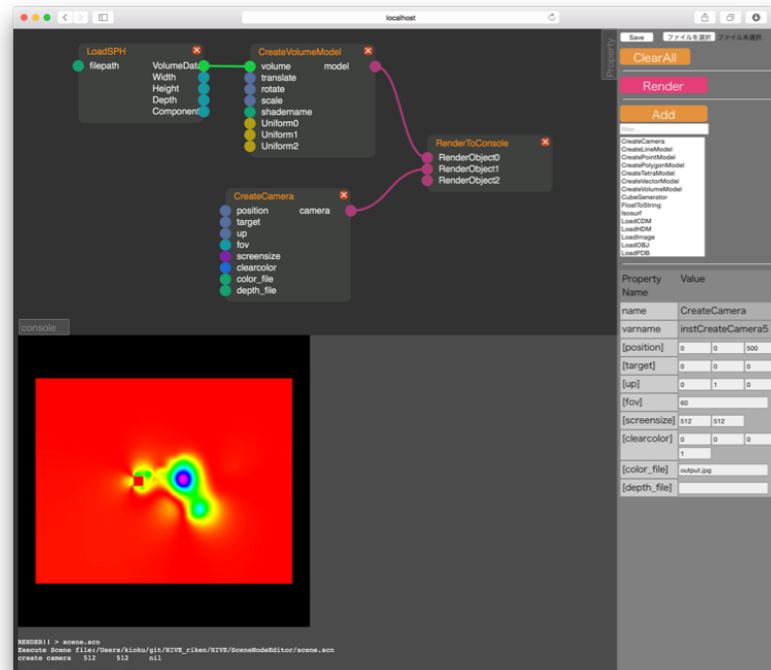
タイムステップデータ のムービー作成方法

タイムステップデータの ムービーの作成方法(1)

- ・ NodeEditorで作成したシーンデータからタイムステップデータのムービー作成方法を紹介します。
- ・ 例として
prs_00000000000.sph
prs_00000000199.sph
prs_00000000399.sph
...
prs_0000017399.sph
- ・ という200step刻みのデータがあるとします。

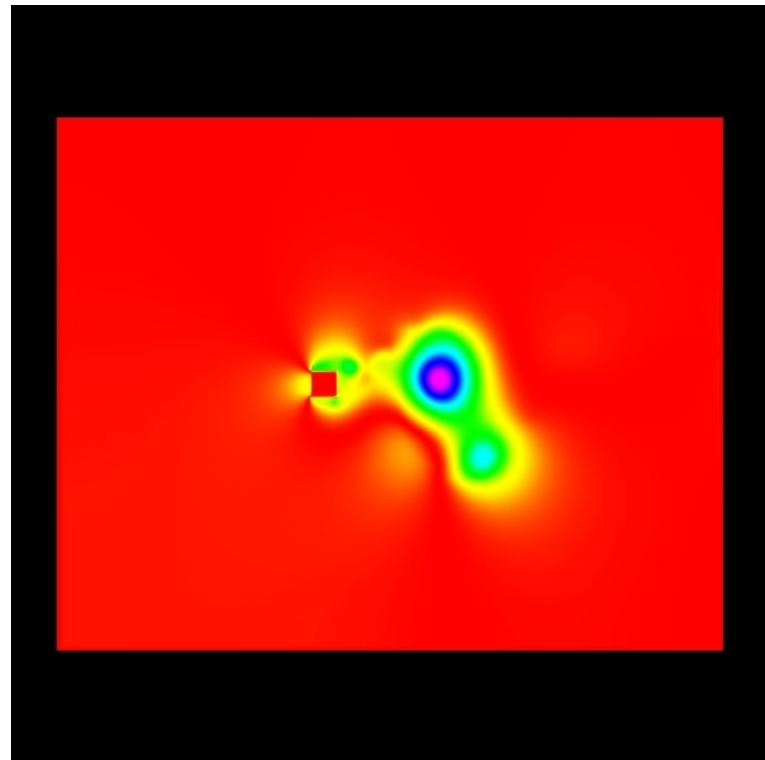
タイムステップデータのムービーの作成方法(2)

- まずNodeEditorを用いて、タイムステップデータの1step分のファイルを用いて、通常の可視化を行います。(prs_0000011999.sphを利用)



タイムステップデータの ムービーの作成方法(3)

- ・ まずNodeEditorによって作成されたscene.scnをhrenderで実行し、正しくレンダリング出来ているか確認します。
- ・ `$../hrender scene.scn`



タイムステップデータの ムービーの作成方法(4)

- ・ scene.scnファイルの中を確認してみましょう
ファイルの末尾に以下のような出力が確認できます。
- ・ filepath='/work/cyl2d/2d/prs_0000011999.sph' となっているところで、入力ファイルが設定されているのが確認できます。

```
local instCreateCamera5 = CreateCamera({position={0,0,500}, target={0,0,0}, up={0,1,0}, fov = 60, screensize={512,512},
clearcolor={0,0,0,1}, color_file='output.jpg', nil})
local instSPHLoader1 = LoadSPH({filepath='/work/cyl2d/2d/prs_0000011999.sph'})
local instVolumeModel2 = CreateVolumeModel({volume=instSPHLoader1:VolumeData(), translate={0,0,0}, rotate={0,0,0},
scale={1,1,1}, shadername='/work/cyl2d/heatmap.frag', Uniform={nil, nil, nil})
local renderConsole4 = RenderToConsole({RenderObject={RenderObject0=instVolumeModel2:model(),
RenderObject1=instCreateCamera5:camera(), nil})
-- Generated Footer by NodeEditor
```


タイムステップデータの ムービーの作成方法(5)

- ・ 入力ファイルをタイムステップの全時間に設定したいため、
以下のようにscene.scnを編集します。

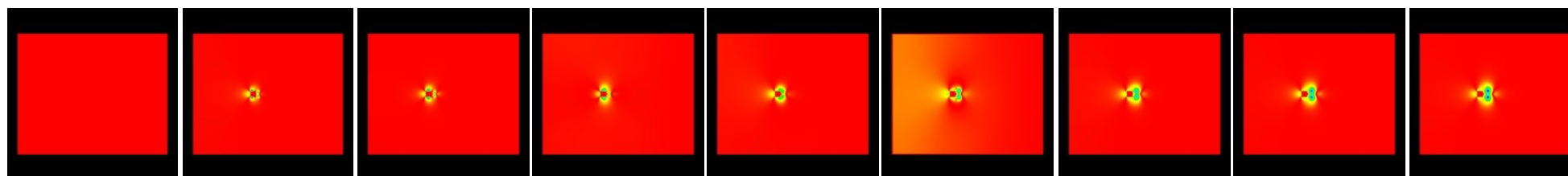
```
for i = 0, 17400, 200 do -- ループの設定
local tstep = math.max(i - 1, 0)
local timestepfile = string.format('/work/cyl2d/2d/prs_%010d.sph', tstep) --入力ファイル名
local outputfile = string.format('output_%010d.jpg', tstep) --出力ファイル名

local instCreateCamera5 = CreateCamera({position={0,0,500}, target={0,0,0}, up={0,1,0}, fov = 60, screensize={512,512},
clearcolor={0,0,0,1}, color_file=outputfile, nil})
local instSPHLoader1 = LoadSPH({filepath=timestepfile})
local instVolumeModel2 = CreateVolumeModel({volume=instSPHLoader1:VolumeData(), translate={0,0,0}, rotate={0,0,0},
scale={1,1,1}, shadername='/work/cyl2d/heatmap.frag', Uniform={nil, nil, nil})
local renderConsole4 = RenderToConsole({RenderObject={RenderObject0=instVolumeModel2:model(),
RenderObject1=instCreateCamera5:camera(), nil})

end
```

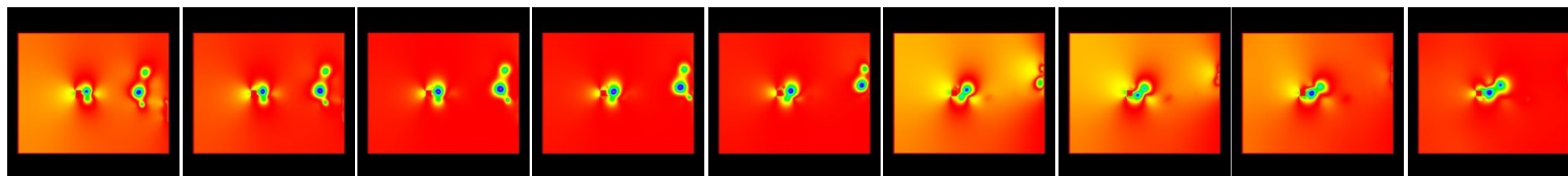
タイムステップデータの ムービーの作成方法(6)

- ・ hrenderでscene.scnをレンダリングすると、各入力ファイルに対応した、以下のファイル群が作成されるのが確認できます。



output_0000000000.jpg

...



output_0000017399.jpg

タイムステップデータの ムービーの作成方法(7)

- ・ ffmpegは連続番号の入力ファイルにしか対応しないため、出力ファイル名の部分を以下のように変更します。

```
local cnt = 0
```

```
for i = 0, 17400, 200 do -- ループの設定
```

```
local tstep = math.max(i - 1, 0)
```

```
local timestepfile = string.format('/work/cyl2d/2d/prs_%010d.sph', tstep) --入力ファイル名
```

```
local outputfile = string.format('output_%010d.jpg', cnt) --出力ファイル名
```

```
cnt = cnt + 1
```

```
local instCreateCamera5 = CreateCamera({position={0,0,500}, target={0,0,0}, up={0,1,0}, fov = 60, screensize={512,512},  
clearcolor={0,0,0,1}, color_file=outputfile, nil})
```

```
local instSPHLoader1 = LoadSPH({filepath=timestepfile})
```

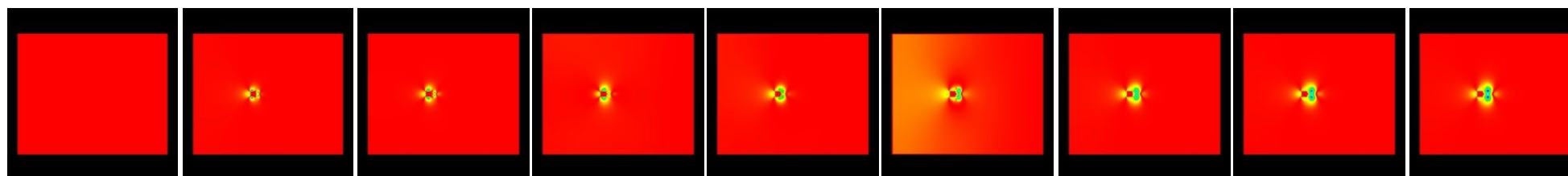
```
local instVolumeModel2 = CreateVolumeModel({volume=instSPHLoader1:VolumeData(), translate={0,0,0}, rotate={0,0,0},  
scale={1,1,1}, shadername='/work/cyl2d/heatmap.frag', Uniform={nil, nil, nil}})
```

```
local renderConsole4 = RenderToConsole({RenderObject={RenderObject0=instVolumeModel2:model(),  
RenderObject1=instCreateCamera5:camera(), nil}})
```

```
end
```

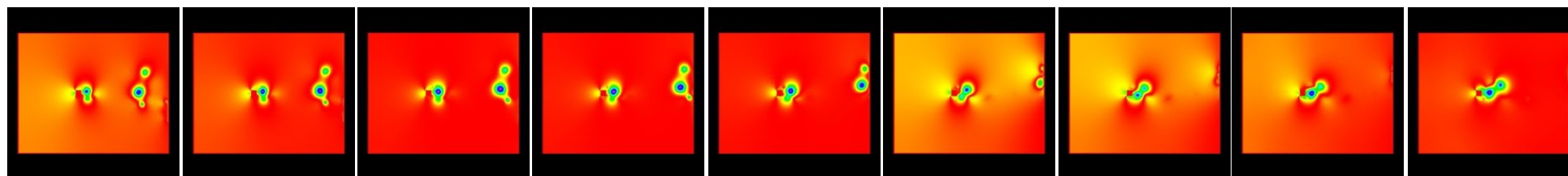
タイムステップデータの ムービーの作成方法(8)

- ・ hrenderでscene.scnをレンダリングすると、各入力ファイルに対応した、以下のファイル群が作成されるのが確認できます。



output_0000000000.jpg

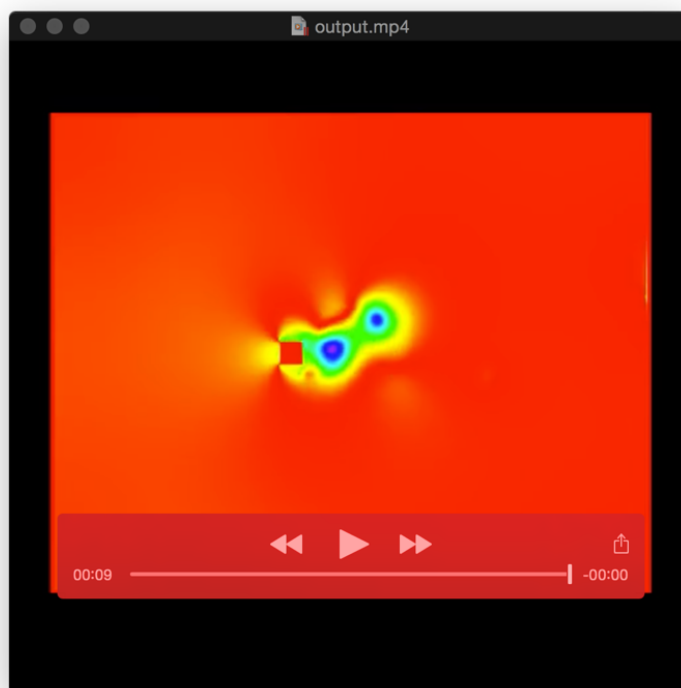
...



output_0000000087.jpg

タイムステップデータの ムービーの作成方法(9)

- ・ 以下のffmpegコマンドでムービーに変換します
 - ・ `$ ffmpeg -r 10 -i output_%010d.jpg output.mp4`
- ・ 今回は10FPSを指定



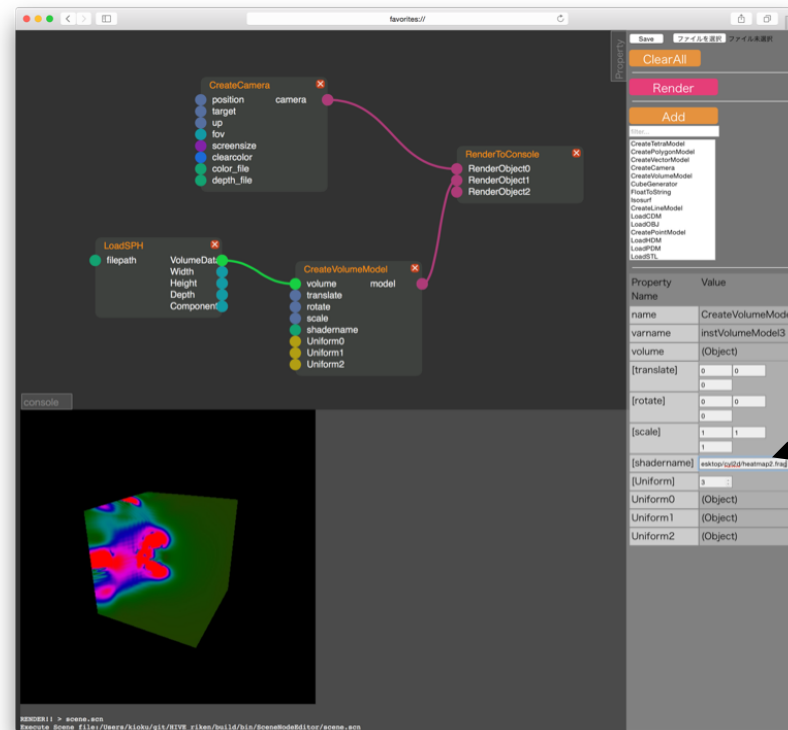
HIVE UIへの独自 シェーダ登録方法

HIVE UIへのシェーダの登録(1)

- ・ NodeEditorで作成したシェーダをHIVE UIにて利用する方法を紹介します

HIVE UIへのシェーダの登録(2)

- NodeEditorにて独自のシェーダを作成、設定し、レンダリングを行えるようにします。



独自シェーダの
パスを設定

HIVE UIへのシェーダの登録(3)

- ・ シェーダからパラメータにしたい変数をuniform変数に設定します。

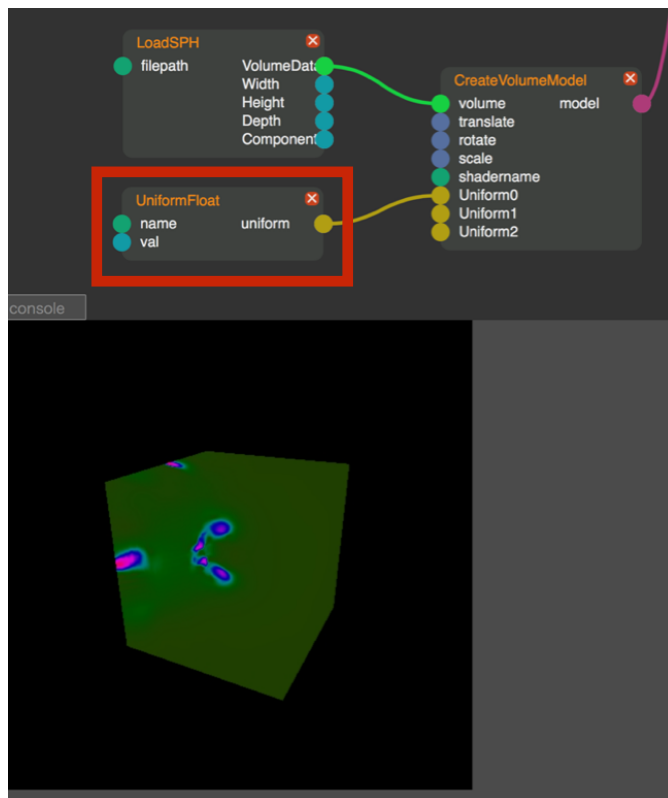
```
vec4 samplingVolume(vec3 texpos, vec4 sum) {  
    vec4 dens = vec4(texture3D(tex0, texpos).rgb, 1.0);  
    float r = length(dens.xyz) * 100.0;  
    r = 1.0 / pow(1.0 + exp(-r), 2.0);  
    vec4 col = vec4(hsv2rgb_smooth(vec3(r, 1.0, 1.0)), r);  
    return col; // add to sum  
}
```



```
uniform float lenscale;  
  
vec4 samplingVolume(vec3 texpos, vec4 sum) {  
    vec4 dens = vec4(texture3D(tex0, texpos).rgb, 1.0);  
    float r = length(dens.xyz) * lenscale;  
    r = 1.0 / pow(1.0 + exp(-r), 2.0);  
    vec4 col = vec4(hsv2rgb_smooth(vec3(r, 1.0, 1.0)), r);  
    return col; // add to sum  
}
```

HIVE UIへのシェーダの登録(4)

- uniform floatノードを追加し、設定したuniform変数と値を設定しVolumeのノードに接続します。

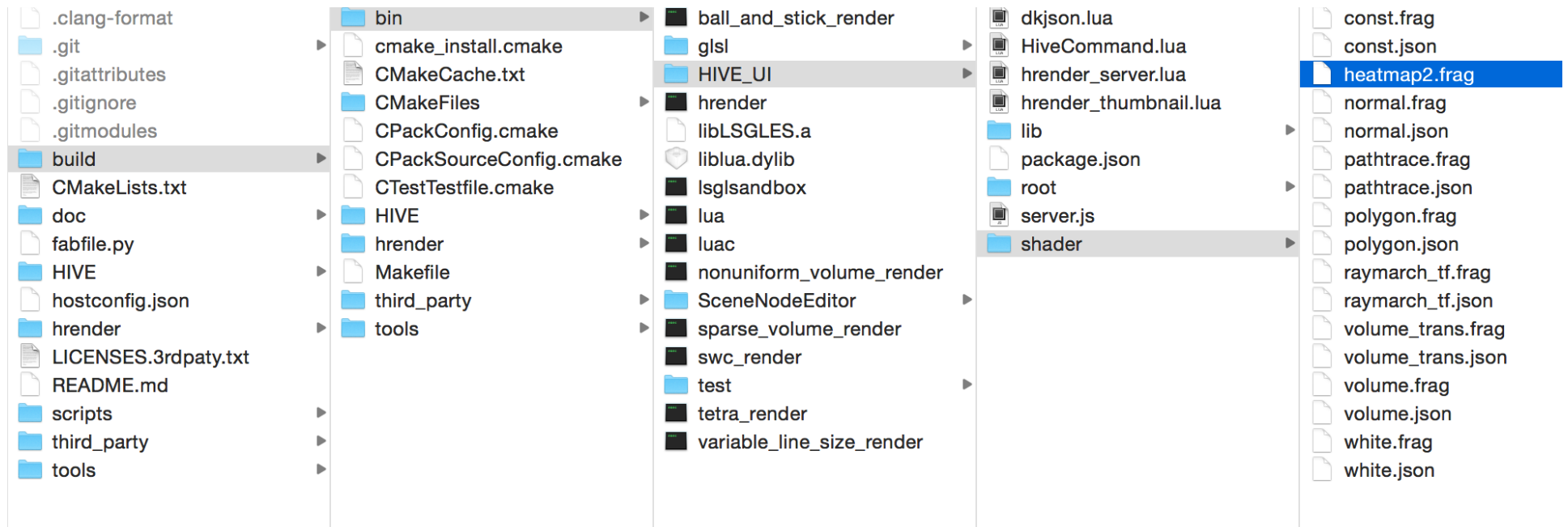


Property Name	Value
name	UniformFloat
varname	instUniformFloat1
[name]	lenscale
[val]	10

値を変更すると結果画像が変わることが確認できます。

HIVE UIへのシェーダの登録(5)

- 作成したshaderをHIVE_UI/shaderフォルダにコピーします。

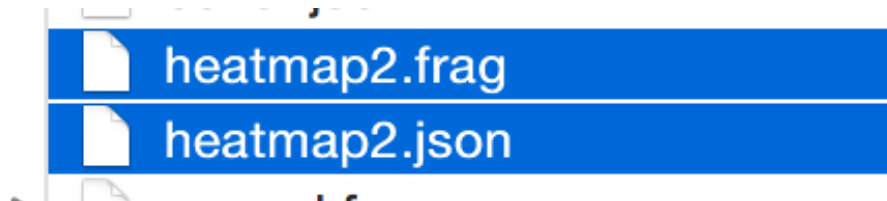


HIVE UIへのシェーダの登録(6)

- 作成したshaderと同一の名前.jsonファイル(今回はheatmap2.fragなのでheatmap2.json)を作成し、中身を以下のように設定します。

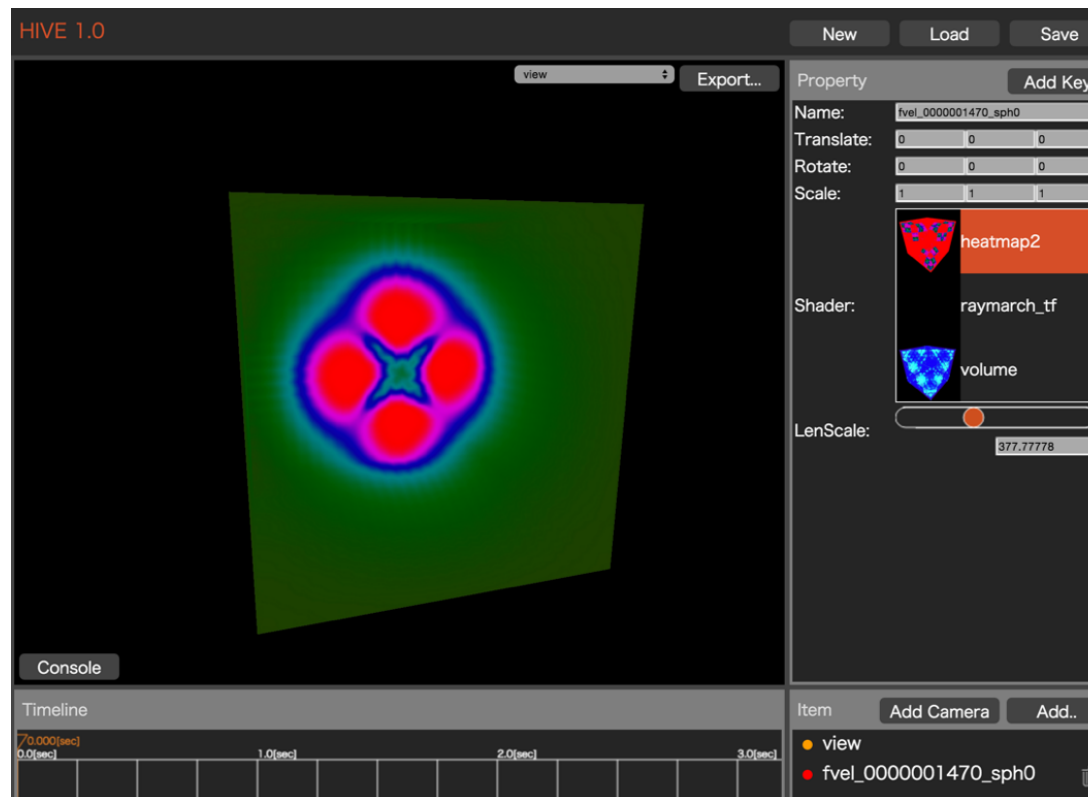
- heatmap2.json

```
{
  "type": "VOLUME",
  "uniforms": [
    { "label": "LenScale", "name": "lenscale", "ui": "slider", "uniform": "float", "val": 1.0, "max": 1000.0, "min": 0.0 }
  ]
}
```



HIVE UIへのシェーダの登録(7)

- HIVE UIを起動し、VOLUMEデータをロードすると先ほど設定したシェーダが選択できようになっており、設定したパラメータが設定できるのが確認できます。

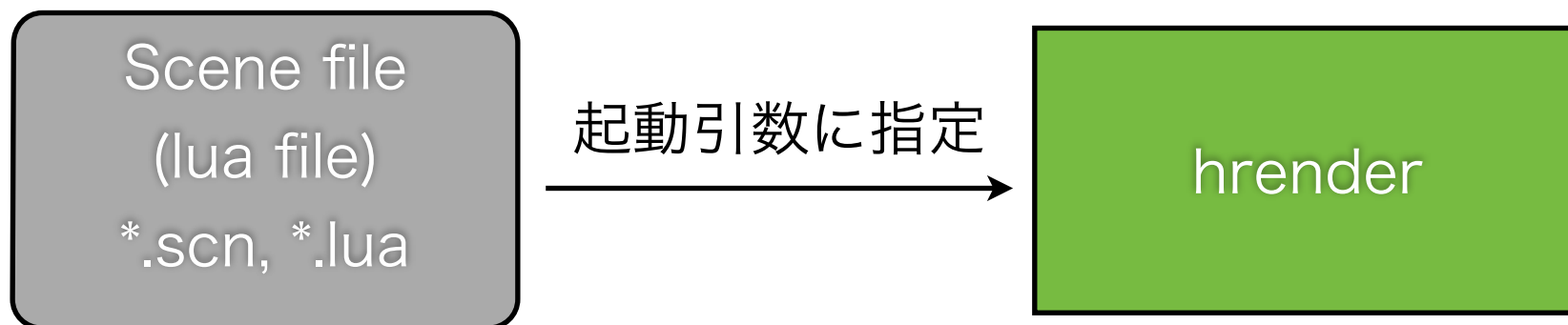


hrenderについて

hrender概要(1)



hrenderはLuaベースのコンソールプログラム



sceneファイルはテキストエディタで編集可能。
SceneNodeEditorを使うとGUIで作成可能。

Luaの実行環境と同等の動作
+
可視化のための関数セット (Scene Script Interface)

Lua

- 軽量なスクリプト言語、高速動作
- 高い拡張性
- Cのみでコンパイル可
 - ポータビリティ、Cとの親和性
- グルー言語
 - コンポーネント間をつなぐワークフロー

hrender概要(2)

- ・ luaと同じ動作をします。

```
$ cat test.lua
```

```
print('hello, world')
```

```
$ lua test.lua
```

```
hello, world
```

```
$ ./hrender test.lua
```

```
RENDER!! > test.lua
```

```
Execute Scene file:~/work/HIVE/build/bin/test.lua
```

```
hello, world
```

```
Exit hrender.
```

Luaの実行環境と同等の動作

```
print('hello, world')
```

```
test.lua  
(Luaスクリプトファイル)
```

```
$ lua  
test.lua  
hello, world
```

Lua

hrender

```
$ ./hrender test.lua  
RENDER!! > test.lua  
Execute Scene file:~/work/HIVE/build/bin/  
test.lua  
hello, world  
Exit hrender.
```

可視化のための関数セット

test.lua

```
local camera = Camera()
camera:SetScreenSize(512,512)
camera:SetFilename('output.jpg')
render({camera})
```

hrender

output.jpg

```
$ ./hrender test.lua
RENDER!! > test.lua
Execute Scene file:~/work/HIVE/build/bin/
test.lua
Debug: FileName = output.jpg
...
Save:output.jpg
Exit hrender.
```

hrenderによるレンダリング

- **\$ hrender /path/to/scene.scn**
- 他の実行環境でのレンダリング
 - 移行の際は、データファイルパスや、シェーダファイルパスをフルパスではなく、scene.scnファイルからの相対パスになるように変更しておくのが望ましい
 - パスの設定はNodeEditorのプロパティ設定からでも、scene.scnファイルの内容を直接変更してもどちらでもよい
 - 可視化のテスト時は、リダクションデータで行い、本番用に別データでレンダリングする際も、この方法が利用できる

Scene Script Interface

- https://github.com/avr-aics-riken/HIVE/blob/master/doc/scene_script_interface.md

Scene Script Interface

Basic function

render()

指定されたRenderObjectでレンダリングを行う

```
render({renderobject1, renderobject2, renderobject3 })
```

第2引数にレンダリング中のコールバック関数を指定可能

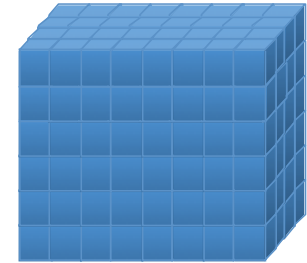
```
render({renderobject1, renderobject2, renderobject3 }, function (progress)
  print('Progress = ' .. progress)
end)
```

ファイル/0
(読み込みと書き込み)
について

HIVEで対応しているデータフォーマット

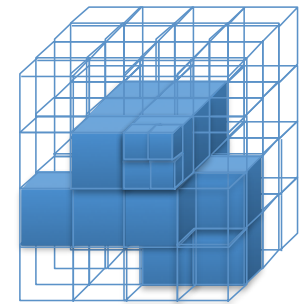
- 一様ボリューム (SPH, VOL, RAW)

CDMLib



- 非一様ボリューム (XYZ)

CDMLib



- PDB (Protein Data Bank 形式)

- 粒子データ (GEO)

PDMlib

- 階層一様ボリューム (BCM, PVTI)

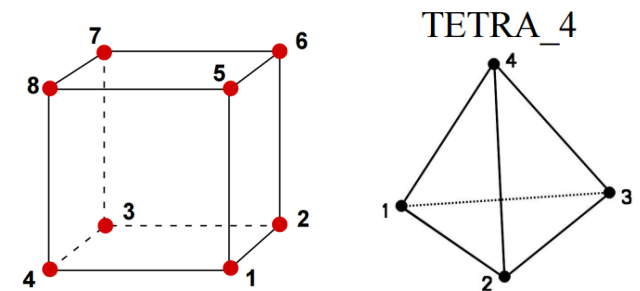
HDMLib

- 非構造データ (CGNS)

UDMLib

- ポリゴンデータ (STL)

```
HEADER  EXTRACELLULAR MATRIX 22-JAN-98 1A3I
TITLE   X-RAY CRYSTALLOGRAPHIC DETERMINATION OF A COLLAGEN-LIKE
        2 PEPTIDE WITH THE REPEATING SEQUENCE (PRO-PRO-GLY)
...
EXPDTA  X-RAY DIFFRACTION
AUTHOR  R.Z.KRAMER,L.VITAGLIANO,J.BELLA,R.BERISIO,L.MAZZARELLA,
        2.B.BRODSKY,A.ZAGARI,H.M.BERMAN
...
REMARK 350 BIOMOLECULE: 1
REMARK 350 APPLY THE FOLLOWING TO CHAINS: A, B, C
REMARK 350 BIOM1  1  1.000000  0.000000  0.000000  0.000000
REMARK 350 BIOM2  1  0.000000  1.000000  0.000000  0.000000
...
SEQUES  1 A  9  PRO PRO GLY PRO PRO GLY PRO PRO GLY
SEQUES  1 B  6  PRO PRO GLY PRO PRO GLY
SEQUES  1 C  6  PRO PRO GLY PRO PRO GLY
...
ATOM    1  N  PRO  A  1      8.316  21.206  21.530  1.00  17.44  N
ATOM    2  CA  PRO  A  1      7.808  20.729  20.398  1.00  17.44  C
ATOM    3  C  PRO  A  1      8.487  20.707  19.192  1.00  17.44  C
ATOM    4  O  PRO  A  1      9.489  21.467  19.006  1.00  17.44  O
ATOM    5  CB  PRO  A  1      6.480  21.723  20.211  1.00  22.28  C
...
HETATM 130  C  ACY   401     3.682  22.541  11.238  1.00  21.19  C
HETATM 131  O  ACY   401     2.807  23.097  10.559  1.00  21.19  O
HETATM 132  OXT ACY   401     4.306  23.101  12.291  1.00  21.19  O
...
```



一様ボリューム

- ・ **SPH**

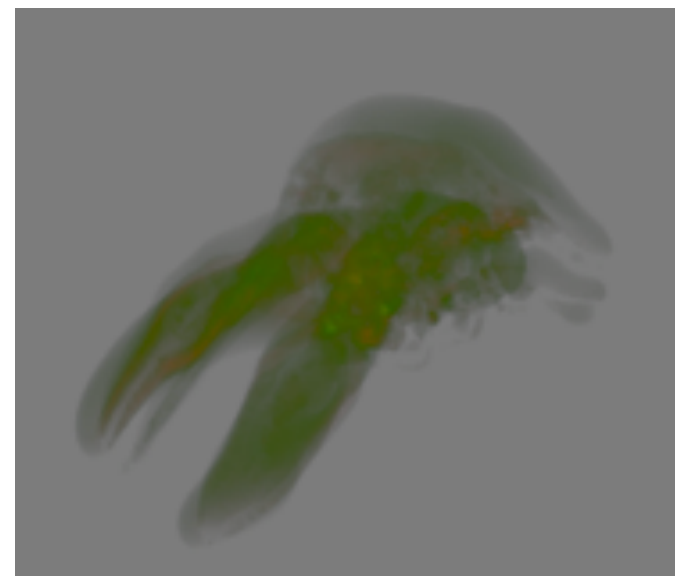
- ・ SPHLoader, SPHSaver

- ・ **VOL (HIVE 独自)**

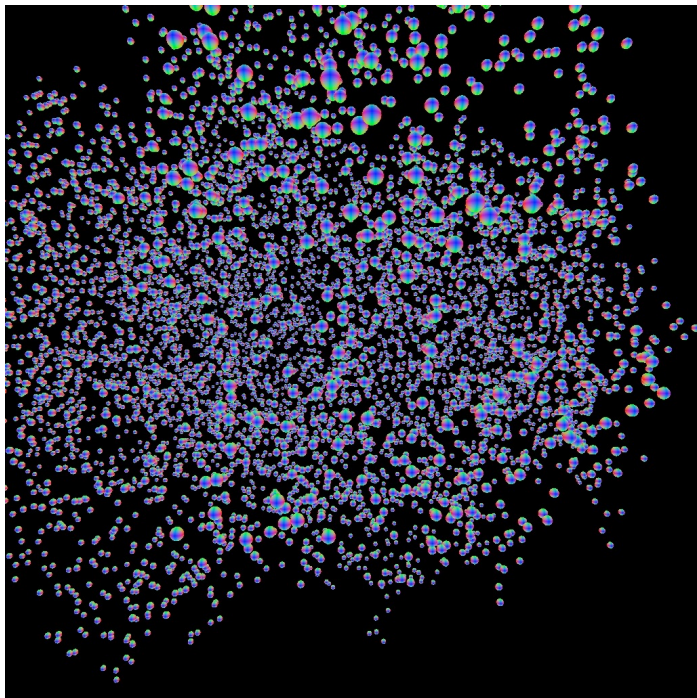
- ・ VOLLoader (書き込みは RawVolumeSaver を利用)

- ・ **RAW (生ボリュームデータ)**

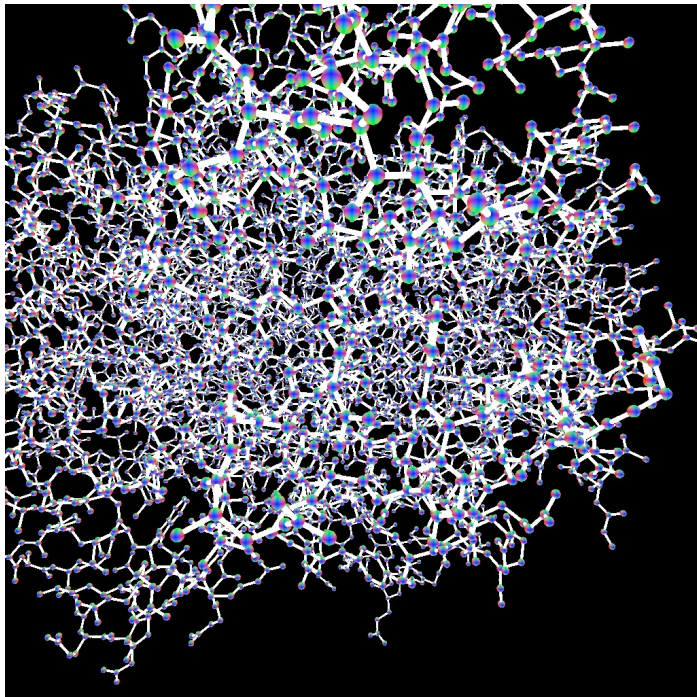
- ・ RawVolumeLoader, RawVolumeSaver
- ・ の読み込みと書き込みに対応しています.
- ・ float 型のみに対応になります.



PDB



ball(atom)



ball and stick(atom&bond)

- ・ PDBLoader: PDB(Protein Data Bank)形式のロードに対応しています.
- ・ Atom を粒子, Bond をラインとして描画できます.
- ・ Bond は Atom 間のファンデルワールス距離関係からローダ側で自動で生成します
- ・ Bond 計算には, 空間データ構造を構築するため, 100 万を超える Atom 数でもスケラブルに Bond を構築することができます
- ・ Atom 番号 99999 を超える Atom 行をパースすることができます(PDB 仕様外であるが, 拡張対応しています)

読み込みサンプルシーンファイル

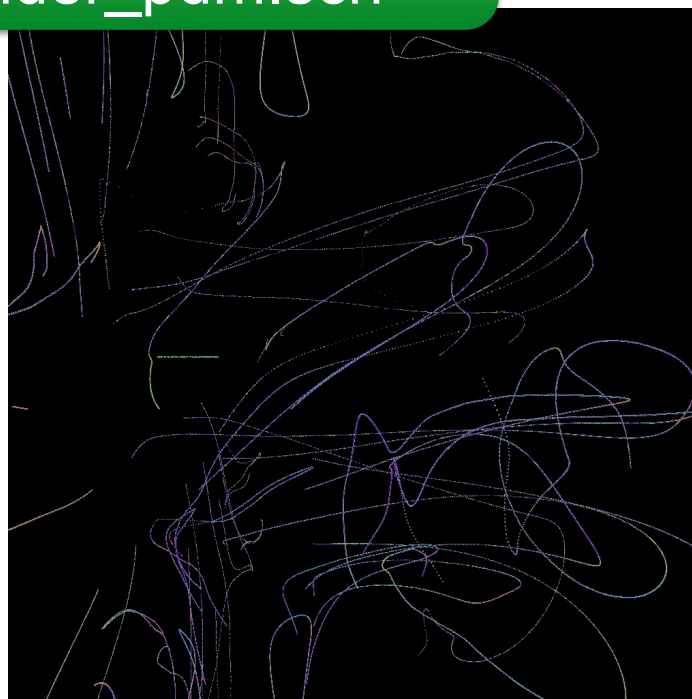
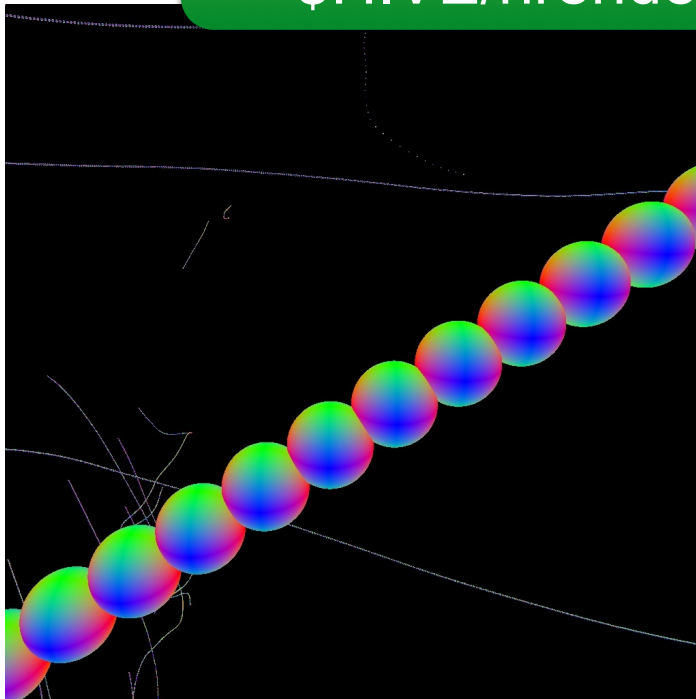
`$HIVE/hrender/test/render_pdb.scn`

粒子データ

- ・PDMlib により, 粒子データを読み込むことができます.
- ・PDMLoader. カスタムアトリビュートの読み込みにも対応しています.
- ・現在の PDMlib の制約により, **.dfi と .scn は同じディレクトリにある必要があります**
(相対パス指定で .dfi の読み込みができない)

読み込みサンプルシーン

`$HIVE/hrender/test/render_pdm.scn`



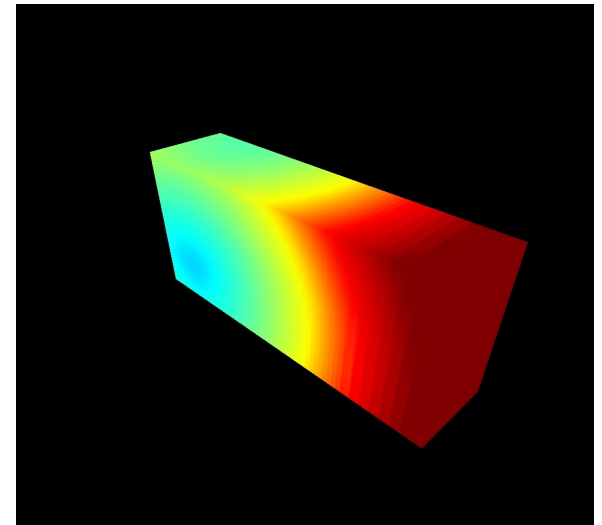
階層一様ボリューム

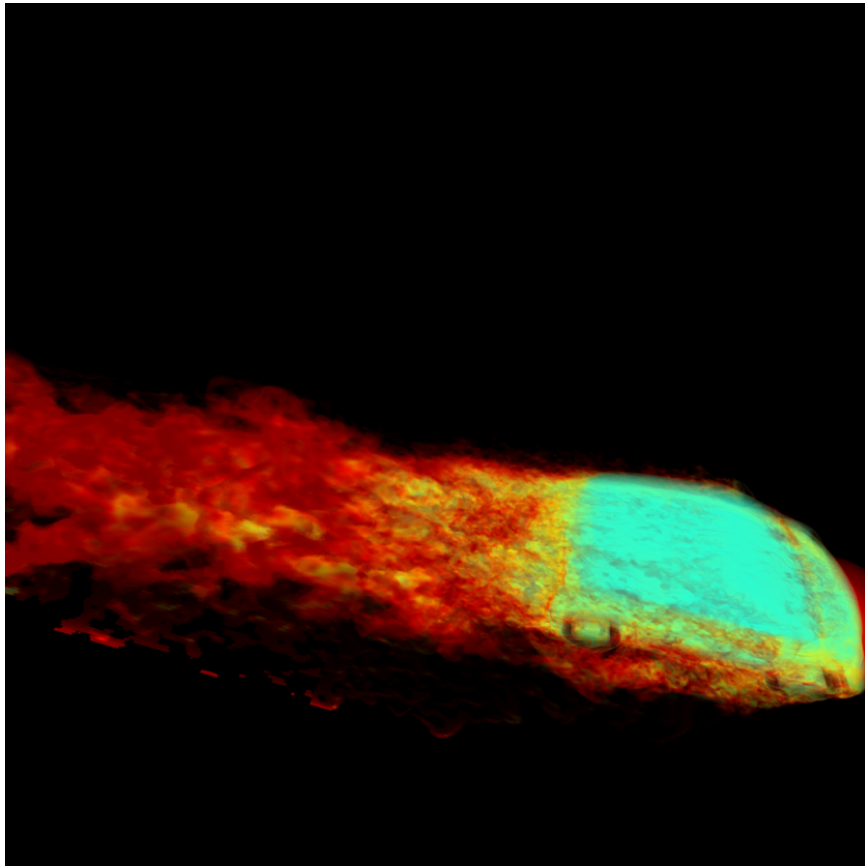
- ・ HDMLib 経由により, 階層一様ボリュームデータの読み込みを行うことができます.
- ・ double 型のボクセルデータは, ロード時に float 型へ変換されます.
- ・ HDMLoader では, 現在 **MPI 1 ノード**での読み込みのみ対応しています.
- ・ CellID フィールドデータは float 型として読み込むことができます.
- ・ LoadField("CellID", "Float32", 1, timestep)
- ・ VTK PImage 形式(.pvti)にも一部対応しています

読み込みサンプルシーン

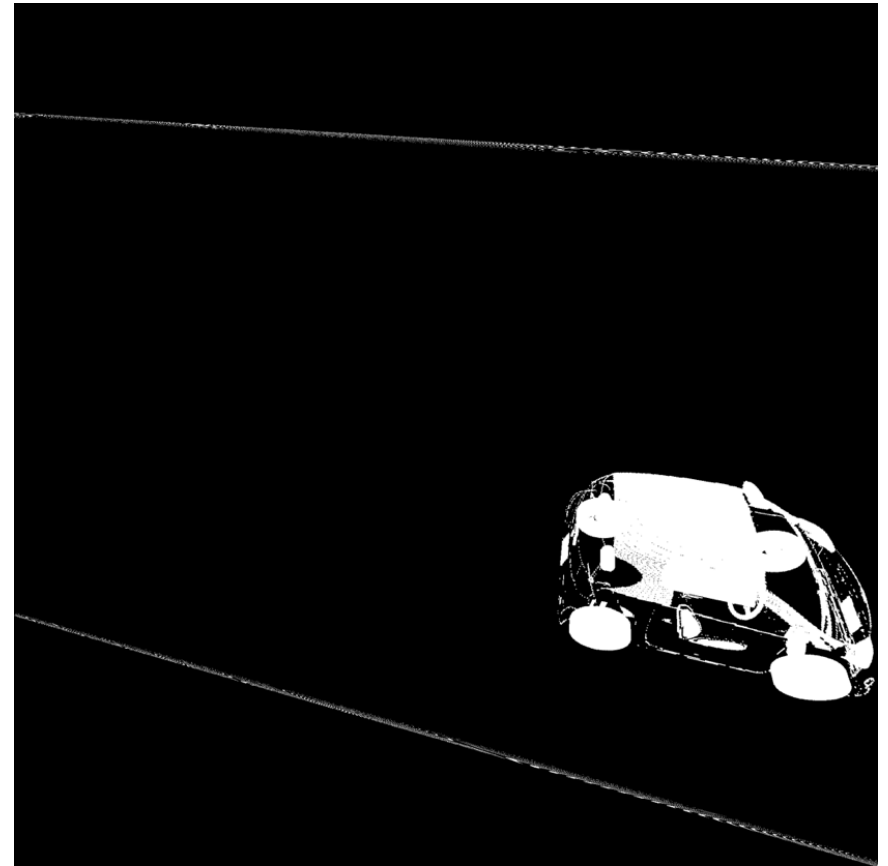
```
$HIVE/hrender/test/render_hdm.scn
```

```
$HIVE/hrender/test/render_pvti.scn
```



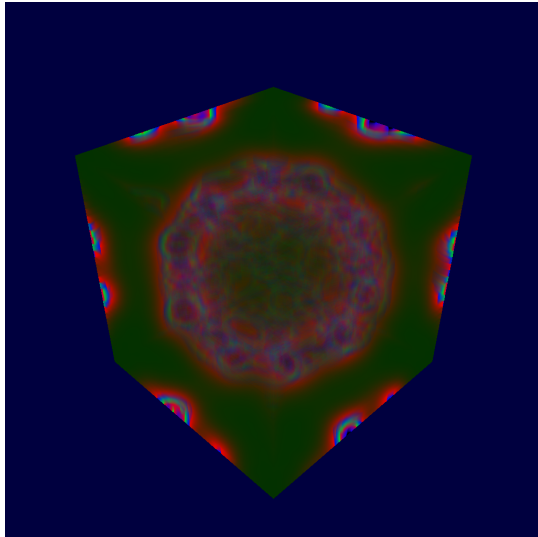


Velocity フィールド可視化

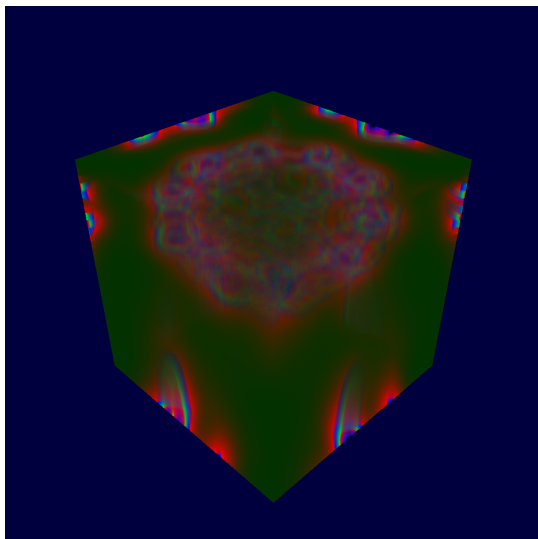


CellID フィールド可視化

非一様ボリューム



一様



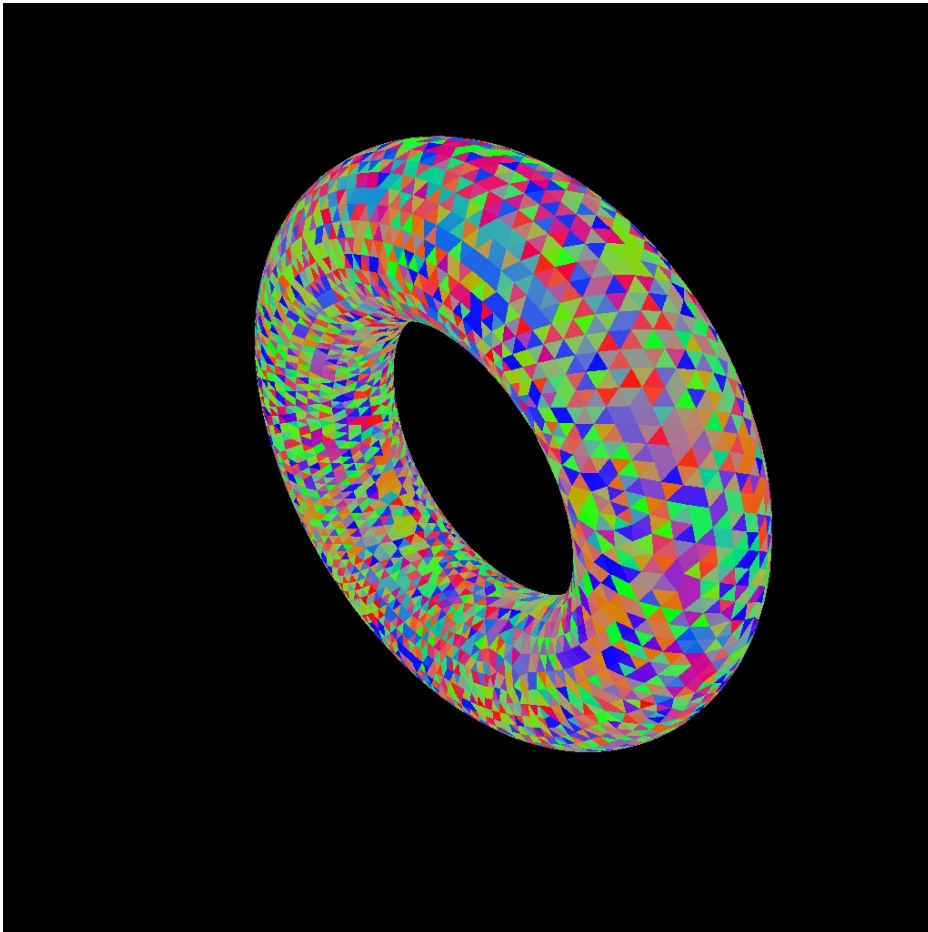
Y 方向に非一様

- ・ CDMlib 経由により, 非一様ボリュームのロードに対応しています.
- ・ CDMLoader
- ・ データが非一様で読みこまれるかどうかは, .dfi の記述に従います.

読み込みサンプルシーン

`$HIVE/hrender/test/render_cdm.scn`

非構造データ



- ・ UDMlib 経由により, 非構造データの読み込みに対応しています.
- ・ UDMLoader
- ・ 三角形, 四面体と六面体の読み込みに対応しています.

読み込みサンプルシーン

`$HIVE/hrender/test/render_udm_tetra.scn`

フィルタ機能について

フィルタプリセット

- ・ よく利用されるフィルタはプリセットとして関数化しています. 現時点では以下の 2 つに対応しています.
- ・ **Laplacian** : 7 点ラプラシアンを計算します.
- ・ **Norm** : ベクトルのノルムを計算します.

ボリュームフィルタ

- ・ C/C++ コードでボリュームデータのフィルタリングカーネルを記述できます(実験的機能).
- ・ HIVE 内でラプラシアン計算などを行うことができます.
- ・ 4 つまで入力のボリュームプリミティブを指定できます.
- ・ float 型, 同サイズのボリュームのみサポートします.
- ・ 一様ボリュームのみの対応になります.
 - ・ 非一様ボリュームは SparseVolumeToVolume ノードを使い, 一様ボリュームに変換します.

利用可能な変数

```
// numComponents 数の (w, h, d) のボリュームサイズに対して, (x, y, z) 位置  
のボクセルの  
// index 番目のボクセルのアドレスを計算するマクロ.  
// x, y, z はそれぞれ (0, w-1), (0, h-1), (0, d-1) の範囲の値にクランプさ  
れます.  
IDX(numComponents, x, y, z, index, w, h, d)  
  
// 入力ボリュームデータへのポインタ.  
float* src0, src1, src2, src3;  
  
// 出力ボリュームデータへのポインタ.  
float* dst;  
  
// ボリュームデータのサイズ. 入力と出力でサイズはすべて等しいとします.  
size_t width, height, depth;  
  
// 現在のボクセル位置  
size_t x, y, z;
```

フィルタ利用例

```
// $HIVE/hrender/test/volume_filter_expr.scn より

local filter = VolumeFilter();

-- lua のヒアドキュメント ([[, ]] で囲う)などにより, コードを記載します.
-- 単純に src0 のスカラーボリュームをコピーするフィルタコード

local filterCode = [[

// IDX(number of components, x, y, z, current component index, w, h, d)
dst[IDX(1,x,y,z,0,width,height,depth)] = src0[IDX(1,x,y,z,
0,width,height,depth)];

]]

filter:CompileOption('gcc', '-O2'); // コンパイラと, コンパイルオプションを指
定します.

filter:Expr(sph:VolumeData(), nil, nil, nil, 1, filterCode);

filter:VolumeData(); // フィルタリングされた volume プリミティブを取得
```

```
// 速度勾配テンソルの第二不変量(= laplacian(P))を計算するフィルタ例

// src0 = volume data of u(3 components)
// Wij = d u_j / d x_i

// IDX(number of components, x, y, z, current component
index, w, h, d)
float W11 = src0[IDX(3,x+1,y,z,0,width,height,depth)] -
src0[IDX(3,x-1,y,z,0,width,height,depth)];
float W12 = src0[IDX(3,x+1,y,z,1,width,height,depth)] -
src0[IDX(3,x-1,y,z,1,width,height,depth)];
float W13 = src0[IDX(3,x+1,y,z,2,width,height,depth)] -
src0[IDX(3,x-1,y,z,2,width,height,depth)];

float W21 = src0[IDX(3,x,y+1,z,0,width,height,depth)] -
src0[IDX(3,x,y-1,z,0,width,height,depth)];
float W22 = src0[IDX(3,x,y+1,z,1,width,height,depth)] -
src0[IDX(3,x,y-1,z,1,width,height,depth)];
float W23 = src0[IDX(3,x,y+1,z,2,width,height,depth)] -
src0[IDX(3,x,y-1,z,2,width,height,depth)];
```

```
float W31 = src0[IDX(3,x,y,z+1,0,width,height,depth)] -  
src0[IDX(3,x,y,z-1,0,width,height,depth)];  
float W32 = src0[IDX(3,x,y,z+1,1,width,height,depth)] -  
src0[IDX(3,x,y,z-1,1,width,height,depth)];  
float W33 = src0[IDX(3,x,y,z+1,2,width,height,depth)] -  
src0[IDX(3,x,y,z-1,2,width,height,depth)];
```

```
W11 *= 0.5;
```

```
W12 *= 0.5;
```

```
W13 *= 0.5;
```

```
W21 *= 0.5;
```

```
W22 *= 0.5;
```

```
W23 *= 0.5;
```

```
W31 *= 0.5;
```

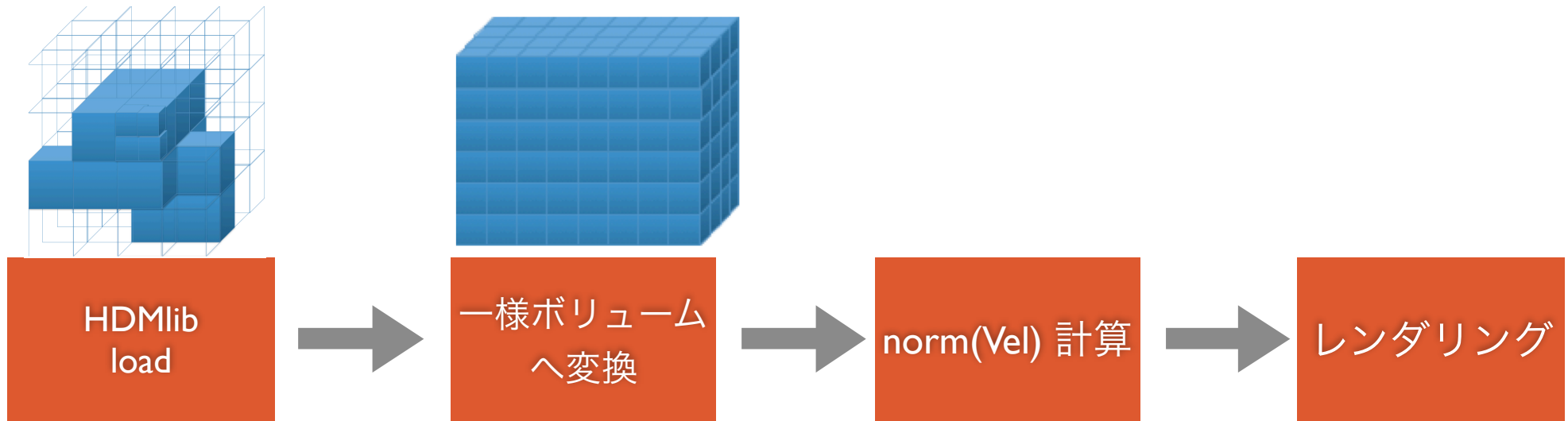
```
W32 *= 0.5;
```

```
W33 *= 0.5;
```

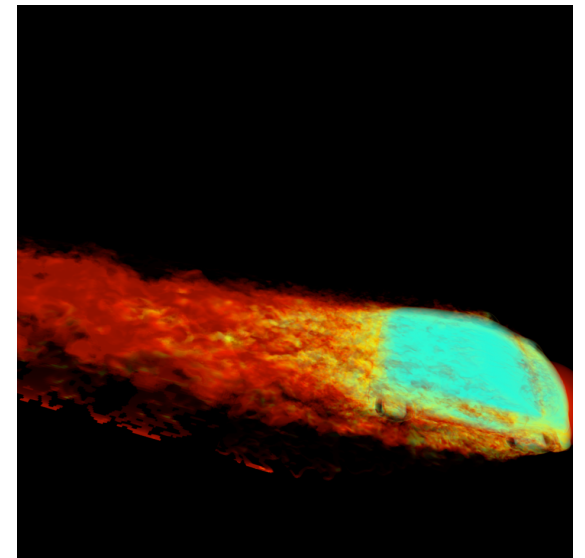
```
float I = W22 * W33 - W32 * W23 + W11 * W22 - W21 * W12 +  
W11 * W33 - W31 * W13;  
dst[IDX(1,x,y,z,0,width,height,depth)] = I;
```

可視化フロー一例

ボリリュームデータ可視化フロー例



この一連の処理を HIVE で
行う手順を参考例として示します。



シーンスクリプト記述

- ・ シーンスクリプト (lua) を記述することで可視化を行う手順を示します.
- ・ HDMLib 形式のデータには, float64 形式のベクトルボリュームデータが “Vel64” フィールド名で存在するとします.


```
-- 1) HDMlib ローダを作成し, 初期化を行います.  
-- Init はシーンスクリプト内で一回のみ呼び出すことが可能です  
local hdm = HDMLoader()  
hdm:Init('cellid.bcm', 'data.bcm')
```

```
-- 1) HDMLib ローダを作成し, 初期化を行います.  
-- Init はシーンスクリプト内で一回のみ呼び出すことが可能です  
local hdm = HDMLoader()  
hdm:Init('cellid.bcm', 'data.bcm')  
  
-- 読み込む timestep インデックスを指定します.  
local timestep = 1  
  
-- 2) フィールドを読み込みます (SparseVolume 形式)  
-- (フィールド名, フィールド型, 要素数, timestep)  
local volVel = hdm:LoadField("Vel64", "Float64", 3, timestep)  
print("Vel", volVel)
```

```
-- 1) HDMLib ロータを作成し, 初期化を行います.  
-- Init はシーンスクリプト内で一回のみ呼び出すことが可能です  
local hdm = HDMLoader()  
hdm:Init('cellid.bcm', 'data.bcm')  
  
-- 読み込む timestep インデックスを指定します.  
local timestep = 1  
  
-- 2) フィールドを読み込みます (SparseVolume 形式)  
-- (フィールド名, フィールド型, 要素数, timestep)  
local volVel = hdm:LoadField("Vel64", "Float64", 3, timestep)  
print("Vel", volVel)  
  
-- 3) フィルタ処理を行うために, SparseVolume から一様 Volume データを生成します.  
local sv2vVel = SparseVolumeToVolume()  
sv2vVel:Create(volVel, 1.0) -- 0.5 にすると半分, 0.25 では 1/4 のサイズのボリュームを生成します.  
local srcVolumeVel = sv2vVel:VolumeData()  
  
-- 各種フィルタを適用したり, HIVE UI で可視化の調整行うときはここで一旦  
-- SPH などの形式でデータを保存しておくとも便利です.  
-- local saver = SPHSaver()  
-- saver:SetVolumeData(srcVolumeVel)  
-- saver:Save(string.format('output.%06d.vel.sph', timestep))
```

```
-- 1) HDMLib ロードを作成し, 初期化を行います.  
-- Init はシーンスクリプト内で一回のみ呼び出すことが可能です  
local hdm = HDMLoader()  
hdm:Init('cellid.bcm', 'data.bcm')  
  
-- 読み込む timestep インデックスを指定します.  
local timestep = 1  
  
-- 2) フィールドを読み込みます(SparseVolume 形式)  
-- (フィールド名, フィールド型, 要素数, timestep)  
local volVel = hdm:LoadField("Vel64", "Float64", 3, timestep)  
print("Vel", volVel)  
  
-- 3) フィルタ処理を行うために, SparseVolume から一様 Volume データを生成します.  
local sv2vVel = SparseVolumeToVolume();  
sv2vVel:Create(volVel, 1.0); -- 0.5 にすると半分, 0.25 では 1/4 のサイズのボリュームを生成します.  
local srcVolumeVel = sv2vVel:VolumeData()  
  
-- 各種フィルタを適用したり, パラメータを変えての可視化を繰り返すときはここで一旦  
-- SPH などの形式でデータを保存しておく便利です.  
-- local saver = SPHSaver()  
-- saver:SetVolumeData(srcVolumeVel)  
-- saver:Save(string.format('output.normVel.%06d.sph', timestep))  
  
-- 4) 一様ボリュームデータに norm を計算するフィルタを適用します.  
local filter = VolumeFilter();  
filter:Norm(srcVolumeVel);  
local filteredVol = filter:VolumeData()
```

-- 5) ボリュームアナライザを適用し, 適切にフィルタ処理が行われたか確認します.

```
local analy = VolumeAnalyzer()
analy:Execute(filteredVol)
print('volume min/max X:', analy:MinX(), analy:MaxX())
print('volume min/max Y:', analy:MinY(), analy:MaxY())
print('volume min/max Z:', analy:MinZ(), analy:MaxZ())
```

-- 5) ボリュームアナライザを適用し、適切にフィルタ処理が行われたか確認します。

```
local analy = VolumeAnalyzer()
analy:Execute(filteredVol)
print('volume min/max X:', analy:MinX(), analy:MaxX())
print('volume min/max Y:', analy:MinY(), analy:MaxY())
print('volume min/max Z:', analy:MinZ(), analy:MaxZ())
```

-- 6) プリセットのレイマーチングシェーダを使い、レンダリング処理を行います。

local tftex = JetTransferFunctionTexture() -- 詳細は \$HIVE/hrender/render_rawvolume.scn 参照

```
local volume = VolumeModel()
volume:Create(filteredVol)
volume:SetShader('def_volume_raymarch_tf.frag') -- ファイルは $HIVE/hrender/test にあります。
volume:SetTexture('tf_tex', tftex:ImageData())
-- CLAMP_TO_EDGE wrap mode for transfer function texture.
volume:SetTextureWrapping('tf_tex', true, true, true)
-- CLAMP_TO_EDGE wrap mode for voxel data.
volume:SetClampToEdge(true, true, true)
```

-- ここで、tf_min, tf_max, tf_opacity を調整することで色合いが変わります。

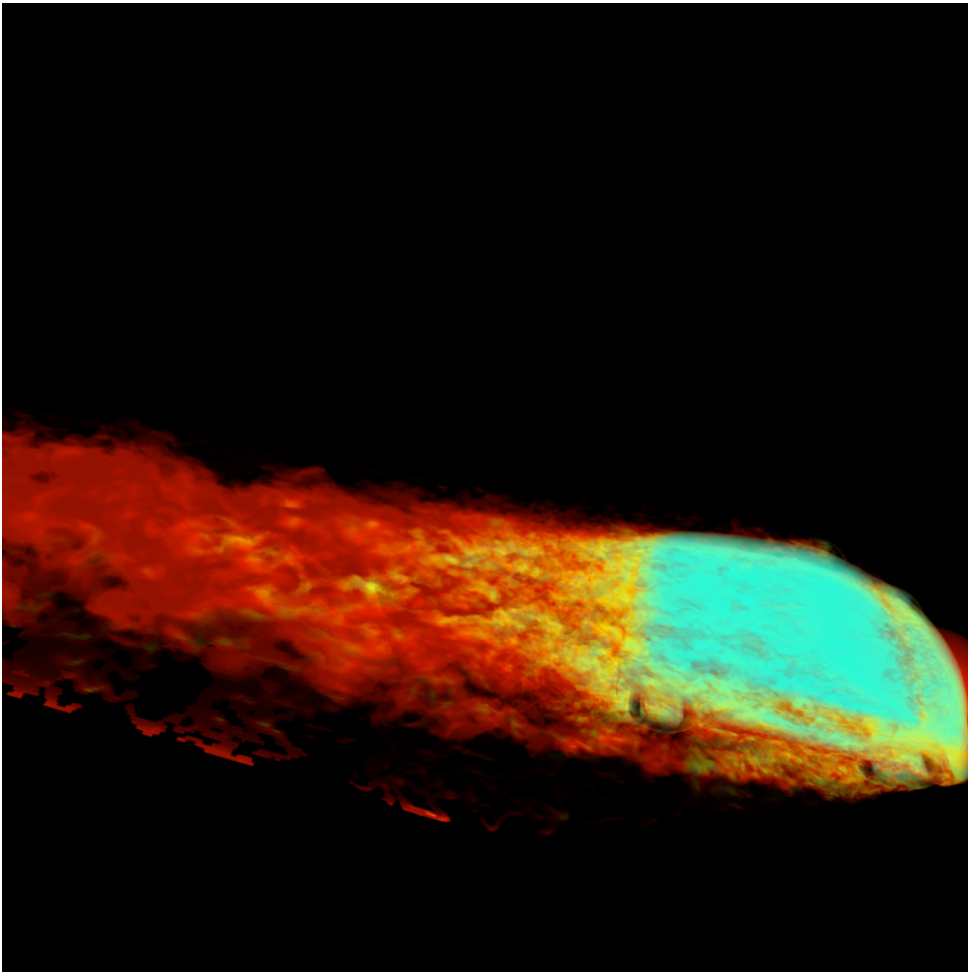
-- VolumeAnalyzer の結果の値を参考にするなどして、値を決めていきます。

```
volume:SetFloat('tf_min', -10.0)
volume:SetFloat('tf_max', 20.0)
volume:SetFloat('tf_opacity', 0.002)
```

```
local camera = Camera()
camera:SetScreenSize(1024, 1024)
camera:LookAt(
    -200, 400, 400,
    -100, 0, 0,
    0, 0, 1, 85 -- z-up
)
camera:SetFilename(string.format('render.%06d.vel.png', timestep))
```

render {camera, volume} -- 実際にレンダリングを実行します。

```
# $HIVE/hrender/test で作業します.  
  
$ cd $HIVE/hrender/test  
$ vi render_vel.scn  
  
$ mpirun -np 1 ../../build/bin/hrender render_vel.scn  
  
# HDD から volume データを読み込むため、最初のロードには数分ほどかかります。  
# render.XXXXXX.vel.png が生成され、期待する画像がレンダリングされていれば成功です。
```



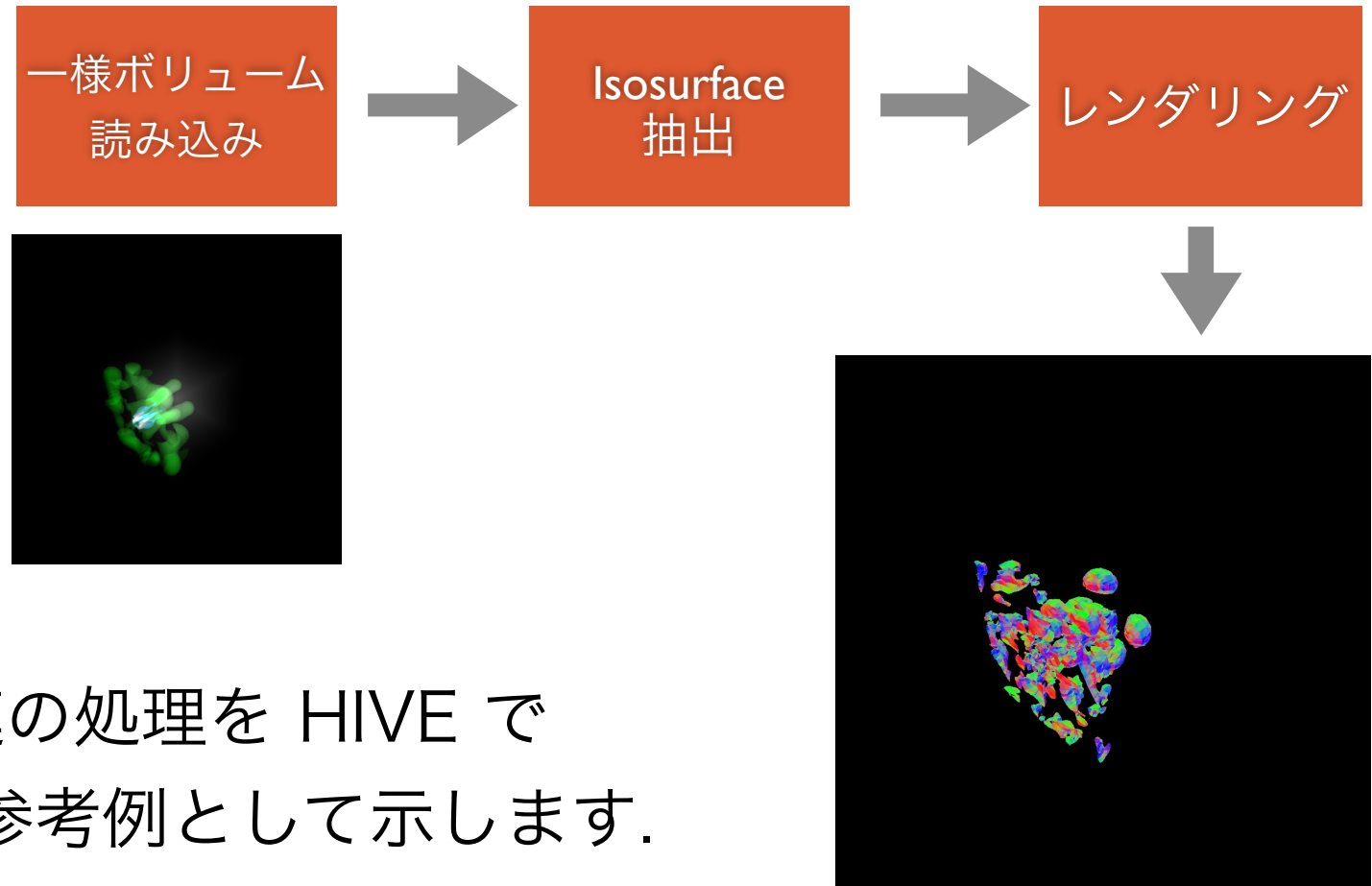
HDD から Volume データを読み込むため、最初のロードには数分ほどかかります。

render.XXXXXX.vel.png が生成され、期待する画像がレンダリングされていれば成功です。

Isosurface

- ・ 一様ボリュームに対して Isosurface メッシュを生成することができます.
- ・ 階層ボリュームなどの形式の場合は、一旦一様ボリュームに変換しておきます.
- ・ 単純なマーチンキューブ法を利用しているため、生成されるメッシュの品質はあまり高くありません

Isosurface



`$HIVE/hrender/test/volume_to_mesh.scn`

-- 1) カメラの情報を設定します。

```
camera:SetScreenSize(1024, 1024)
camera:SetFilename('image_sph_isosurf.jpg')
```

```
camera:LookAt(
    -50,80,80,
    40,0,0,
    0,1,0,
    60
)
```

-- 2) ボリュームデータ (SPH) を読み込みます。

```
local sph = SPHLoader()
sph:Load('prs_0000002000.sph')
local volumedata = sph:VolumeData()
```

-- 3) Isosurface を抽出します。

```
local surfacer = VolumeToMeshData()

local isovalue = 0.0005 -- iso 値
surfacer:Create(volumedata) -- 対象のボリュームデータをセット
surfacer:SetIsoValue(isovalue) -- Iso 値をセット
surfacer:IsoSurface() -- 実際に Isosurface 処理を行う。
```

-- 4) 生成された Isoruface メッシュをレンダリングします。

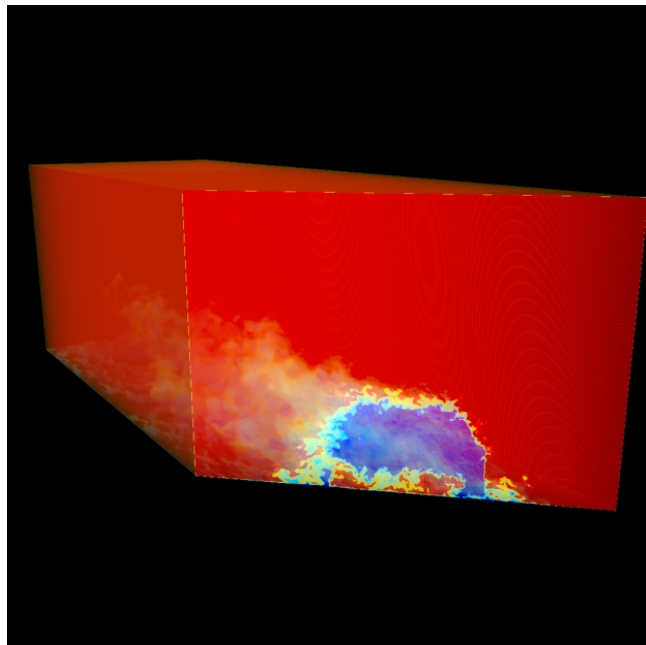
```
local model = PolygonModel()
local isosurface = surfacer:MeshData() -- ポリゴンによる isosurface メッシュを取得
```

```
model:Create(isosurface)
model:SetShader('normal.frag')
```

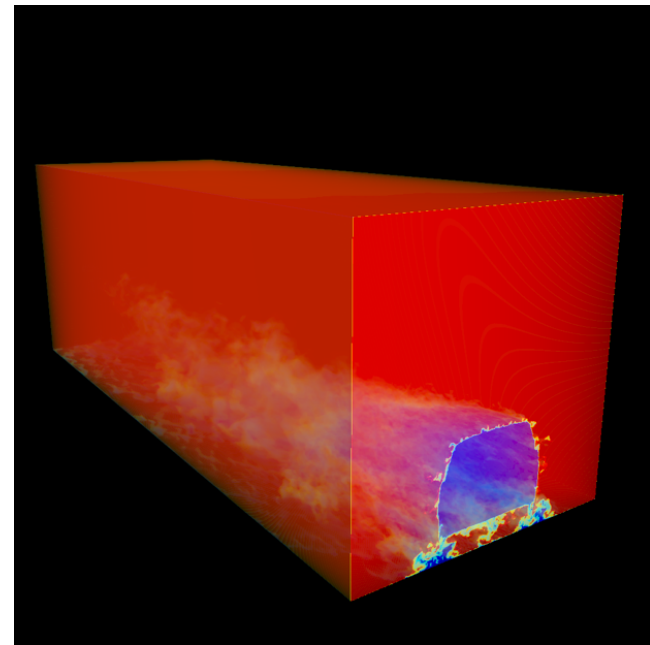
```
render {camera, model}
```

任意断面

- ・ シェーダで平面との交差判定を行うことで, 任意断面を表現します.



斜め切断面



垂直切断面

```
-- $HIVE/hrender/test/render_volume_cut_plane.scn
```

```
...
```

```
-- レイマーチングに加えて断面の処理も行うシェーダ
```

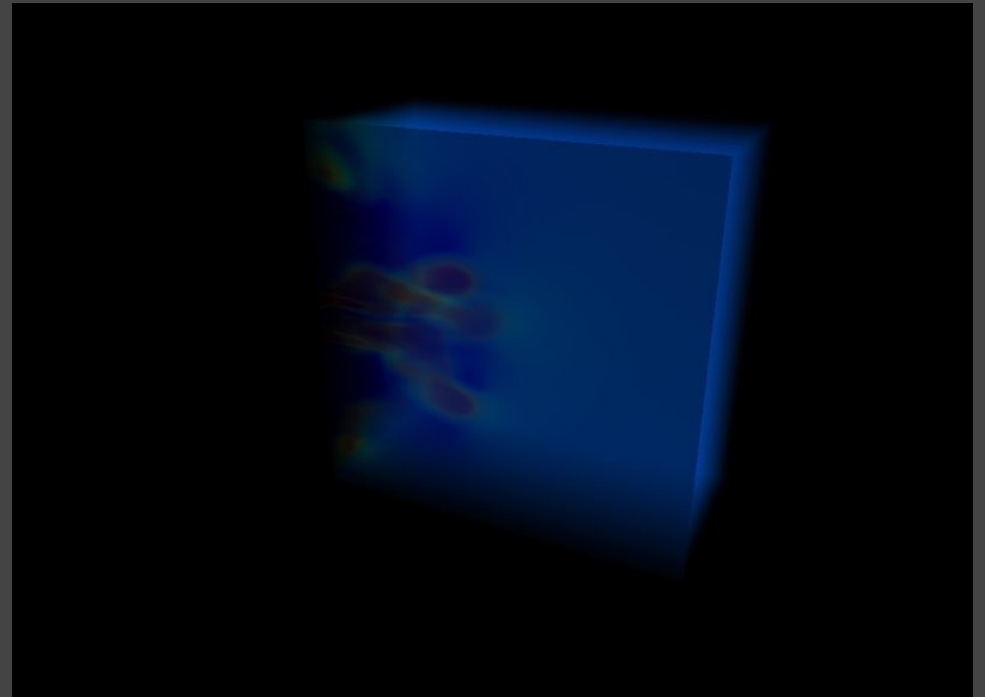
```
volume:SetShader('def_volume_cut_plane.frag')
```

```
-- 断面のパラメータ(平面方程式)を指定します.
```

```
volume:SetVec4('cut_plane', 0.0, 0.0, 1.0, 0.0);
```

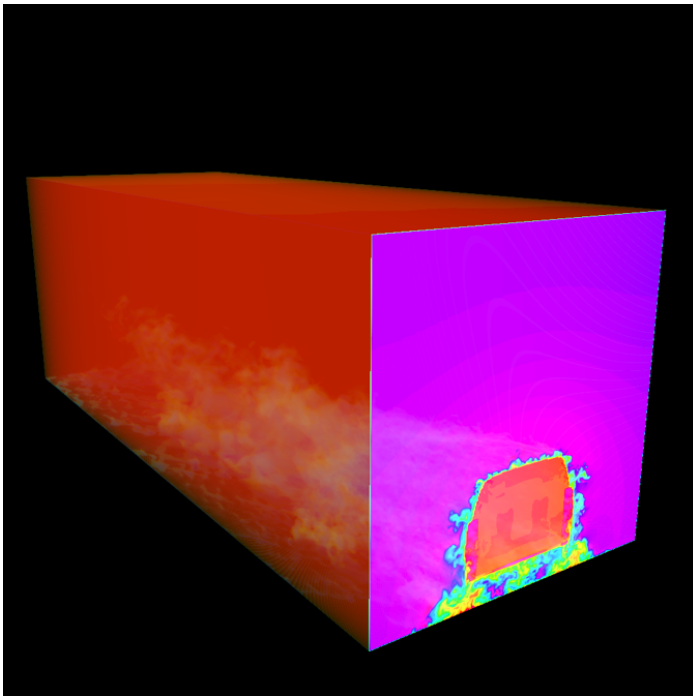
```
volume:SetFloat('enable_cut_plane', 1.0); -- 断面を有効にする.
```

```
render {camera, volume}
```

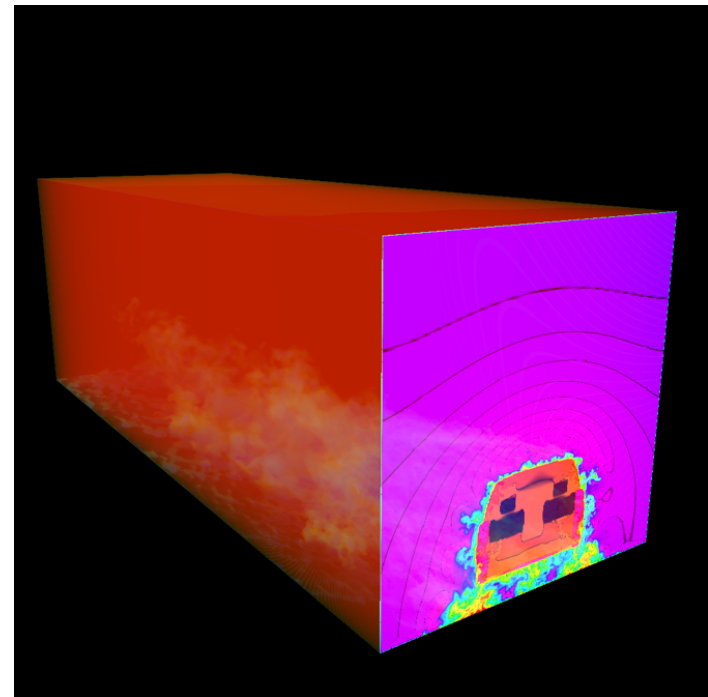


コンター

- ・ シェーダで階調を変えて色つけすることで、コンターを表現します。
- ・ コンターラインは等幅で表現はできないため、コンターパラメータや視点によってはうまくコンターラインを描画できないケースがあります。



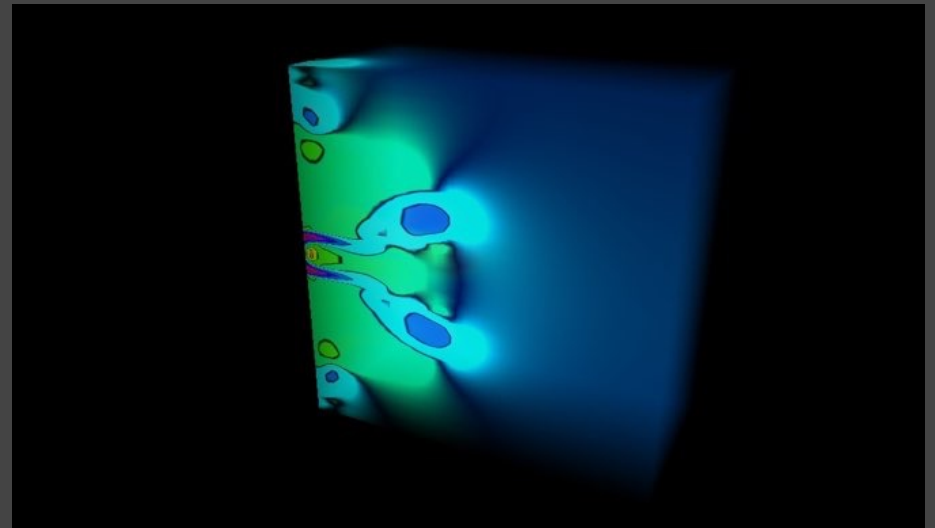
塗りつぶしのみ



コンターラインあり

```
-- $HIVE/hrender/test/render_volume_cut_plane_contour.scn
...
-- レイマーチングに加えて断面のコンター処理も行うシェーダ
volume:SetShader('def_volume_cut_plane_contour.frag')
-- 断面のパラメータ(平面方程式)を指定します。
volume:SetVec4('cut_plane', 0.0, 0.0, 1.0, 0.0)
volume:SetFloat('enable_cut_plane', 1.0); -- 断面を有効にする。
-- コンターのパラメータを指定します。
-- (最小値、 最大値、 ステップ幅、 コンターライン幅)
volume:SetVec4('contour_param', -0.5, 0.5, 0.1, 0.1)

render {camera, volume}
```



MPI並列レンダリング について

MPI 並列レンダリング

- ・ MPI Compositor を利用(HIVE_BUILD_WITH_COMPOSITOR オプションを指定して hrender をビルド)することにより, 画面分割およびデータ並列での並列レンダリングを行うことができます.
 - ・ 画面分割 : シーンスクリプトで設定を行うことで, HIVE(SURFACE)側が自動で画面を分割します.
 - ・ 全データがノードに収まり, レンダリングに時間がかかるときに有効です(8K, 16K 解像度で可視化など).
 - ・ データ並列 : ユーザが明示的にどのノードにデータを配置するかを SceneScript に記述する必要があります.
 - ・ データが大きくて各ノードに全部のデータが格納できないときに有効です.

SceneScript で利用可能な MPI 関連関数

- `bool mpiMode()`

- `hrender` が MPI ビルドされているかの情報を取得します

- `int mpiRank()`

- MPI のランク番号を返します.

- `int mpiSize()`

- MPI の総ランク数を返します.

- `screenParallelRendering(bool)`

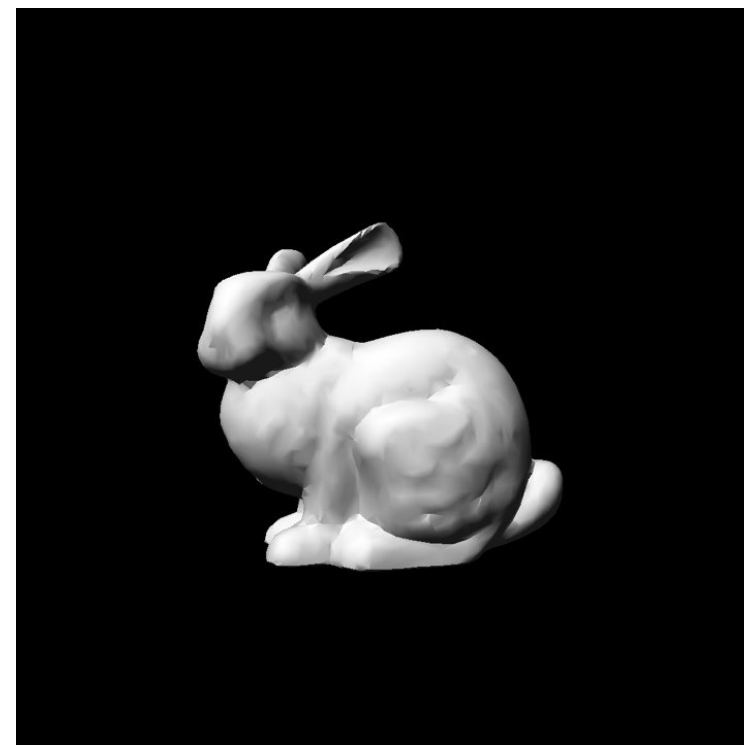
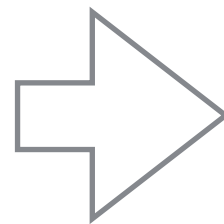
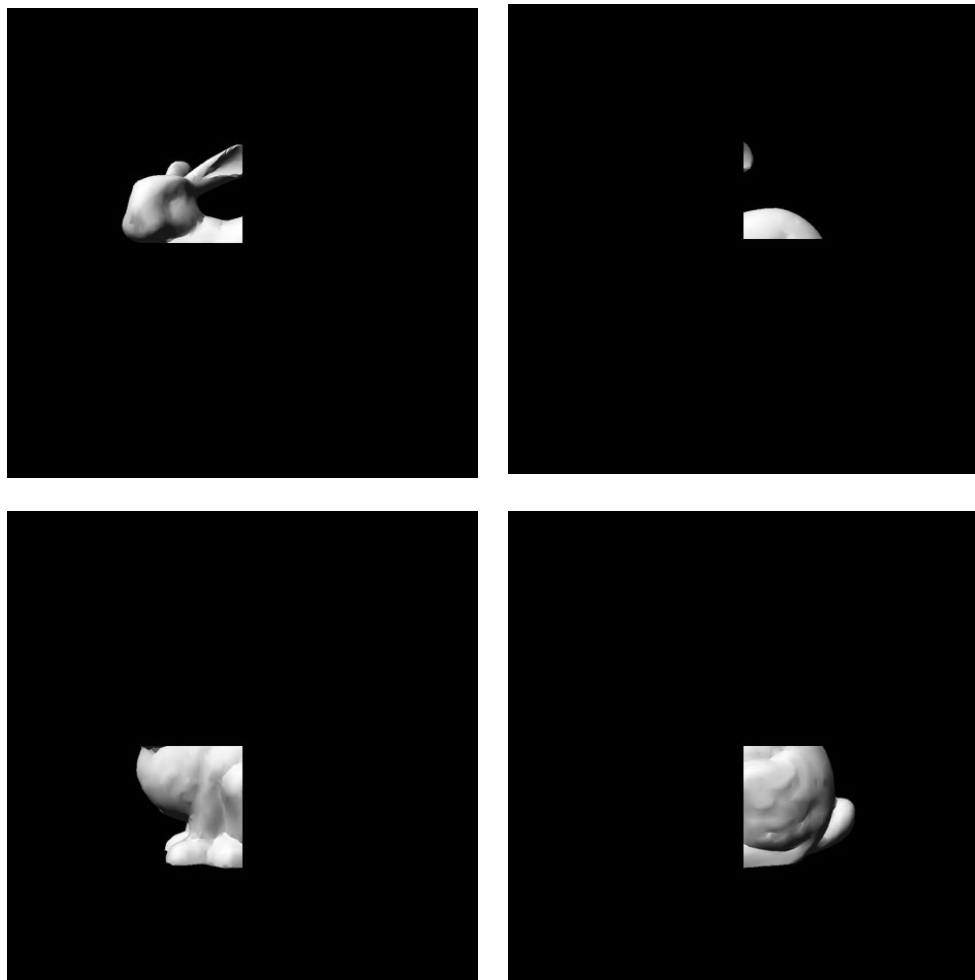
- 画面分割でレンダリングするかを指定します (default = false)

- N ランク時に画面を自動で N 等分します.

- 条件によっては分割できないケースがあります.

- ランク数や画面解像度には 2 の倍数など割り切りがしやすい値を設定してください.

画面分割レンダリング例



画面分割シーンスクリプト例

```
-- $HIVE/hrender/test/render_obj_screenparallel.scn

...

-- MPI 有効でビルドされていれば, screen parallel レンダリングを有効にする.
if mpiMode() == true then
  screenParallelRendering(true)
end

render { camera, model }

-- 画像は rank = 0 のみ出力されます.
-- 各ランクの画像を保存したいときは camera:GetImageBuffer() と ImageSaver を
  利用することができます
```

データ並列シーンスクリプト例

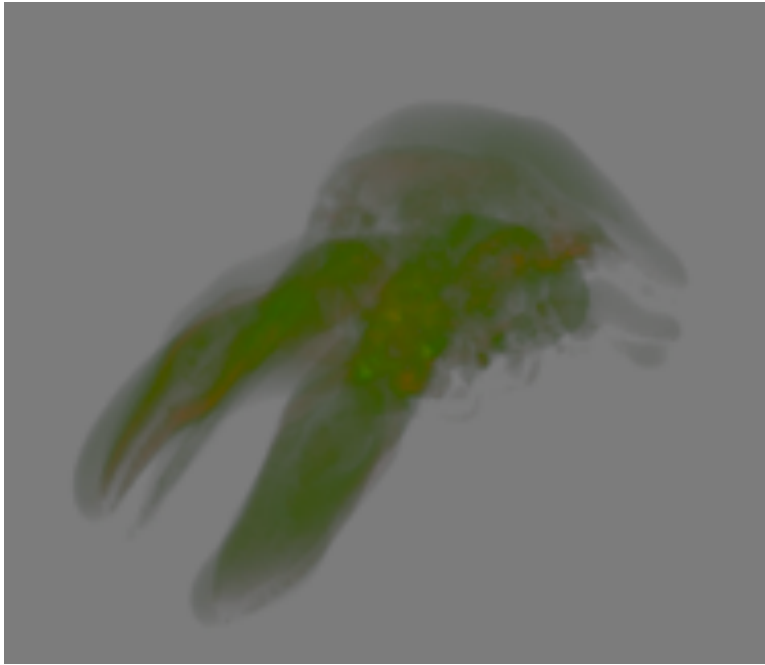
```
...  
  
-- ランク数に応じて、事前に分割済みのファイルをロードする。  
local obj = OBJLoader()  
obj:Load(string.format('input.%06d.obj', mpiRank()))  
  
-- optional: ランクに応じて位置のオフセットを与える。  
-- obj:SetTranslate(mpiRank() * 100, 0, 0)  
  
...  
  
render { camera, model }  
  
-- 画像は rank = 0 のみ出力されます。  
-- Compositor により各ランクの画像は上書きされるため、各ランクの画像を取得しても期待  
する画像にはなっていません。
```

シェーダについて

シェーディング詳細

- ・ HIVE では, GLSL によりシェーディング(色付け)をカスタマイズし, ユーザにより多彩な表現を行うことができます.

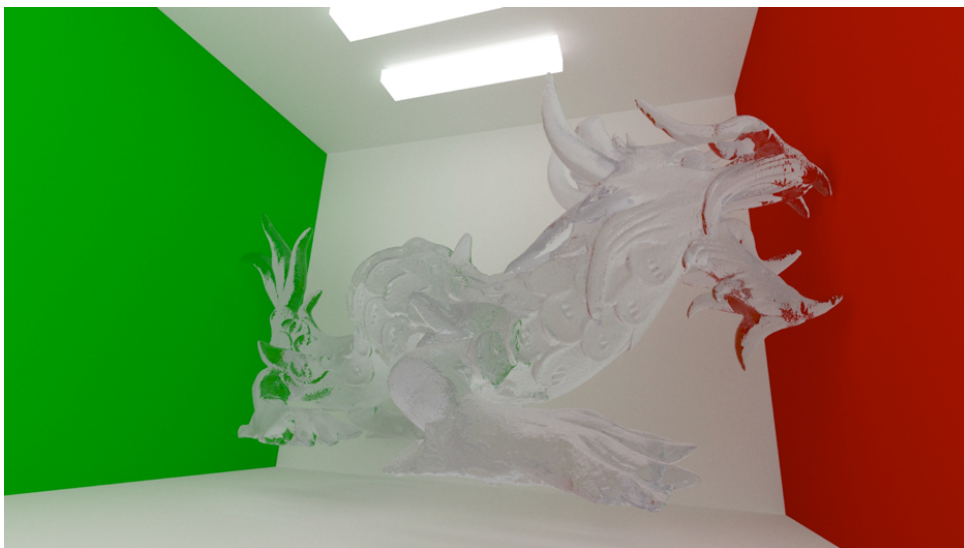
HIVE シェーダ例



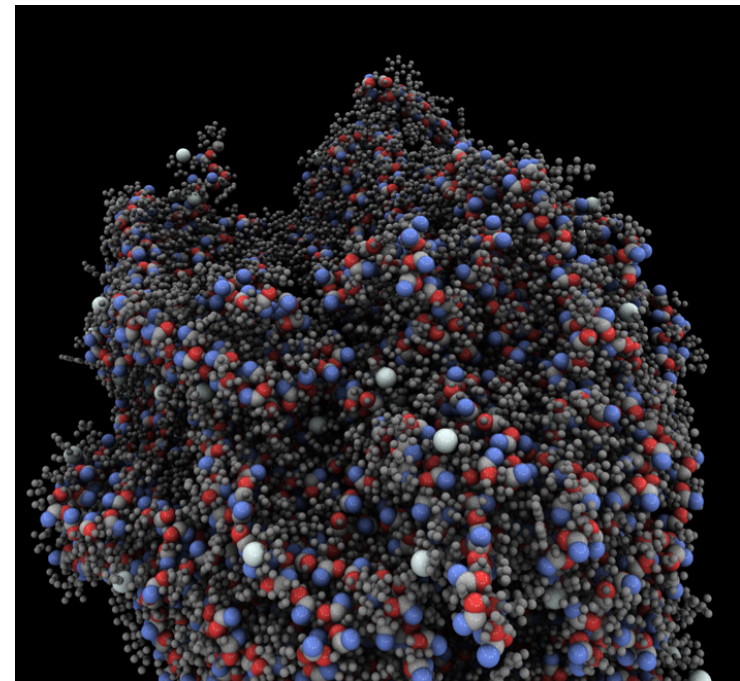
レイマーチング



レイマーチングベース Isosurface

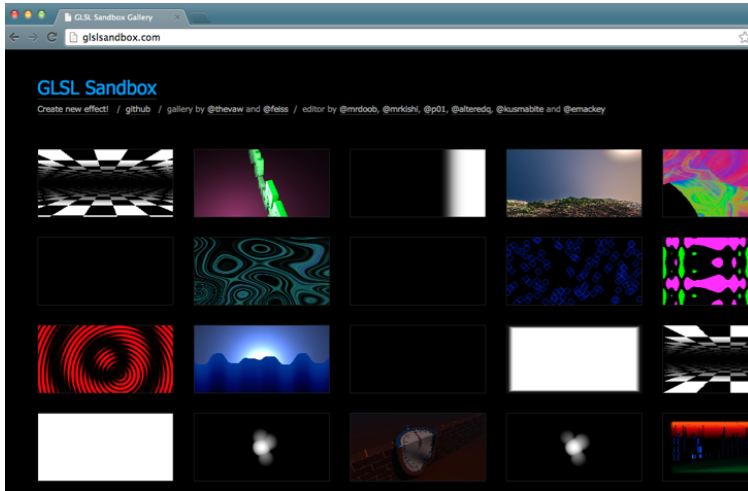


パストレーシング

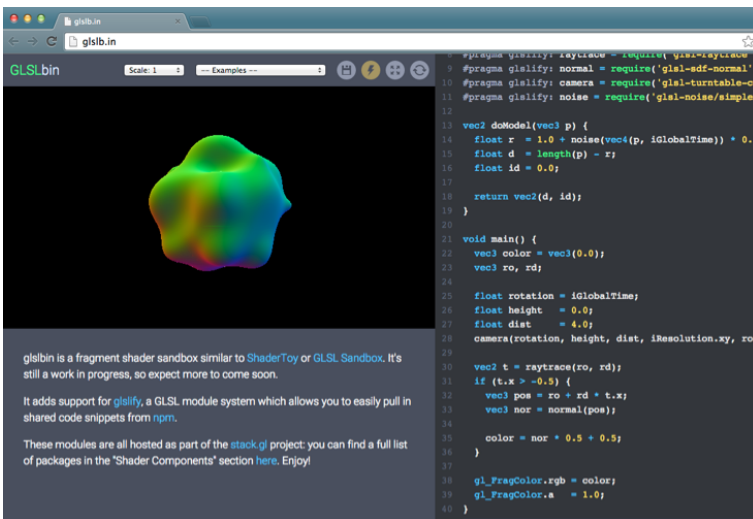


アンビエントオクルージョン

GLSL



GLSL sandbox



glslbin

- ・ シェーダは GLSL(OpenGL Shading Language)で記載します。
- ・ hrender(SURFACE) 利用時は C/C++ に変換されて実行されるため, CPU で動作します(x86, SPARC 対応)。
- ・ GLES 2.0(WebGL) のフラグメントシェーダに概ね対応しています。
- ・ 頂点シェーダには対応していません。
- ・ GLSL を試したり, GLSL の記述でどのようなシェーディングができるのかは, glslsandbox.com や glslbin を参考ください。

シェーダの記述

```
// tutorial.frag
// 最小の GLSL シェーダサンプル。
// 単純に物体を白色で塗り潰します。

#extension GL_LSGL_trace : enable
#extension GL_OES_texture_3D : enable

#ifdef GL_ES
precision mediump float;
#endif

void main(void)
{
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    return;
}
```

- main() 関数がエントリポイントになります。
- gl_FragColor に出力の色を書き込みます (RGBA)
- alpha = 0.0 で完全透明,
alpha = 1.0 で完全不透明になります

拡張について

- ・ 以下の拡張機能をサポートしています。

```
// trace(), isectinfo() などのレイトレース用関数を使えるようにします
#extension GL_LSGL_trace : enable

// random() 関数を使えるようにします。
#extension GL_LSGL_random : enable

// 3D texture を有効にします。
#extension GL_OES_texture_3D : enable
```

```
// モデル(ポリゴン, ボリューム, 粒子, ライン)共通の
// シェーダ設定項目

// モデル変換を設定
model:SetTranslate(tx, ty, tz)
model:SetScale(sx, sy, sz)
model:SetRotate(rx, ry, rz)
model:SetTransformMatrix(tbl)

// シェーダを割り当て
model:SetShader('shaderfile.frag')

// uniform float 変数を設定
model:SetFloat('name', value)

// uniform vec2, vec3, vec4 変数を設定
model:SetVec2('name', value0, value1)
model:SetVec3('name', value0, value1, value2)
model:SetVec4('name', value0, value1, value2, value3)

// テクスチャ画像を設定
// uniform sampler2D としてシェーダ内でアクセス可能.
model:SetTexture('name', image)

// テクスチャのフィルタリングを設定
model:SetTextureFiltering('name', trueOrFalse)

// テクスチャのラップモードを設定する.
// ボリュームデータや 1D texture の場合に, true を設定することで wrapping しないようにできる
model:SetTextureWrapping('name', clampToEdgeS, clampToEdgeT, clampToEdgeR)
```

シェーディング空間

- ・ シェーディング空間は World 空間を基準とします.
- ・ `isectinfo()` で取得される位置ベクトルは World 空間, 法線ベクトルは `inverse transpose` 変換により World 空間へと変換されています
- ・ その他ユーザ定義の `varying` 変数は, 必要であれば適宜 world 空間に変換されるように手動で処理する必要があります.
- ・ `lsgl_World`
- ・ `lsgl_WorldInverse`
- ・ `lsgl_WorldInverseTranspose`
- ・ が predefined された `mat4 uniform` 変数として利用可能です.

```
// ボリュームプリミティブの移動・回転・スケールを考慮してシェーディングを行うサンプル。
// ボリュームプリミティブの表面の色を返す。

#extension GL_LSGl_trace : enable
#extension GL_OES_texture_3D : enable

#ifdef GL_ES
precision mediump float;
#endif

// == -> HIVE(hrender)が自動で定義する uniform 値
uniform sampler3D tex0; // ボリュームテクスチャ
uniform float width, height, depth; // volume dim
uniform mat4 lsgl_WorldInverse; // world -> local 変換行列
// == <-

void main(void)
{
    vec3 p, n, dir;
    isectinfo(p, n, dir); // p は world 空間
    // lsgl_WorldInverse をかけることにより, local 空間へと変換
    vec3 localP = (lsgl_WorldInverse * vec4(p, 1.0)).xyz;
    vec3 localTexCoord = localP / vec3(width, height, depth); // -> [-1, 1]^3
    localTexCoord = 0.5 * localTexCoord + 0.5; // -> [0, 1]^3

    vec4 dens = texture3D(tex0, localTexCoord);
    gl_FragColor = vec4(normalize(dens.xyz), 1.0);
    return;
}
```

HIVE(SURFACE) 拡張

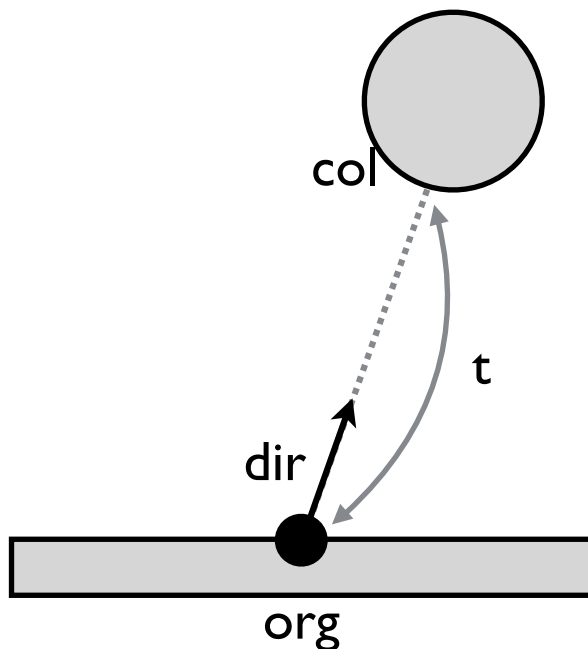
- ・ HIVE(SURFACE) でのみ利用可能な GLSL 拡張について説明します.
- ・ GL_LSGL_random
 - ・ random()
- ・ GL_LSGL_trace
 - ・ trace()
 - ・ raydepth()
 - ・ isectinfo()
 - ・ rayattrib()

```
float random(out float rnd)
```

- ・ 一様疑似乱数を取得します.
- ・ $[0.0, 1.0)$ の値を返します.
- ・ マルチスレッドセーフの実装となっています.

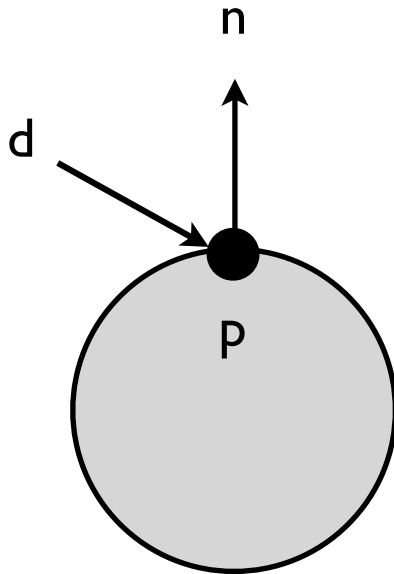
```
float trace(vec3 org,  vec3 dir);  
float trace(vec3 org,  vec3 dir,  out vec4 col);  
float trace(vec3 org,  vec3 dir,  out vec4 col,  float attrib);
```

- org から dir 方向へレイをトレースし, 交点を見つけます.
- 返り値に交点までの距離 t を返します.
- 交点が見つからない場合は, 負の最大値が帰ります.
- col(第三引数)が指定された場合, 交点が見つかったときにその場所でシェーディングを行い, 色を返します.
- attrib(第四引数)でレイにカスタム値を持たせることができます(後続のシェーダ内で rayattrib() で値を取得可能)




```
float isectinfo(out vec3 p, out vec3 n, out vec3 d);
```

- ・ 交点情報を取得します.



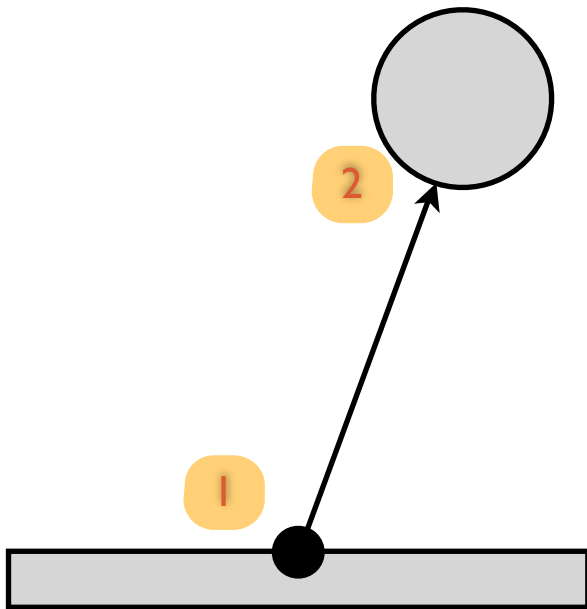
- ・ 返り値：視点(一つ前のシェーディング点)からの距離
- ・ p : 交点位置(world 空間)
- ・ n : 幾何法線ベクトル(world 空間)
- ・ d : 入射ベクトル(world 空間)

```
int raydepth(out int depth)
```

- ・ 現在のレイの反射回数を返り値, および引数 depth に返します.
- ・ trace() 関数によりシェーダが再帰的に呼ばれるため, ユーザは raydepth で反射回数を見て明示的に打ち切りをシェーダ内で行う必要があります.
- ・ 60 回以上反射する場合は, 無限ループを避ける為に強制的にプログラムが終了します.

```
float rayattrib(out float attrib);
```

```
// 2nd shader call  
float attr;  
rayattrib(attr); // => 3.14
```



```
// 1st shader call  
trace(org, dir, col, 3.14);
```

- ・ 一つ前のシェーダで trace 関数の第四引数にセットされたアトリビュート値を取得します.
- ・ レイに情報を持たせて何かカスタムの処理をするときに利用します.

```
// シンプルなパストレーシングシェーダ 1/3
// trace と random を利用してモンテカルロパストレーシングを行います。
#extension GL_LSGL_trace : enable
#extension GL_LSGL_random : enable

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.141592

// 正規直交基底ベクトルを生成。ランダムなレイの方向を生成するのに利用
void orthoBasis(out vec3 basis0,out vec3 basis1,out vec3 basis2, vec3 n)
{
    basis2 = vec3(n.x, n.y, n.z);
    basis1 = vec3(0.0, 0.0, 0.0);

    if ((n.x < 0.6) && (n.x > -0.6))
        basis1.x = 1.0;
    else if ((n.y < 0.6) && (n.y > -0.6))
        basis1.y = 1.0;
    else if ((n.z < 0.6) && (n.z > -0.6))
        basis1.z = 1.0;
    else
        basis1.x = 1.0;

    basis0 = cross(basis1, basis2);
    basis0 = normalize(basis0);

    basis1 = cross(basis2, basis0);
    basis1 = normalize(basis1);
}
```

```
// シンプルなパストレーシングシェーダ 2/3
```

```
vec3 randRay(vec3 nn)
{
    float r0, r1;
    random(r0);
    random(r1);
    float theta = sqrt(r0);
    float phi   = 2.0 * PI * r1;
    vec3 ref;

    ref.x = cos(phi) * theta;
    ref.y = sin(phi) * theta;
    ref.z = sqrt(1.0 - theta * theta);
    vec3 basis0 = vec3(1,0,0);
    vec3 basis1 = vec3(0,1,0);
    vec3 basis2 = vec3(0,0,1);
    orthoBasis(basis0,basis1,basis2, nn);
    vec3 rray;
    rray.x = ref.x * basis0.x + ref.y * basis1.x + ref.z * basis2.x;
    rray.y = ref.x * basis0.y + ref.y * basis1.y + ref.z * basis2.y;
    rray.z = ref.x * basis0.z + ref.y * basis1.z + ref.z * basis2.z;
    return rray;
}

void main(void)
{
    int depth; raydepth(depth);
    if (depth > 10) { // レイの反射回数が多い場合は打ち切り
        gl_FragColor = vec4(0.01, 0.01, 0.01, 1.0);
        return;
    }
    vec3 p, n, dir;
    isectinfo(p, n, dir); // 交点の情報を取得
```

```
// シンプルなパストレーシングシェーダ 3/3
```

```
vec3 Lo = vec3(0.0);

int numSamples = 1;

if (depth == 0) numSamples = 128;

int s;
for (s = 0; s < numSamples; s++) {
    float rnd; random(rnd);
    if (rnd < 0.5) { // russian roulette
        vec3 V = normalize(-dir);
        if (dot(n, V) < 0.0) { // face forward
            n = -n;
        }

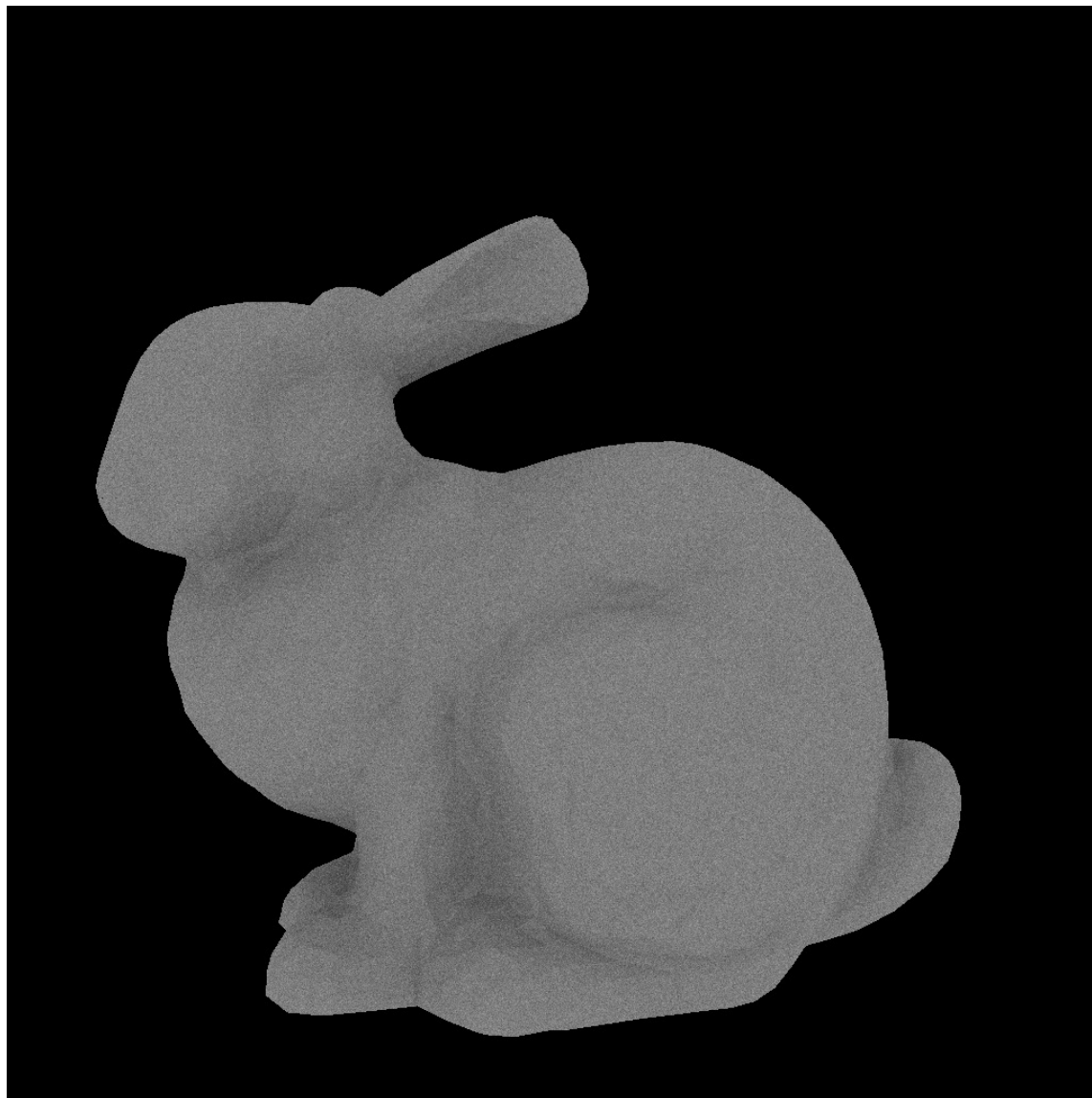
        vec3 wi = randRay(n);

        vec4 traceCol = vec4(0.0);
        float t = trace(p + 0.001 * n, wi, traceCol, 0.0);
        if (t < -1.0) {
            Lo += vec3(1.0);
        } else {
            Lo += traceCol.rgb;
        }
    }
}

Lo /= float(numSamples);

gl_FragColor = vec4(Lo, 1.0);
}
```

パストレーシングシェーダ適用結果



シェーダファイル例

- ・ `$HIVE/hrender/test/***.frag`
- ・ にいくつかシェーダのプリセットがあります.
- ・ SURFACE のexamples にも参考となるシェーダがあります
- ・ <https://github.com/avr-aics-riken/SURFACE/tree/master/examples>