Space Programming with CityCompiler

CityCompilerで空間をプログラミングしよう!



建物へのプロジェクションマッピングやたくさんのモニタを使用するような大きな規模のインスタレーションを作りたいと思った時、環境やハードウェアがある程度めぐまれていないとあれこれ試行錯誤するのは難しいです。そんな時に便利なのがCityCompilerです。CityCompilerを使えば、バーチャルな世界にカメラやディスプレイ、プロジェクタなどをあれこれ置いてシミュレーションを行うことができます。CityComplierを使って空間のプログラミングに挑戦しましょう!

CityCompilerとは What is CityCompiler?

CityCompilerは空間を使ったインタラクティブなインスタレーションを作るためのプロトタイピング環境です。Google SketchUpで作られた現実世界の3Dモデルと Processingのソースコードを組み合わせて、バーチャルな3D空間上でインタラクティブシステムのシミュレーションを行うことができます。

CityCompilerは慶應義塾大学 中西泰人研究室（http://unitedfield.net/about/）において開発されています。CityCompilerそのものはJavaのクラスライブラリです。Javaのソースコードで提供され、Eclipseなどの統合開発環境（IDE）で利用することができます。

セットアップと動作確認 Setup

No. 1

プログラミングする人もしない人も、とにかくインストールして動かしてみましょう！

■ インストールするもの

CityCompilerを使うには以下のものが必要です。

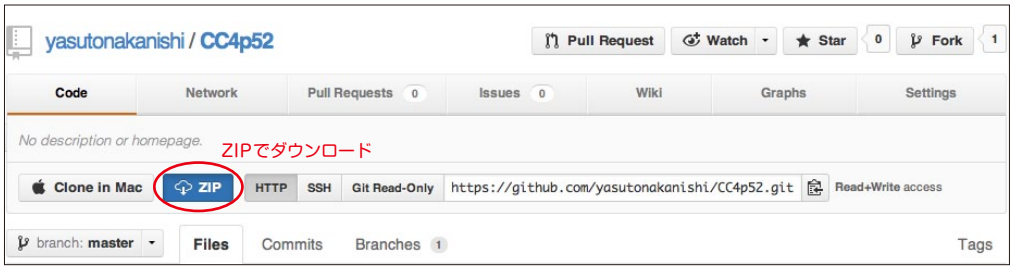
・Eclipse（ http://www.eclipse.org/ ）
・JDK（ Java Development Kit: http://www.oracle.com/technetwork/java/javase/downloads/index.html ）
・Processing（ http://processing.org/ ）
・CityCompilier（ https://github.com/yasutonakanishi/ ）

CityCompilerではJavaのプログラミングを行うため、EclipseとJDKが必要です。 EclipseはJavaやC++など各種プログラミングを行うための統合開発環境です。 JDKはJavaのプログラミングを行うための開発キットです。 Windowsの場合は、日本語化されたEclipseにJDKを同梱したセット（Pleiades All in One: http://mergedoc.sourceforge.jp/）が配布されていますので、こちらを利用するのが便利です。 Processingはメディアアートやビジュアルデザインのためのプログラミング言語およびその統合開発環境です。v2.0を使うことができます。 まずはEclipseとJDK、Processingをインストールしてください。

N° 2

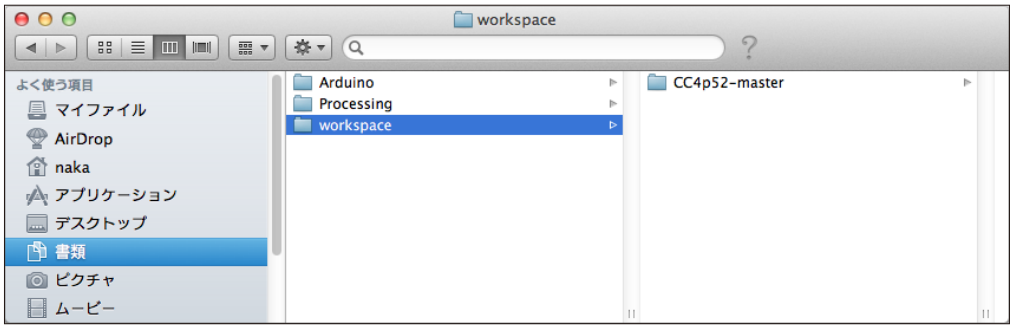
CityCompilerのセットアップ

① CityCompilerの配布ページ（GitHub: https://github.com/yasutonakanishi/CC4p52 ）にアクセスし、「ZIP」ボタンを押してzip形式でファイル一式をダウンロードします。

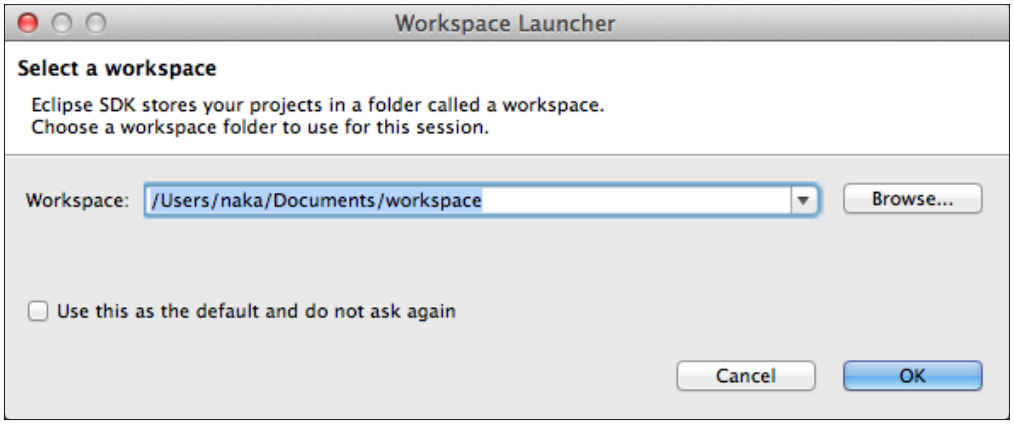


② ダウンロードしたZIPファイル（CC4p52-master.zip）を解凍します。

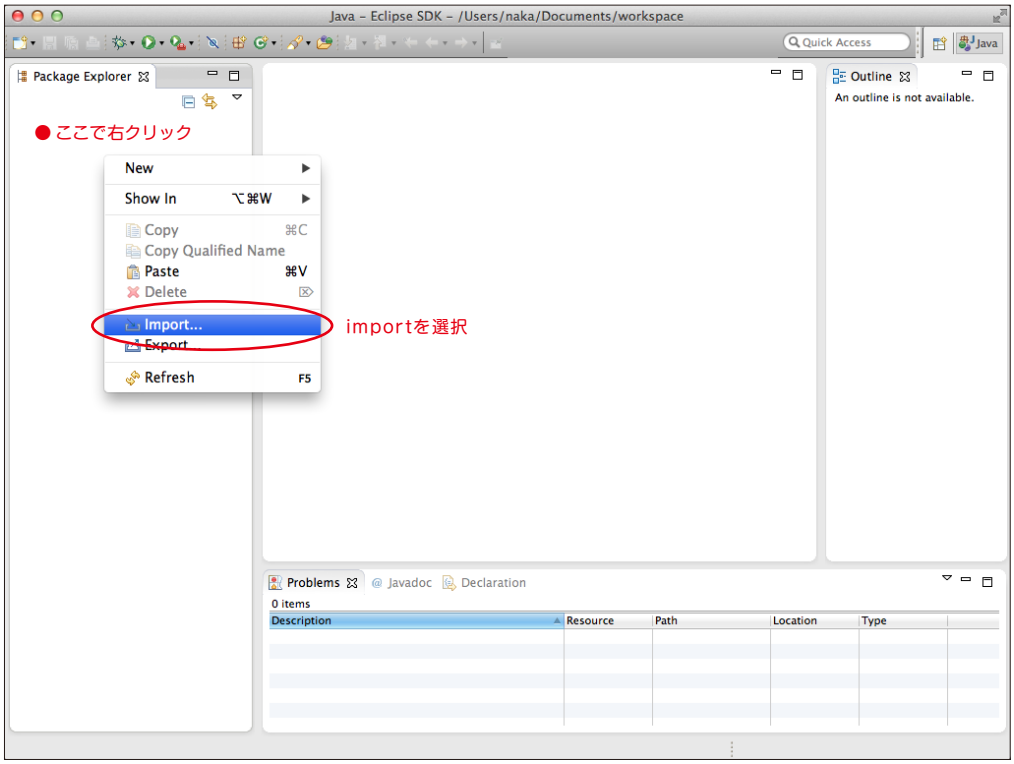
③ Eclipseのワークスペース（作業用のディレクトリ）を任意の場所に作り、そこに解凍してできた「CC4p52-master」というフォルダを移動します。ワークスペースはWindowsであれば「C:\workspace」、Macであれば「/Users/ユーザ名/Documents/workspace」などに作ります。



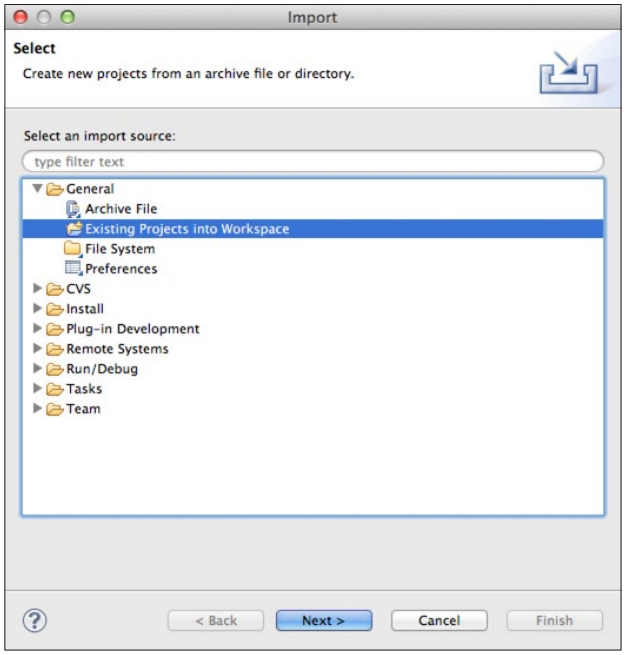
④ Eclipseを起動し、ワークスペースを選択します。



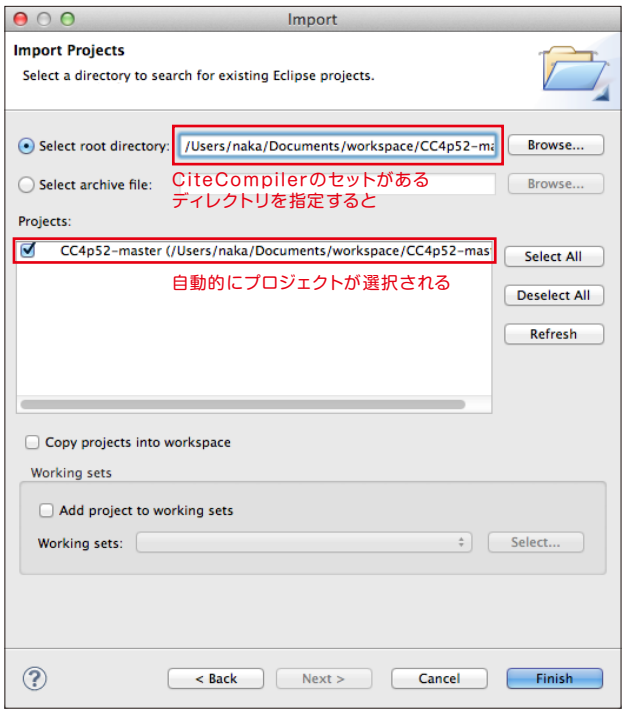
⑤ Project Explorerのところで右クリックし、[Import] を選択します。



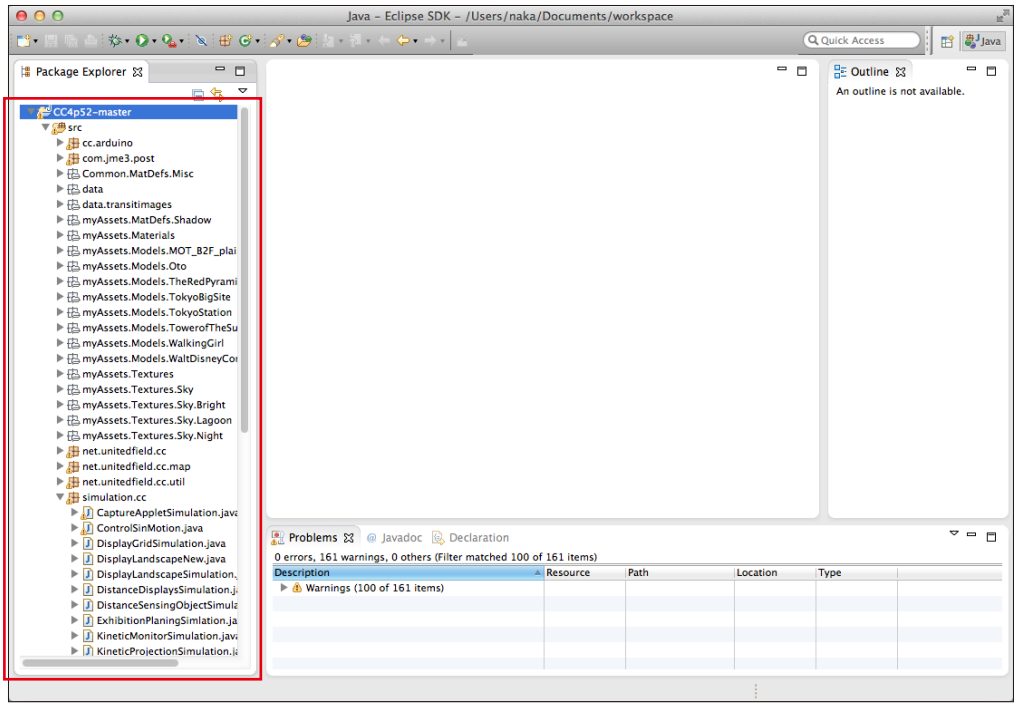
⑥ 出てきたダイアログでは、[General] - [Existing Projects into Workspace] を選択して [Next] を押します。



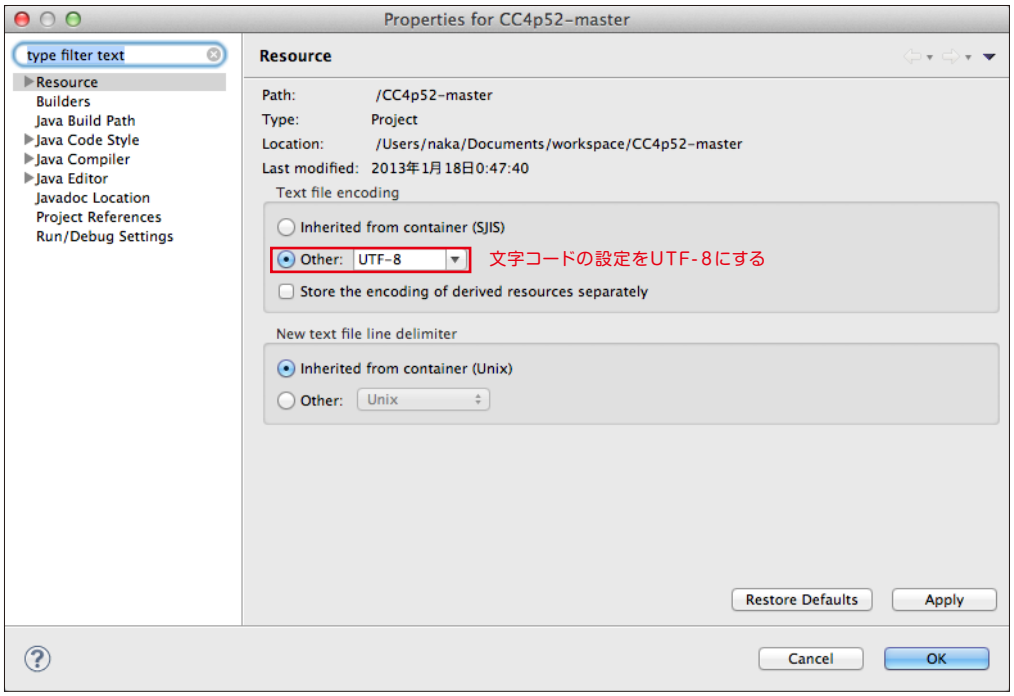
⑦ [Select root directory:] という項目で、先ほど「CC4p52-master」を置いたディレクトリを指定します。プロジェクトが検出されると、一覧のところに「CC4p52-master」が現れます。プロジェクトが選択された状態なのを確認したら [Finish] ボタンを押します。



⑧ 以下のようにプロジェクトが追加されたらOKです。



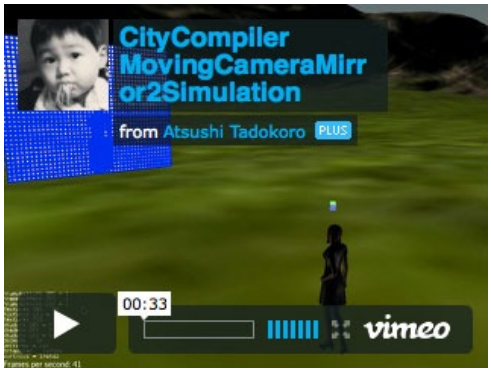
⑨ Macの場合、もしエラーが出ていたら、プロジェクト(CC4p52-master)を右クリックしてメニューから[Properties]を選択し、出てきた画面で文字エンコードを「SJIS」から「UTF-8」に変更してください。



N°. 3

■ 既存のサンプルの実行

動作確認を兼ねて既存のサンプルプログラムをいくつか実行してみましょう。 プロジェクト・エクスプローラーで [CC4p52-master] - [src] - [simulation.cc] と辿り、そこに並んでいるいずれかのjavaファイルをダブルクリックします。 ファイルを開いたら、実行ボタンを押してください。実行するとJMEの設定ウィンドウが表示される場合がありますが、そのまま [Ok] ボタンを押してください。



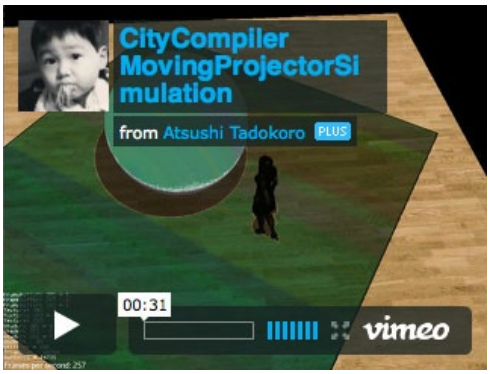
MovingCameraMirror2Simulation
動くカメラと画像処理のサンプル
http://vimeo.com/54132143

vDisplayGridSimulation
上空にたくさんのディスプレイ
http://vimeo.com/54132080

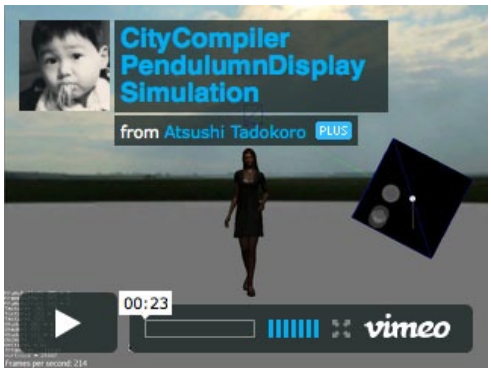
ShadowProjection2DisneyHallSimulation
ディズニーホールにプロジェクションマッピング
http://vimeo.com/54132925

ShadowProjection2BigsightSimulation
東京ビッグサイトにプロジェクションマッピング
http://vimeo.com/54132641

LakeDisplaySimulation
湖畔に映り込む球体ディスプレイ
http://vimeo.com/54132145

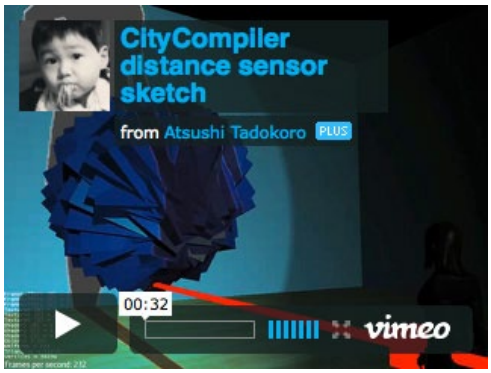


MovingProjectorSimulation
動くプロジェクタのサンプル
http://vimeo.com/54132144

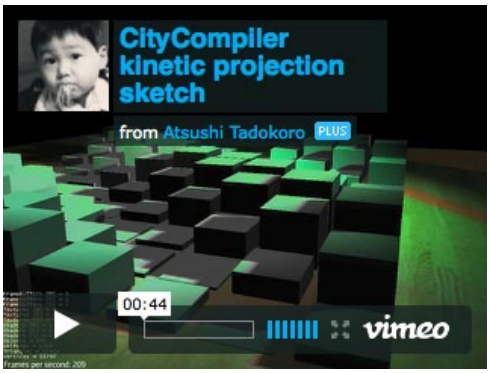


PendulumnDisplaySimulation
振り子状ディスプレイとそれに連動した物理演算
http://vimeo.com/54134052

ExhibitionPlaningSimlation
ディスプレイを美術館風に配置
http://vimeo.com/54133977



DistanceSensingObjectSimulation
距離センサと連動したオブジェ
http://vimeo.com/48711689



KineticProjectionSimulation
うにうに変形する物体にプロジェクション
http://vimeo.com/48711452

KineticMonitorSimulation
色を変化させながらうにうに変形する物体
http://vimeo.com/48710981

DisplayLandscapeNew
任意の場所にディスプレイを配置
http://vimeo.com/48710794

CityCompilerを使ったプログラミングの全体像
Overview of CityCompiler

プログラミングをはじめる前に全体の概要について説明します。これが把握できていると以降のプログラミングもスムーズに進められるでしょう。

No. 1

書くのは2種類のコード

CityComplierを使ったプログラミングでは、大きく分けて2種類のコードを書きます。グラフィックスに関するコードと、3Dバーチャル空間に関するコードです。

グラフィックスに関するコード（Processing）

```
setup () {
 …
}
draw () {
 …
}
```

3Dバーチャル空間に関するコード（JMEとCityCompiler）

```
simpleIntApp () {
 …
}
simpleIUpdate () {
 …
}
destroy () {
 …
}
main () {
 …
}
```

CityCompilerプログラミングで書く2種類のコード

グラフィックスに関するコードでは、ディスプレイやプロジェクタに表示する2Dまたは3Dのグラフィックスに関する処理を記述します。このコードはProcessingを使って記述します。グラフィックス処理の記述にProcessingを使うことのメリットは、OpenProcessing（http://www.openprocessing.org/）などで共有されている既存のコードを再利用でき、なおかつバーチャル空間と現実世界の双方で同じコードを利用することができるという点です。

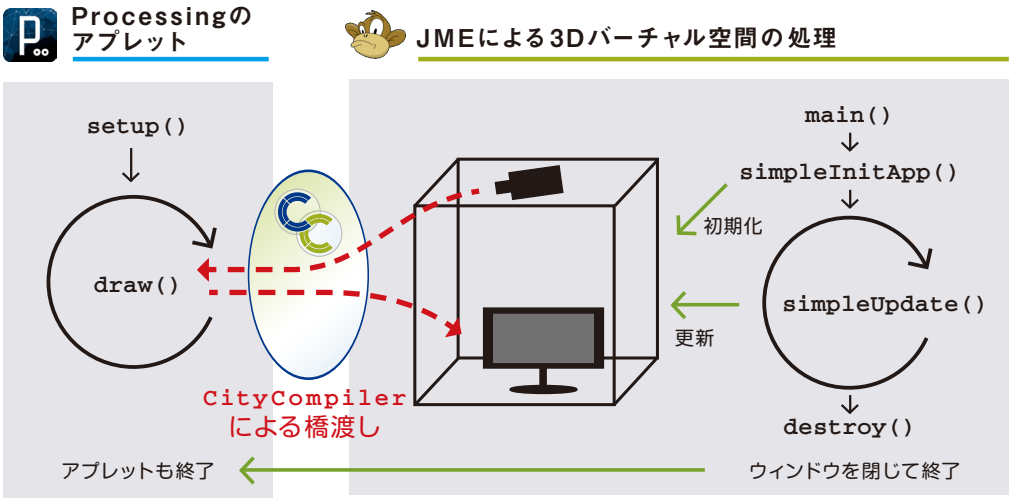
3Dバーチャル空間に関するコードでは、草原や室内などの箱庭的な空間を作り、カメラやプロジェクタをどこに置くのか、それらがどのように動くのかを記述します。バーチャル空間に関する処理は jMonkeyEngine（JME: http://jmonkeyengine.com/）というJava用のゲームエンジンライブラリとCityComplierを使って記述します。

これら2つのコードはいずれもJava形式のファイルです。グラフィックスに関するコードはProcessingのIDE上で書いたものをJava形式に変換して使います。変換と言っても、Processingは内部的にはJavaとして動作しているため、元のコードとほぼ同等のコードが生成されます。

N° 2

全体の処理の流れ

全体の処理構造を下図に示します。JMEによる3Dバーチャル空間の処理とProcessingのアプレットという2種類のスレッドが動作しています。



CityCompilerのプログラムにおける全体の処理の流れ

**Processingで書く
グラフィック側のコード**

```
public class MyPApplet extends PApplet {
  public void setup() {
  ...
  }

  public void draw() {
  ...
  }
}
```

**jMEを使って書く
バーチャル空間側のコード**

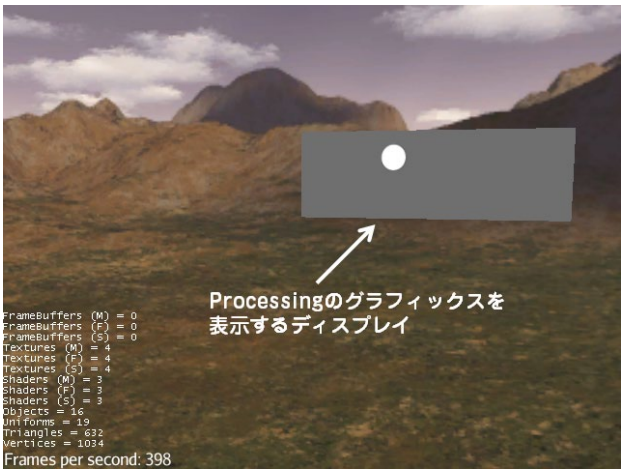
```
public class MySimulation extends SimpleApplication {
  public static void main(String[] args) {
        SimpleApplication app = new MySimulation();
        app.start();
  }
  public void simpleInitApp() {
  ...
  }

  public void simpleUpdate(float tpf) {
        .../* 毎フレーム行う処理をここに記述 */
  }

  public void destroy() {
  ...
  }
}
```

プログラムを実行すると、まずsimpleInitApp()というメソッドによってバーチャル空間の初期化が行われます。その後、終了の合図があるまでsimpleUpdate()によってバーチャル空間の更新が繰り返し行われます。バーチャル空間の初期化時には、カメラやディスプレイなどのオブジェクトを作成しますが、この時、それらに対してProcessingのアプレットを紐づけます。Processing側では、setup()での初期化後、終了の合図があるまで

draw()による描画処理が繰り返し実行されます。
バーチャル空間内のカメラが撮影した画像をProcessing側のCaptureに渡したり、逆にProcessing側でimage()で描画したグラフィックスの画像をバーチャル空間のディスプレイに渡したり、という「橋渡し」をやるのがCityComplierの役目です。

ゼロからプログラムを書いてみよう　Let's write a source code

いちばん簡単なサンプルとして、バーチャル空間に1つのディスプレイを置き、そこにProcessingで作られたグラフィックスを表示する、というのに挑戦してみましょう。これで基本的なプログラミングの手順を習得しましょう。



こんな感じのを作ります

N° 1

Processingでコードを書こう

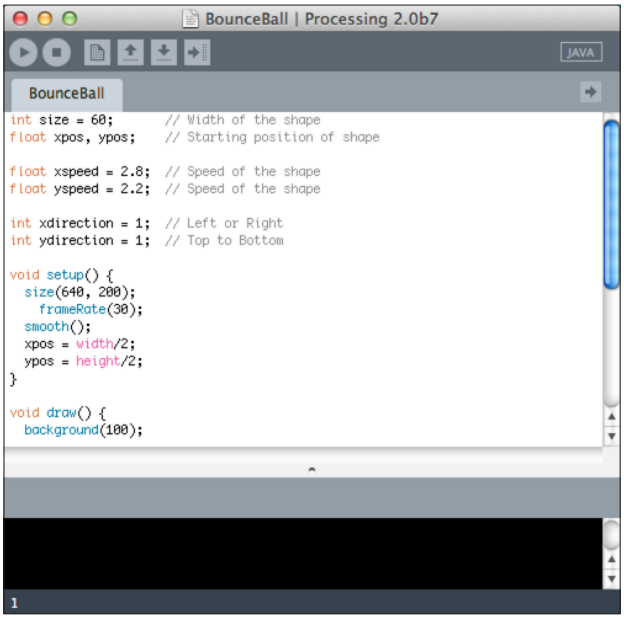
まずはディスプレイに表示するグラフィカルなコンテンツをProcessingで作りましょう。以下は画面上に表示されたボールが跳ねるコードです。「BounceBall」という名前でスケッチを保存してください。

BounceBall.pde

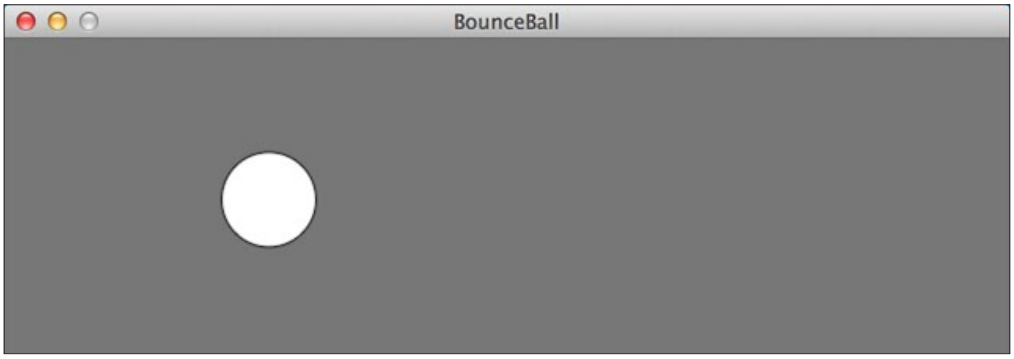
```
int size = 60;       // Width of the shape
float xpos, ypos;    // Starting position of shape

float xspeed = 2.8;  // Speed of the shape
float yspeed = 2.2;  // Speed of the shape

int xdirection = 1;  // Left or Right
int ydirection = 1;  // Top to Bottom

void setup() {
  size(640, 200);
```

```
12      frameRate(30);
13    smooth();
14    xpos = width/2;
15    ypos = height/2;
16  }
17
18  void draw() {
19    background(100);
20
21    xpos = xpos + ( xspeed * xdirection );
22    ypos = ypos + ( yspeed * ydirection );
23
24    if (xpos > width-size || xpos < 0) {
25      xdirection *= -1;
26    }
27    if (ypos > height-size || ypos < 0) {
28      ydirection *= -1;
29    }
30
31    ellipse(xpos+size/2, ypos+size/2, size, size);
32  }
```

まずはこれをProcessingで書いて動かしてみましょう。Processingを起動してソースコードをコピペしたら、[Run]ボタンを押して実行してください。



実行するとこんな感じのウィンドウが表示されます。

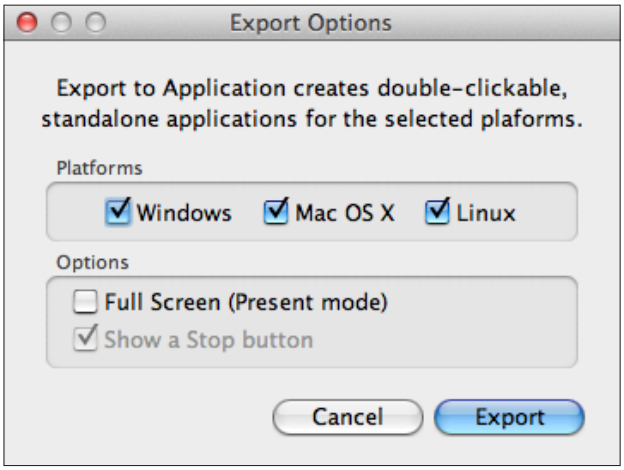


ボールが飛んで画面の端で跳ね返ります

N°. 2

EclipseでProcessingのコードを動かそう

Processingで実行してみて動くのが確認できたら、Javaのコードに変換します。Processingの [File] メニューから [Export Application] を選択してください。出てきたダイアログでいま使用しているOSにチェックが入っているのを確認したら、[Export] ボタンを押してください。



Exportすると、スケッチを保存したフォルダの中に「application.windows32」や「application.macosx」という名前のフォルダが自動的に作られ、OSごとの実行ファイルが生成されます。そして、さらにその中にある「source」というフォルダの中にjava形式のソースコードが入っています。ここでのお目当てはjava形式のデータです。

先ほどのコードを変換して得られるBounceBall.javaは以下のようなコードになっています。

BounceBall.java

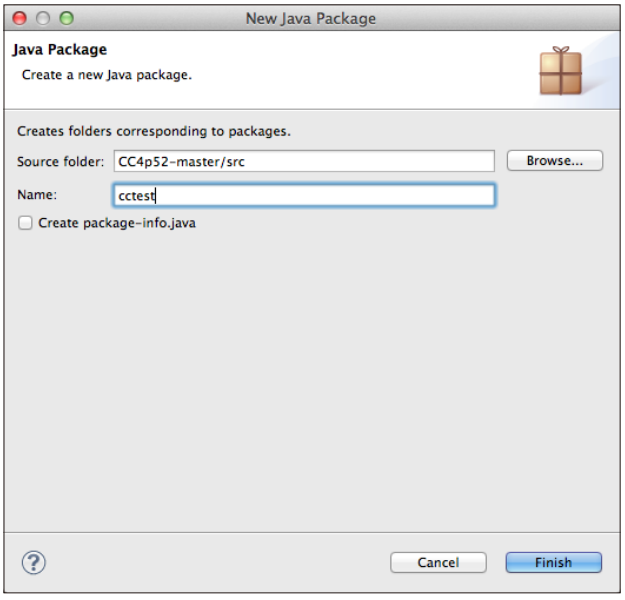
```
1   import processing.core.*;
2   import processing.xml.*;
3
4   import java.applet.*;
5   import java.awt.Dimension;
6   import java.awt.Frame;
7   import java.awt.event.MouseEvent;
8   import java.awt.event.KeyEvent;
9   import java.awt.event.FocusEvent;
10  import java.awt.Image;
11  import java.io.*;
12  import java.net.*;
13  import java.text.*;
14  import java.util.*;
15  import java.util.zip.*;
16  import java.util.regex.*;
17
18  public class BounceBall extends PApplet {
19
20  int size = 60;       // Width of the shape
21  float xpos, ypos;    // Starting position of shape
22
23  float xspeed = 2.8f;  // Speed of the shape
24  float yspeed = 2.2f;  // Speed of the shape
25
26  int xdirection = 1;  // Left or Right
```

```
27  int ydirection = 1;  // Top to Bottom
28
29  public void setup() {
30    size(640, 200);
31    frameRate(30);
32    smooth();
33    xpos = width/2;
34    ypos = height/2;
35  }
36
37  public void draw() {
38    background(100);
39
40    xpos = xpos + ( xspeed * xdirection );
41    ypos = ypos + ( yspeed * ydirection );
42
43    if (xpos > width-size || xpos < 0) {
44      xdirection *= -1;
45    }
46    if (ypos > height-size || ypos < 0) {
47      ydirection *= -1;
48    }
49
50    ellipse(xpos+size/2, ypos+size/2, size, size);
51  }
52    static public void main(String args[]) {
53      PApplet.main(new String[] { "--bgcolor=#F0F0F0", "BounceBall" });
54    }
55  }
```

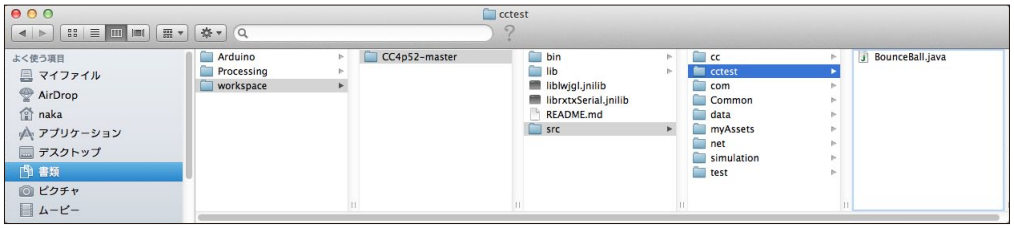
Processingで書いたコードの前後にいろいろと追加されているのがわかります。何をやっているのかというと、Javaとして動かすためにPApplet型の派生クラスにしています。また、そのために必要なライブラリをimportし、実行できるようにエントリポイント（static public void main(String args[])）を追加しています。すなわち、Processingは内部でこういうコードに変換してからJavaとして実行していたわけです。

ではこのJavaのコードをEclipse上で動かしてみましょう。

① Eclipseを起動し、CityCompilerのプロジェクトフォルダがあるワークスペースを開きます。
② プロジェクト[CC4p52-master]を右クリックし、[New]-[Package]を選択します。
③ 名前に適当なパッケージ名を付けます。ここでは「cctest」とします。



④ この操作によって、ワークスペースの中に「CC4p52-master/src/cctest」というディレクトリが自動的に作られますので、そこに先ほど生成した BounceBall.java をコピーします。

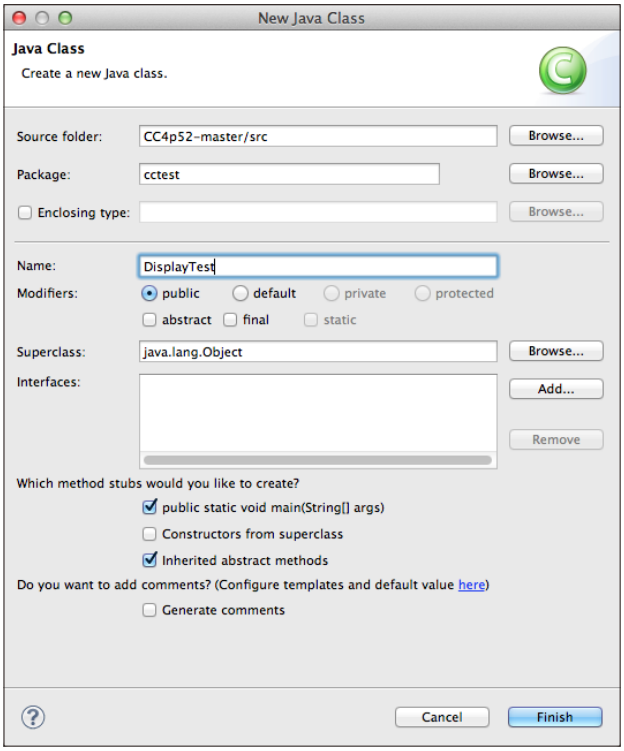


N° 3

EclipseでProcessingのコードを動かそう

ここからは新しくクラスを作り、バーチャル空間側のコードを書いていきます。

① 先ほど自分で作ったパッケージ（cctest）の上で右クリックし、[New] - [Class]を選択します。
② [Name] という項目で適当な名前を付けます。ここでは「DisplayTest」という名前を付けてください。
③ [Superclass]ではcom.jme3.app.SimpleApplicationを指定します。
④ [Modifiers]では[public]を選択します。
⑤ [public static void main(String[] args)]と[Inherited abstract methods]にチェックを入れます。
⑥ 以上の設定ができたら、[Finish] ボタンをクリックします。



この作業を終えると、ベースとなる何もしないプログラムが生成されます。ここに自分でコードを書いていくことになります。

DisplayTest.java

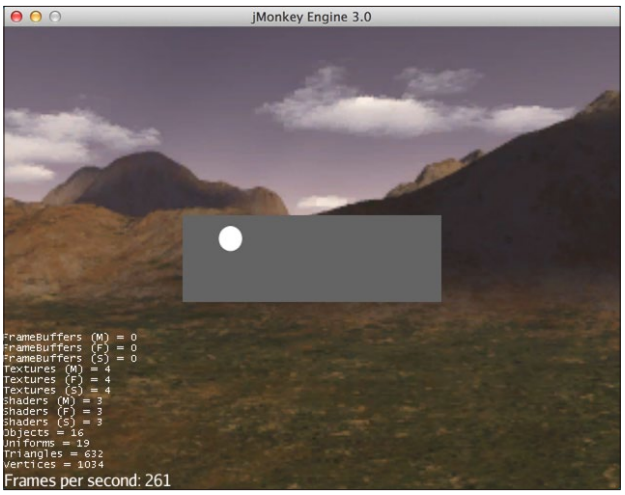
```
1   package cctest;
2
3   import com.jme3.app.SimpleApplication;
4
5   public class DisplayTest extends SimpleApplication {
6
7       @Override
8       public void simpleInitApp() {
9           // TODO Auto-generated method stub
10
11      }
12
13      /**
14       * @param args
15       */
16      public static void main(String[] args) {
17          // TODO Auto-generated method stub
18
19      }
20
21  }
```

では以下のようにコードを書き加えてみましょう。

DisplayTest.java

```
1   package cctest;
2
3   import com.jme3.app.SimpleApplication;
4   import com.jme3.util.SkyFactory;
5   import net.unitedfield.cc.PAppletDisplayGeometry;
6
7   public class DisplayTest extends SimpleApplication {
8
9       public void simpleInitApp() {
10
11          // 空を作成
12          rootNode.attachChild(SkyFactory.createSky(assetManager, "Textures/Sky/Bright/
            BrightSky.dds", false));
13
14          // Processing のアプレットを作成
15          BounceBall applet = new BounceBall();
16
17          // ディスプレイを作成し、そこに Processing のアプレットを設定
18          PAppletDisplayGeometry display = new PAppletDisplayGeometry("display",
            assetManager, 6, 2, applet, 640, 200, true);
19          rootNode.attachChild(display);
20          display.setLocalTranslation(0, 0, -3);
21  }
22
23      public void simpleUpdate(float tpf) {
24          /* 毎フレーム行う処理をここに記述 */
25      }
26
27      public void destroy() {
28          super.destroy();
29          System.exit(0);
30      }
31
32      public static void main(String[] args) {
33          SimpleApplication app = new DisplayTest();  // SimpleApplication のインスタンス
                                                           の生成
34          app.setPauseOnLostFocus(false);             // フォーカスがロストした場合にポーズし
                                                           ない設定
35          app.start();                                // シミュレーションのスタート
36      }
37  }
```

実行すると、以下のようになります。荒野の中にProcessingのプログラムが走っているディスプレイが鎮座しているシュールな絵になりました。マウスの移動で首振り、ホイールで前後の移動ができます。また、キーボードのAとDで左右、WとSで前後の移動をし、方向キーで首振りができます。



N° 4

バーチャル空間側のコードを理解しよう

このコードは、main()、simpleInitApp()、simpleUpdate()、destroy()の４つのメソッドから構成されています。それぞれの役割を細かく見ていきましょう。

main() からスタート

main()が最初に処理が実行される場所です。ここではSimpleApplicationのインスタンスを生成し、app.start()によってシミュレーションをスタートさせます。また、必要に応じてウィンドウの挙動に関するオプションを設定します。ここでは画面からフォーカスが失われた場合でもポーズせずに描画処理を続行する設定を行っています。

main()の中身

```
1  public static void main(String[] args) {
2          SimpleApplication app = new DisplayTest();  // SimpleApplicationのインスタンスの生成
3          app.setPauseOnLostFocus(false);             // フォーカスがロストした場合にポーズしない設定
4          app.start();                                // シミュレーションのスタート
5  }
```

simpleInitApp() で空間の構成要素を設定

simpleInitApp()がバーチャル空間の初期化時に実行されるメソッドです。ここに「何をどこ置くのか」に関する処理を記述します。 上のコードでは、「空」と「ディスプレイ」を空間に追加しています。空間にオブジェクトを追加する処理を行っているのがrootNode.attachChild()という部分です。空を作っては空間に追加、ディスプレイを作っては空間に追加、という感じでいろんな要素を空間に追加していきます。

simpleInitApp()

```
1   public void simpleInitApp() {
2
3       // 空を作成
4       rootNode.attachChild(SkyFactory.createSky(assetManager, "Textures/Sky/Bright/
        BrightSky.dds", false));
5
6       // Processing のアプレットを作成
7       BounceBall applet = new BounceBall();
8
9       // ディスプレイを作成し、そこに Processing のアプレットを設定
10      PAppletDisplayGeometry display = new PAppletDisplayGeometry("display",
            assetManager, 6, 2, applet, 640, 200, true);
11      rootNode.attachChild(display);
12      display.setLocalTranslation(0, 0, -3);
13  }
14
```

ここでいちばん大事なのが、Processingの処理結果を表示するディスプレイを作成しているところです。まず、BounceBall applet = new BounceBall()とやってProcessingのアプレットのインスタンスを作成します。次に、PAppletDisplayGeometryによってディスプレイを作成します。コンストラクタの第2・第3引数が空間中での物理サイズです。物理サイズの単位はメートルです。第4引数にProessingのアプレットを指定し、第5・第6引数にProcessing側でsize()で指定している値を与えます。最後の引数でアプレットを別のウィンドウとして表示するかどうかを指定します。これらの設定によってディスプレイに対してProcessingのアプレットが紐づけられます。作ったディスプレイはrootNode.attachChild()によって空間に追加します。そして最後に、display.setLocalTranslation()によって 空間中の位置を設定します。

PAppletDisplayGeometryのコンストラクタの引数

```
1   PAppletDisplayGeometry( String       name,          // 名前
2                           AssetManager assetmanager,  // assetManager
3                           float        width,         // バーチャル空間内におけるディスプレイの
                                                           横幅
4                           float        height,        // バーチャル空間内におけるィスプレイの高
                                                           さ
5                           PApplet      applet,        // Processing のアプレット
6                           int          appletWidth,   // Processing のアプレットの画面の横幅
7                           int          appletHeight,  // Processing のアプレットの画面の高さ
8                           boolean      frameVisible   // 別ウィンドウでアプレットの実行結果を
                                                           表示するか
9                       );
```

simpleUpdate() には毎フレームの処理を記述

このプログラムでは特に処理を記述していませんが、simpleUpdate()にはバーチャル空間が更新されるタイミングで実行したい処理を記述します。例えば、ディスプレイやカメラが空間中を移動するアニメーションを作りたい場合、ここにその動き方を記述します。引

数の tpf は time per frame の略でフレーム間の経過時間を意味します。描画にかかる時間は変動しますので、物体を一定の速度で動かしたければ、tpfに比例した移動量を与えてください。

simpleUpdate()

```
1   public void simpleUpdate(float tpf) {
2       /* 毎フレーム行う処理をここに記述 */
3   }
```

終了処理はdestroy()という名前のメソッドを作ってそこに記述します。このメソッドが終了時に自動的に呼ばれます。慣例的にsuper.destroy()とSystem.exit(0)の２つをやると覚えてしまってOKです。

destroy()

```
1   public void destroy() {
2       super.destroy();
3       System.exit(0);
4   }
```

N° 5

■ うまく動きましたか？

以上がCityCompilerを使ったプログラミングの基本的な流れです。これがわかっていれば、既存のサンプルコードを切ったり貼ったりして新しいものが作れるはずです。既存のコードのパッケージ構成は以下のようになっています。

・「simulation.cc」- 応用作品のソースコードが入っています。
・「simulation.p5」- simulation.*の中で使われているProcessingのアプレットのソースコードが入っています。
・「simulaiton.workshop」- ワークショップの説明で使うソースコードが入っています。
・「test.cc」- ディスプレイ、カメラ、プロジェクタ、距離センサについての簡単なサンプルが入っています。
・「test.p5」- test.*の中で使われているProcessingのアプレットのソースコードが入っています。
・「test.jme」- JMEのサンプルが入っています。ここに入っているコードにはCityCompilerのコードは含まれていません。

最初のうちは、simulationやtest.ccに入っているコードで呼ばれているProcessingのアプレットを他のものに差し替えたり、simpleUpdate()の中を適当に書き変えてディスプレイやプロジェクタの動き方を変更したり、いろいろやってみましょう。

カメラを使ってみよう　Let's use virtual camera

バーチャル空間にカメラとディスプレイを置いて、カメラで撮影した結果をそのままディスプレイに表示させるのをやってみましょう。

■ バーチャル空間側のコード

さきほど紹介したディスプレイを置くだけのサンプルにカメラを追加しています。また、simpleUpdate()でカメラを回転させています。

CameraTest.java

```
1  package cctest;
2
3  import com.jme3.app.SimpleApplication;
4  import com.jme3.util.SkyFactory;
5  import net.unitedfield.cc.PAppletDisplayGeometry;
6  import net.unitedfield.cc.CaptureCameraNode;
7
8
9  public class CameraTest extends SimpleApplication {
10
11     private CaptureCameraNode captureCameraNode; // カメラ
12
13     // 初期化
14     public void simpleInitApp() {
15
16         // 空を作る
17         rootNode.attachChild(SkyFactory.createSky(assetManager, "Textures/Sky/Bright/
           BrightSky.dds", false));
18
19         // Processing のアプレット
20         CameraPApplet applet = new CameraPApplet();
21
22         // ディスプレイを作る
23         PAppletDisplayGeometry display = new PAppletDisplayGeometry("display",
           assetManager, 4, 3, applet, 320, 240, true);
24         rootNode.attachChild(display);
25         display.setLocalTranslation(0, 0, -3);
26
27         // カメラを作る
28         captureCameraNode = new CaptureCameraNode("cameraNode", 320, 240,
           assetManager, renderManager, renderer, rootNode);
29         rootNode.attachChild(captureCameraNode);
30         captureCameraNode.setLocalTranslation(0.0f, 0.5f, 8.0f); // カメラの位置を設定
31         if (applet.realDeployment==false) {
32             applet.setCapture(captureCameraNode.getCapture()); // Processing のキャプチャを設定
```

```
        }
    }

    // 更新処理
    public void simpleUpdate(float tpf) {
        captureCameraNode.rotate(0f, 0.2f*tpf, 0f); // カメラの回転
    }

    // 終了処理
    public void destroy() {
        super.destroy();
        System.exit(0);
    }

    // メイン
    public static void main(String[] args) {
        SimpleApplication app = new CameraTest();
                app.start();
    }

}
```

グラフィックス側のコード

このコードは、バーチャル空間に置かれるバーチャルカメラと、PCに接続されているリアルカメラの両方を扱うコードになっています。realDeploymentという変数で本物を使うかどうかを切り変えていて、realDeployment = trueのときにリアルカメラを使います。リアルカメラの場合は new Capture(this, width, height) によってカメラを初期化し、video.startによってキャプチャをスタートさせます。バーチャルカメラの場合は、setCapture()という自前のメソッドで外部（バーチャル空間側）から与えられるキャプチャを設定します。

CameraPApplet.java

```
package cctest;

import processing.core.*;
import processing.video.*;

public class CameraPApplet extends PApplet {

    public boolean realDeployment = false;
    Capture video = null;
    PImage videoImage = null;

    public void setup() {
        size(320, 240, P2D);

        if (realDeployment) {
            video = new Capture(this, width, height);
            video.start();
        }

        videoImage = new PImage(width,height);
    }

    public  void setCapture(Capture capture){
        this.video = capture;
    }

    public void draw() {
          video.read();

          // カメラ画像の表示
          //this.image(video, 0, 0);
          video.loadPixels();
          arrayCopy( video.pixels, videoImage.pixels);
          videoImage.updatePixels();
          image(videoImage,0,0);

          // 白い太枠を表示
          strokeWeight(15);
          stroke(255);
          noFill();
          rect(0,0,width,height);
    }

    public static void main(String[] args){
```

```
45          PApplet.main(new String[] { "--bgcolor=#c0c0c0", "CameraApplet" });
46      }
47  }
```

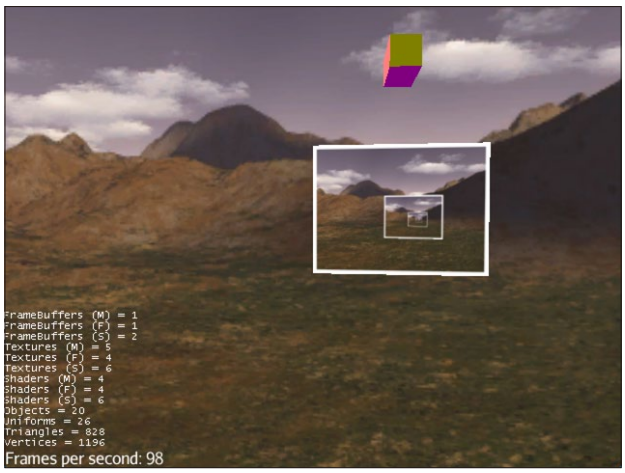
カメラの使い方

カメラはCaptureCameraNodeで作ります。注意すべきポイントは、第2・第3引数で設定されるカメラの画像サイズです。これは必ずProcessing側で設定しているサイズと同じにしてください（ここでは320, 240）。カメラを作ったらrootNode.attachChild()で空間に追加し、.setLocalTranslation()で位置を決めます。最後に、バーチャル空間にあるカメラのデータをProcessing側で処理できるようにするために、.getCapture()によってCaptureを取得し、setCapture()でProcessing側に渡します。

カメラを作るコード

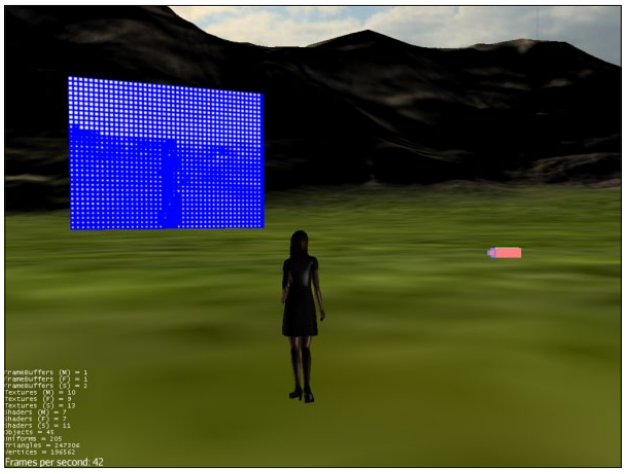
```
1  // カメラを作る
2  captureCameraNode = new CaptureCameraNode("cameraNode", 320, 240, assetManager,
   renderManager, renderer, rootNode);
3  rootNode.attachChild(captureCameraNode);
4  captureCameraNode.setLocalTranslation(0.0f, 0.5f, 8.0f); // カメラの位置を設定
5  if (applet.realDeployment==false) {
6      applet.setCapture(captureCameraNode.getCapture()); // Processing のキャプチャを設定
7  }
```

このサンプルでは、カメラの映像をそのままディスプレイに出しています。ディスプレイとカメラが向かい合わせになった時に起こる「合わせ鏡」現象もしっかり再現されます。



ディスプレイとカメラの「合わせ鏡」現象！

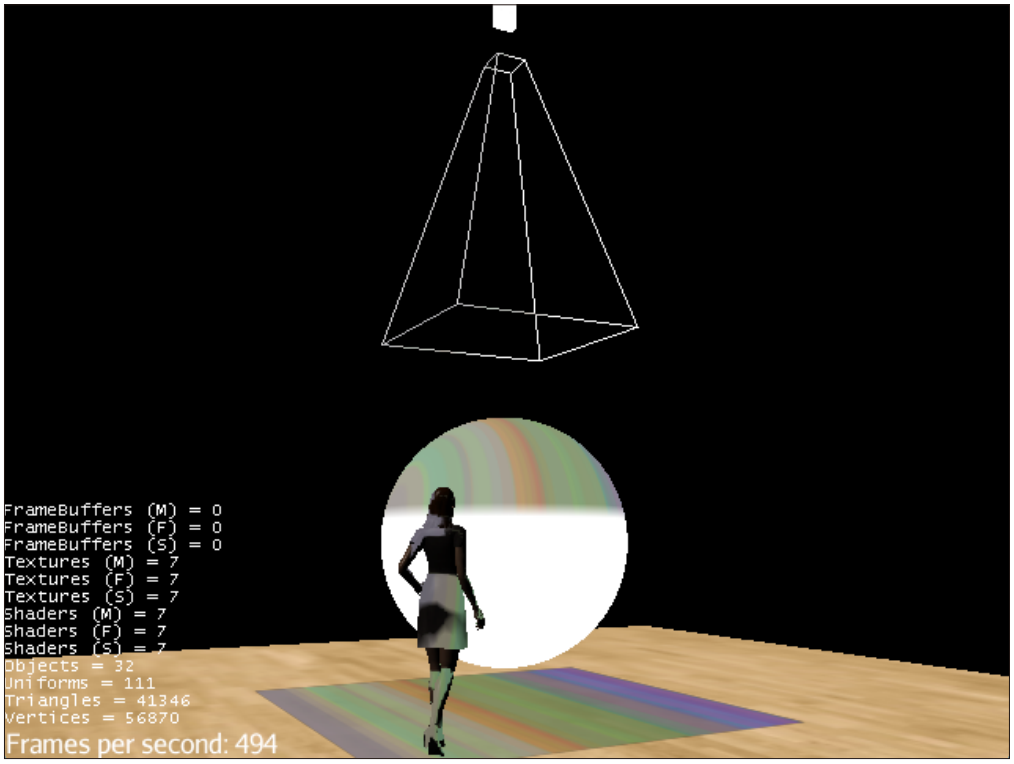
もちろんなんらかの画像処理を行った結果を表示することもできます。 画像処理をやりたいときは MovingCameraMirror2Simulation というサンプルが参考になります。このサンプルで呼び出されているProcessingのアプレットのコードはMirror2PAppletです。



カメラの回転と画像処理を行うサンプル
（MovingCameraMirror2Simulation）

プロジェクタを使ってみよう　Let's use virtual projector

プロジェクタがあればプロジェクションマッピングが作れるようになります。 また、単純にディスプレイをプロジェクタに置き換えるだけでもいきなり楽しくなります。 ここではプロジェクタ、床、物体、人のモデルなどを出す方法を紹介します。



通常版プロジェクタでの実行結果

■ バーチャル空間側のコード

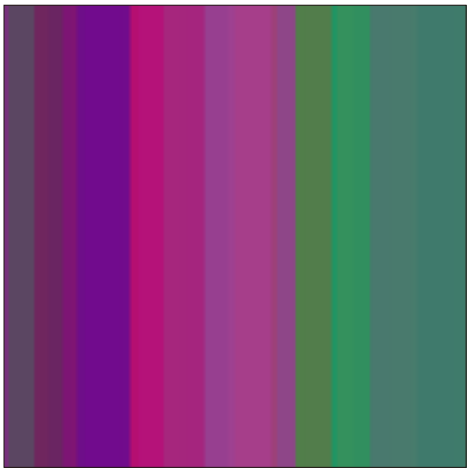
CameraPApplet.java

```
1  package cctest;
2
3  import processing.core.PApplet;
4  import test.p5.ColorBarsPApplet;
5  import net.unitedfield.cc.PAppletProjectorNode;
6  import net.unitedfield.cc.PAppletProjectorShadowNode;
7
8  import com.jme3.app.SimpleApplication;
9  import com.jme3.light.DirectionalLight;
10 import com.jme3.material.Material;
11 import com.jme3.math.Vector3f;
12 import com.jme3.post.TextureProjectorRenderer;
13 import com.jme3.renderer.queue.RenderQueue.Bucket;
14 import com.jme3.renderer.queue.RenderQueue.ShadowMode;
15 import com.jme3.scene.Geometry;
16 import com.jme3.scene.Spatial;
17 import com.jme3.scene.shape.Box;
18 import com.jme3.scene.shape.Sphere;
19
20
21 public class ProjectorTest extends SimpleApplication  {
22
23     // 初期化
24     public void simpleInitApp() {
25
26         // Processing のアプレットの作成
27         PApplet applet = new ColorBarsPApplet();
28
29         // プロジェクタの設定（通常版）
30         PAppletProjectorNode projector = new PAppletProjectorNode("projector0",
           assetManager, applet, 200, 200, false);
31         rootNode.attachChild(projector);                          // 空間にプロジェクタ
           を追加
32         rootNode.attachChild(projector.getFrustmMdel());          // プロジェクタの視野
           角を表示
33         projector.setLocalTranslation(new Vector3f(0,6,0));       // プロジェクタの位置
34         projector.lookAt(new Vector3f(0, 0, 0), Vector3f.UNIT_X); // プロジェクタの注視
           点
35         TextureProjectorRenderer ptr = new TextureProjectorRenderer(assetManager); //
           このプロジェクタ用のレンダリングの設定
36         ptr.getTextureProjectors().add(projector.getProjector());
37         viewPort.addProcessor(ptr);
38
39         /*
40         // プロジェクタの設定（Shadow 版）
41         PAppletProjectorShadowNode projector = new PAppletProjectorShadowNode(
           "Projector0", viewPort, assetManager, 1024, 1024, applet, 200, 200, false);
42         rootNode.attachChild(projector);                          // 空間にプロジェクタ
           を追加
43         projector.setLocalTranslation(new Vector3f(0,6,0));       // プロジェクタの位置
44         projector.lookAt(new Vector3f(0, 0, 0), Vector3f.UNIT_X); // プロジェクタの注視
           点
45         */
46
47         // 照明
48         DirectionalLight dl = new DirectionalLight();
49         dl.setDirection(new Vector3f(-0.1f, -1f, -1).normalizeLocal());
50         rootNode.addLight(dl);
51
52         // 床面
53         Material textureMat = new Material(assetManager, "Common/MatDefs/Misc/
           Unshaded.j3md");
54         textureMat.setTexture("ColorMap", assetManager.loadTexture("myAssets/Textures/
           woodFloor.jpg"));
55         Box floor = new Box(Vector3f.ZERO, 5.0f, 0.01f, 5.0f);
56         Geometry floorGeom = new Geometry("Floor", floor);
57         floorGeom.setMaterial(textureMat);
58         rootNode.attachChild(floorGeom);
59
60         // 球
61         Material whitemat = assetManager.loadMaterial("Common/Materials/WhiteColor.
           j3m");
62         Sphere sp = new Sphere(64, 64, 1.0f);
63         Geometry sphereGeom = new Geometry("Sphere", sp);
64         sphereGeom.updateModelBound();
65         sphereGeom.setMaterial(whitemat);
66         sphereGeom.setLocalTranslation(0, 1.3f, 0);
67         rootNode.attachChild(sphereGeom);
68
69         // 女性
70         Spatial girl = assetManager.loadModel("myAssets/Models/WalkingGirl/
           WalkingGirl.obj");
71         girl.rotate(0, (float)(Math.PI)*1.3f, 0);
72         girl.setLocalTranslation(1f, 0, 1f);
73         this.rootNode.attachChild(girl);
74
75         // 影の設定
76         girl.setShadowMode(ShadowMode.CastAndReceive);
77         floorGeom.setShadowMode(ShadowMode.CastAndReceive);
```

```
78          sphereGeom.setShadowMode(ShadowMode.CastAndReceive);
79
80          // 視点
81          cam.setLocation(new Vector3f(0, 1.7f, 6));
82      }
83
84      // 更新処理
85      public void simpleUpdate(float tpf) {
86          /* プロジェクタや物体、人などを動かしたいときはここにその処理を書きます */
87      }
88
89      // 終了処理
90      public void destroy() {
91          super.destroy();
92          System.exit(0);
93      }
94
95      // メイン
96      public static void main(String[] args){
97          SimpleApplication app = new ProjectorTest();
98          app.start();
99      }
100 }
```

グラフィックス側のコード

虹模様のカラーバーが動くアプレットです。これはあくまで一例ですので、写真や動画、幾何学模様のアニメーションなどいろいろ試してみてください。



カラーバーのアニメーション

ColorBarsPApplet.java

```
1   package cctest;
2
3   import processing.core.PApplet;
4
5   public class ColorBarsPApplet extends PApplet {
6
7       int BAR_NUM = 100;
8       float[] x = new float[BAR_NUM];
9       float[] xSpeed = new float[BAR_NUM];
10      float[] bWidth = new float[BAR_NUM];
11      int[] bColor = new int[BAR_NUM];
12
13      public void setup() {
14          size(200, 200);
15          frameRate(30);
16          smooth();
17          colorMode(HSB, 360, 100, 100, 100);
18          noStroke();
19          for (int i=0; i<BAR_NUM; i++) {
20              x[i] = random(width);
21              xSpeed[i] = random(-1, 1);
22              bWidth[i] = random(2, 200);
23              bColor[i] = color(random(360), random(90, 100), random(50, 100), 50);
24          }
```

```
25        }
26
27        public void draw() {
28            background(0);
29            for (int i=0; i<BAR_NUM; i++) {
30                fill(bColor[i]);
31                rect(x[i], 0, bWidth[i], height);
32                x[i] += xSpeed[i];
33                if (x[i] > width || x[i] < -bWidth[i]) {
34                    xSpeed[i] *= -1;
35                }
36            }
37        }
38    }
```

■ プロジェクタ・照明・物体の作り方と影の設定

いろいろな処理をやっていてさも複雑そうに見えますが、内容ごとに処理がきれいにまとまっているのでプログラムの構造は単純です。一行一行無理して理解しようとせず、「あーこの物体を入れたいときはこの数行を入れればいいのか」ぐらいの理解で大丈夫です。個別に見ていきましょう。

▪ プロジェクタ

プロジェクタは映像出力装置なので、ディスプレイと似たような扱いです。最初にProcessingのアプレットを作り、プロジェクタを作るときに紐づけます。プロジェクタを作ったら空間に追加し、必要に応じてプロジェクタの位置と向き（注視点）を設定します。

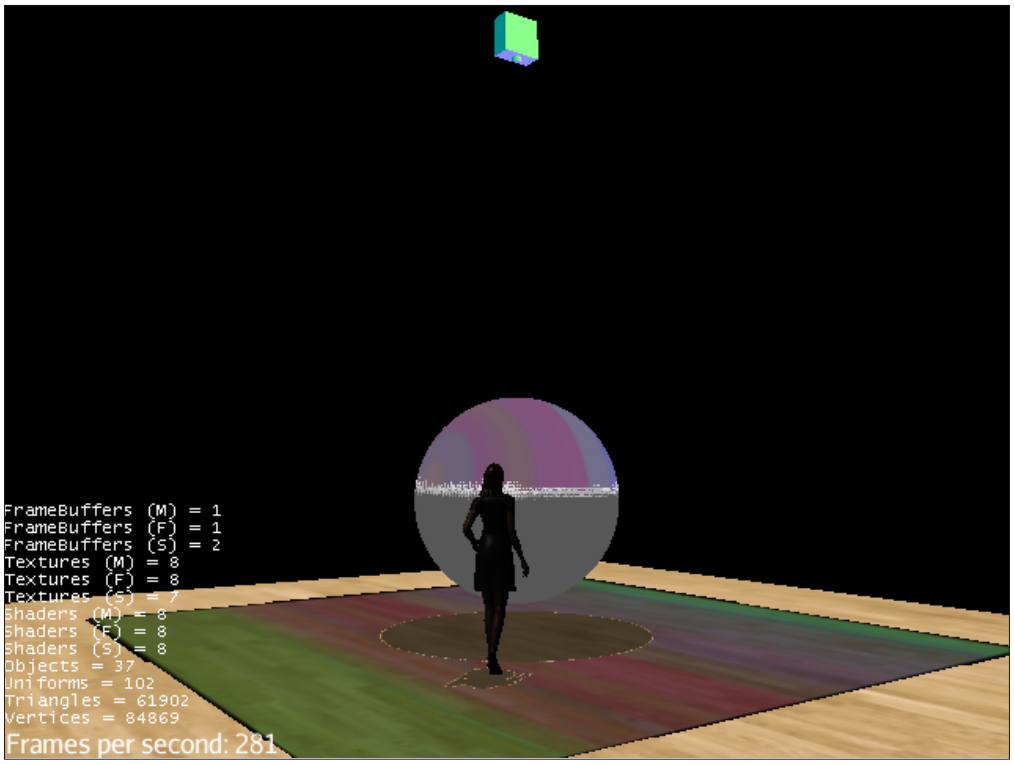
通常版プロジェクタ

```
1   // プロジェクタの設定（通常版）
2   PAppletProjectorNode projector = new PAppletProjectorNode("projector0", assetManager,
    applet, 200, 200, false);
3   rootNode.attachChild(projector);                              // 空間にプロジェクタを追加
4   rootNode.attachChild(projector.getFrustmMdel());              // プロジェクタの視野角を表示
5   projector.setLocalTranslation(new Vector3f(0,6,0));           // プロジェクタの位置
6   projector.lookAt(new Vector3f(0, 0, 0), Vector3f.UNIT_X); // プロジェクタの注視点
7   TextureProjectorRenderer ptr = new TextureProjectorRenderer(assetManager); // このプロ
    ジェクタ用のレンダリングの設定
8   ptr.getTextureProjectors().add(projector.getProjector());
9   viewPort.addProcessor(ptr);
```

プロジェクタには「PAppletProjectorNode」と「PAppletProjectorShadowNode」との２種類があります。「Shadow」と付くほうは、物体によって生じる影を考慮した投影が行われます。 Shadow版プロジェクタを使いたいときは、上記の通常版プロジェクタに関する処理をコメントアウトし、代わりにShadow版プロジェクタに関する以下の処理を有効にしてください。

Shadow版プロジェクタ

```
1   // プロジェクタの設定（Shadow版）
2   PAppletProjectorShadowNode projector = new PAppletProjectorShadowNode("Projector0",
    assetManager, viewPort, 1024, 1024, applet, 200, 200, false);
3   rootNode.attachChild(projector);                              // 空間にプロジェクタを追加
4   projector.setLocalTranslation(new Vector3f(0,6,0));           // プロジェクタの位置
5   projector.lookAt(new Vector3f(0, 0, 0), Vector3f.UNIT_X); // プロジェクタの注視点
```



Shadow版プロジェクタでの実行結果

■ 照明

照明にはいろいろな種類がありますが、ここで扱っているのは平行光源です。太陽光と同じだと考えてください。平行光源はDirectionalLightによって作成します。平行光源では、ある方向から空間全体に光が降り注ぐため、位置情報は持っていません。ここでは.setDirection()で方向を設定しています。空間への照明の追加はrootNode.addLight()という点に気を付けてください。

照明の作り方

```
1  // 照明
2  DirectionalLight dl = new DirectionalLight();
3  dl.setDirection(new Vector3f(-0.1f, -1f, -1).normalizeLocal());
4  rootNode.addLight(dl);
```

■ 物体

ここでは床面、球、女性という３種類のCG物体が登場します。それぞれ数行のコードで空間に追加することができます。

床のCGを追加する

```
1  // 床面
2  Material textureMat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
3  textureMat.setTexture("ColorMap", assetManager.loadTexture("myAssets/Textures/
   woodFloor.jpg"));
4  Box floor = new Box(Vector3f.ZERO, 5.0f, 0.01f, 5.0f);
5  Geometry floorGeom = new Geometry("Floor", floor);
6  floorGeom.setMaterial(textureMat);
7  rootNode.attachChild(floorGeom);
```

球のCGを追加する

```
// 球
Material whitemat = assetManager.loadMaterial("Common/Materials/WhiteColor.j3m");
Sphere sp = new Sphere(64, 64, 1.0f);
Geometry sphereGeom = new Geometry("Sphere", sp);
sphereGeom.updateModelBound();
sphereGeom.setMaterial(whitemat);
sphereGeom.setLocalTranslation(0, 1.3f, 0);
rootNode.attachChild(sphereGeom);
```

女性のCGを追加する

```
// 女性
Spatial girl = assetManager.loadModel("myAssets/Models/WalkingGirl/WalkingGirl.obj");
girl.rotate(0, (float)(Math.PI)*1.3f, 0);
girl.setLocalTranslation(1f, 0, 1f);
this.rootNode.attachChild(girl);
```

それぞれ作り方が微妙に異なっています。床と球では基礎形状（BoxおよびSphere）を作成してそこに材質を与える処理を行っているのに対し、女性のCGではあらかじめ材質情報を持っているOBJ形式のCGデータをロードしています。人や建物などの複雑な形状を持つものに関しては、なんらかのモデリングツール（例えばGoogleSketchUp）でOBJ形式のCGデータを作成し、それを読み込むのが手軽です。

CityCompilerのセットの中にはいくつか建物のCGデータが同梱されています。例えば、同梱されている東京駅のモデルを出すには以下のようなコードを書きます。

東京駅のCGを追加する

```
Spatial model = assetManager.loadModel("myAssets/Models/TokyoStation/TokyoStation.obj");
rootNode.attachChild(model);
```

■ 影の設定

GeometryやSpatialで作られた物体に対して、それぞれどのように影が映るかを設定する必要があります。これには .setShadowMode() というメソッドを使います。通常は引数に ShadowMode.CastAndReceive（影を出す＆受ける）を指定してください。

物体ごとに影の映り方を決める

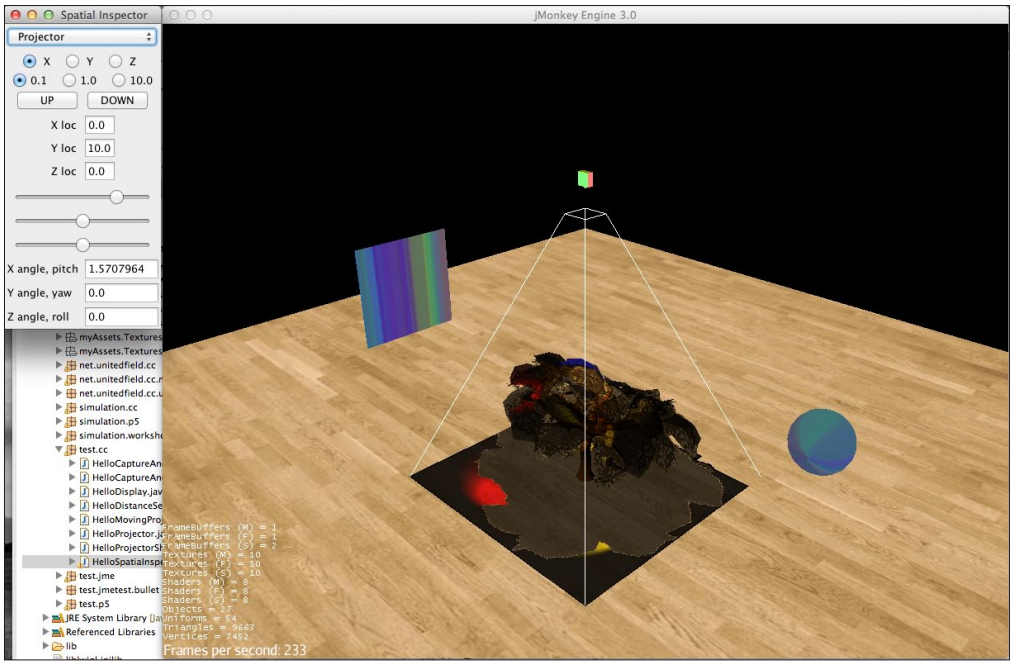
```
// 影の設定
girl.setShadowMode(ShadowMode.CastAndReceive);
floorGeom.setShadowMode(ShadowMode.CastAndReceive);
sphereGeom.setShadowMode(ShadowMode.CastAndReceive);
```

GUIからディスプレイやカメラを移動させよう

Let's move camera or display with GUI

ディスプレイやプロジェクタ、カメラなどをバーチャル空間に置いてみた後に、位置や向きをいろいろと変えてみたくなる場合があります。GUIウィンドウを使ってそれぞれの物体を動かすことができるSpatialUtilクラスが用意されています。

次のサンプルは、その上からプロジェクターが木の床の上に置かれた樹に静止画を投影し、その横にアプレットを表示する平面のディスプレイと球体のディスプレイが置かれている、というサンプルです。



HelloSpatialInspectorの実行画面

HelloSpatialInspecor.java

```
1   package test.cc;
2
3   import net.unitedfield.cc.PAppletDisplayGeometry;
4   import net.unitedfield.cc.PAppletProjectorShadowNode;
5   import net.unitedfield.cc.util.SpatialInspector;
6   import processing.core.PApplet;
7   import test.p5.ColorBarsPApplet;
8
9   import com.jme3.app.SimpleApplication;
10  import com.jme3.light.AmbientLight;
11  import com.jme3.light.DirectionalLight;
12  import com.jme3.material.Material;
13  import com.jme3.math.ColorRGBA;
14  import com.jme3.math.Vector3f;
15  import com.jme3.renderer.queue.RenderQueue.Bucket;
16  import com.jme3.renderer.queue.RenderQueue.ShadowMode;
17  import com.jme3.scene.Geometry;
18  import com.jme3.scene.Mesh;
19  import com.jme3.scene.Spatial;
20  import com.jme3.scene.shape.Box;
21  import com.jme3.scene.shape.Sphere;
22  import com.jme3.texture.Texture2D;
23  import com.jme3.util.SkyFactory;
24
```

```
25  public class HelloSpatialInspector extends SimpleApplication {
26
27      @Override
28      public void simpleInitApp() {
29          //cam
30          cam.setLocation(Vector3f.UNIT_XYZ.mult(15.0f));   // camera moves to 15, 15,
            15
31          cam.lookAt(new Vector3f(0,5,0), Vector3f.UNIT_Y); // and looks at 0,0,0.
32          flyCam.setMoveSpeed(10);
33          flyCam.setDragToRotate(true);
34
35          // light
36          DirectionalLight dl = new DirectionalLight();
37          dl.setDirection(new Vector3f(-0.1f, -1f, -1).normalizeLocal());
38          dl.setColor(ColorRGBA.Orange);
39          rootNode.addLight(dl);
40          // floor
41          Material textureMat = new Material(assetManager, "Common/MatDefs/Misc/
            Unshaded.j3md");
42          textureMat.setTexture("ColorMap", assetManager.loadTexture("myAssets/Textures/
            woodFloor.jpg"));
43          Box floor = new Box(Vector3f.ZERO, 20.0f, 0.01f, 20.0f);
44          Geometry floorGeom = new Geometry("Floor", floor);
45          floorGeom.setMaterial(textureMat);
46          rootNode.attachChild(floorGeom);
47          // tree
48          Spatial tree = assetManager.loadModel("Models/Tree/Tree.mesh.j3o");
49          tree.setQueueBucket(Bucket.Transparent);
50          rootNode.attachChild(tree);
51
52          // ProjectorShadowNode
53          PAppletProjectorShadowNode ppg = new PAppletProjectorShadowNode("Projector"
            ,assetManager,viewPort, 200,200, (Texture2D)assetManager.loadTexture(
            "Interface/Logo/Monkey.png"));
54          rootNode.attachChild(ppg);
55          rootNode.attachChild(ppg.getFrustmModel());
56          ppg.setLocalTranslation(new Vector3f(0,10,0));
57          ppg.lookAt(new Vector3f(0, 0, 0), Vector3f.UNIT_Y);
58          //projector is a kind of Shadow, and following processes are necessary for
            Shadow Rendering.
59          floorGeom.setShadowMode(ShadowMode.Receive);
60          tree.setShadowMode(ShadowMode.CastAndReceive); // tree makes and receives
            shadow
61
62          // PApplet and PAppletDisplayGeometry
63          // flat display
64          PApplet applet0 = new ColorBarsPApplet();
65          PAppletDisplayGeometry flatDisplay = new PAppletDisplayGeometry("FlatDisplay",
            assetManager, 4, 4, applet0, 200, 200, false);
66          rootNode.attachChild(flatDisplay);
67          flatDisplay.setLocalTranslation(-10, 4, 0);
68          flatDisplay.rotate(0, (float)Math.PI/2, 0);
69          // sphere display
70          PApplet applet1 = new ColorBarsPApplet();
71          Mesh sphere = new Sphere(20, 20, 0.8f);
72          PAppletDisplayGeometry sphereDisplay = new PAppletDisplayGeometry(
            "SphereDisplay", sphere, assetManager, applet1, 200, 200, false);
73          rootNode.attachChild(sphereDisplay);
74          sphereDisplay.setLocalTranslation(8, 5, 0);
75
76          /*
77           * Get a instance of SpatialInspector, and add it to each object as control.
78           */
79          SpatialInspector spatialInspector = SpatialInspector.getInstance();
80          ppg.addControl(spatialInspector);
81          tree.addControl(spatialInspector);
82          flatDisplay.addControl(spatialInspector);
83          sphereDisplay.addControl(spatialInspector);
84          spatialInspector.show();
85          this.setPauseOnLostFocus(false);
86      }
87
88      public void destroy() {
89          super.destroy();
90          System.exit(0); // we should terminate the thread of PApplet.
91      }
92
93      public static void main(String[] args) {
94          SimpleApplication app = new HelloSpatiaInspector();
95          app.start();
96      }
97  }
```

No. 1

SpatialInspectorにオブジェクトを追加しよう

まずはそれぞれのインスタンスをnewして配置します。simpleInitApp()の後の所でそれぞれのオブジェクトをGUIから動かせるようにSpatialInspectorのインスタンスを取得します。このGUIはひとつのシミュレーションにひとつあれば十分なので、Singletonパターンで実装しています。

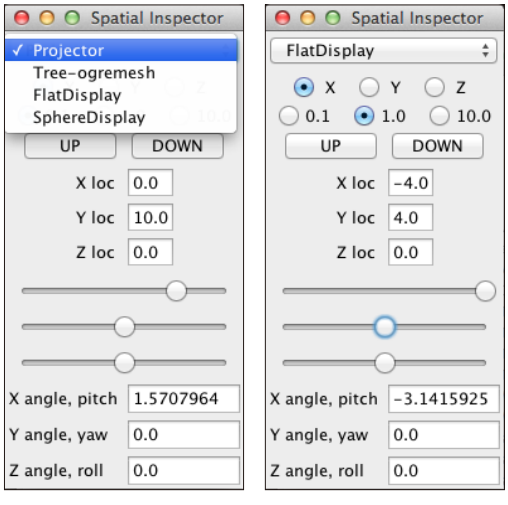
このサンプルでは生成したオブジェクトのうち、樹とプロジェクタ、平面ディスプレイと球体ディスプレイの位置や向きをGUIから変えられるよう、それぞれのオブジェクトに対してaddControl(spatialInspector);とメソッドを呼んでいます。このaddControlというメソッドですが、もともとはゲームの中のサブキャラや敵キャラを決まった動きをさせるために準備されているクラス群がcom.jme3.scene.control.*にあります。メインのオブジェクトを動かしたい場合にはsimpleUpdate()の中等に場所や向きを変えるようにしますが、このクラス群はそこにサブキャラや敵キャラのコードが沢山書かれないようにするためのものです（詳しくはこちら: http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:custom_controls）。

そしてspatialInspector.show();とするとインスペクタが表示されます。気を付けておくべきことは、this.setPauseOnLostFocus(false);もあわせて呼んでおくことです。これによって、インスペクタを操作している間にjMEがポーズされなくなります。そうしないと、ボタンを押して移動したのに反映されない…でもウィンドウをクリックしたらすごく動いていた！ ということが起きます。

No. 2

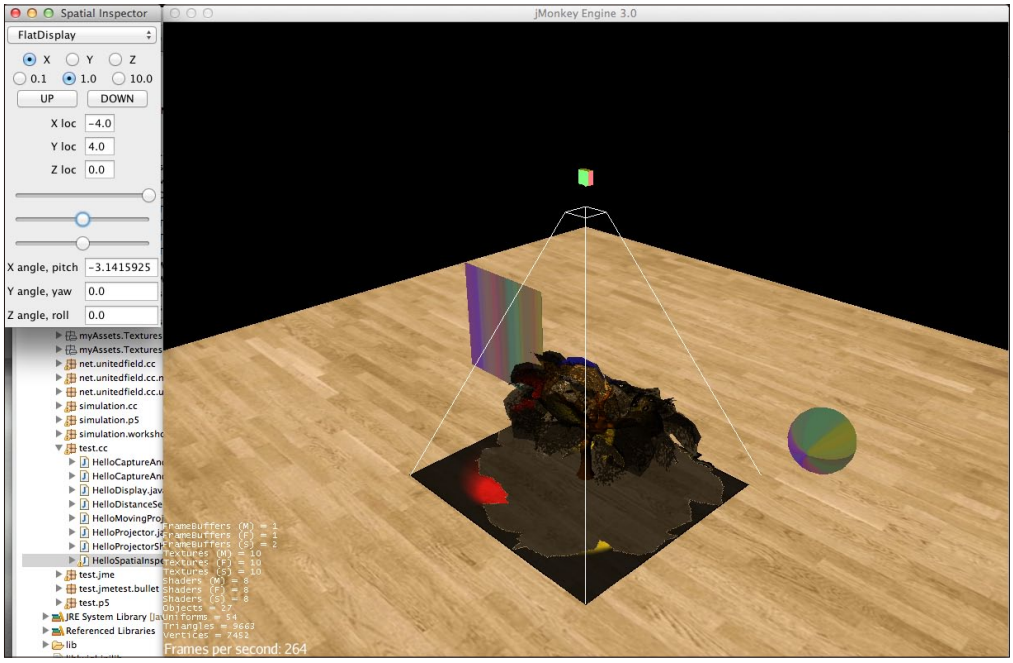
GUIからオブジェクトの位置や向きを変えてみよう

addControlを呼び出されたオブジェクトはGUIのコンボボックスに登録されているので、位置／向きを変えたいオブジェクトをその中から選びます。

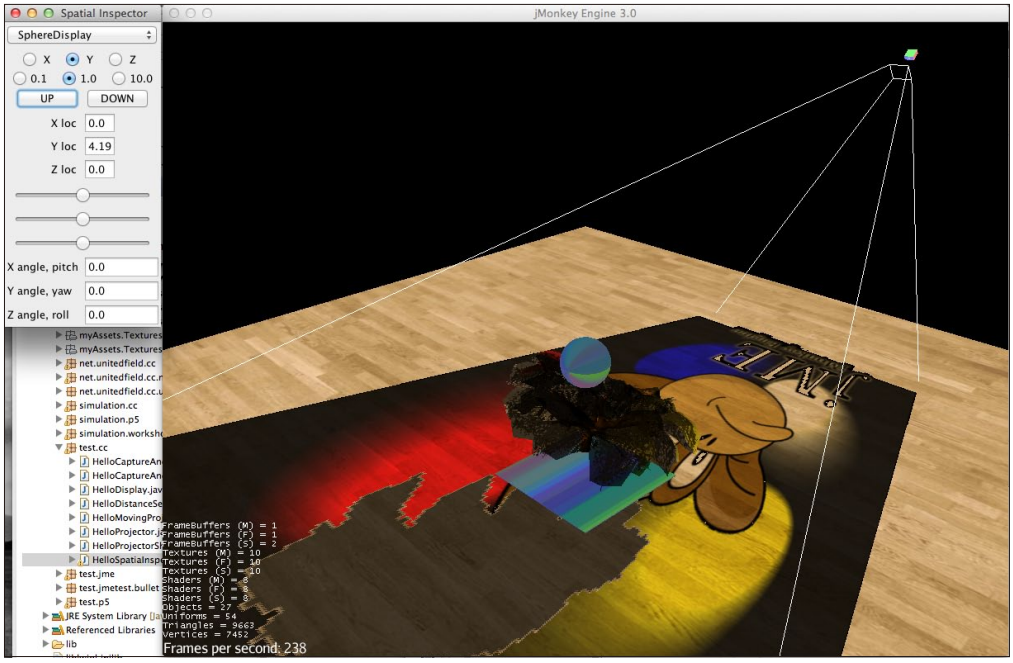


位置／向きを変えたいオブジェクトを選び、
そのオブジェクトの位置や向きを変える

a上半分が位置を変更するためのラジオボタンおよびボタン、今の座標を示すテキストフィールドです。下半分が向きを変えるためのスライダーと今の角度を示すテキストフィールドです。位置を変える際には、ラジオボタンでX/Y/Z座標のどれを変えるか・ボタンを押す度に移動する量（0.1mか1mか10mか）を選び、UP/DOWNのボタンで座標の値を変更します。向きを変える場合には、上からピッチ:X／ヨー:Y／ロール:Zをスライダーで変更します。位置と向きの現在の値がテキストフィールドに表示されるので、位置と向きが決まったらソースコードにコピペしておくと良いでしょう。



FlatDisplayを選び、角度を変えてからX座標を1.0ずつ増やしているところ



ディスプレイ2つとプロジェクタの位置と向きも変えてみました。
球体ディスプレイと平面ディスプレイからは影が落ちない設定になっているので、木の影だけが床に落ちています。

距離センサーを使ってみよう

Let's use distance sensor

Processingはマウスやキーボード、ディスプレイといったGUIの入出力機器を取り扱うだけでなく、ProcessingとArduinoを組み合わせることでセンサやモータといった入出力機器を使ったシステムを作ることができます。

具体的な使い方は

yoppa.org

ArduinoとProcessingの連携1：センサの情報を視覚化する
(http://yoppa.org/bma10/1289.html)

ArduinoとProcessingの連携2：大きな値を送信する、データの流れを視覚化する
(http://yoppa.org/bma10/1334.html)

ArduinoとProcessingの連携3：「植物シンセ」を作る
(http://yoppa.org/bma10/1365.html)

橋本直 ARプログラミング Processingでつくる拡張現実感のレシピ オーム社(http://www.amazon.co.jp/dp/4274211746/) などがとても参考になると思います。

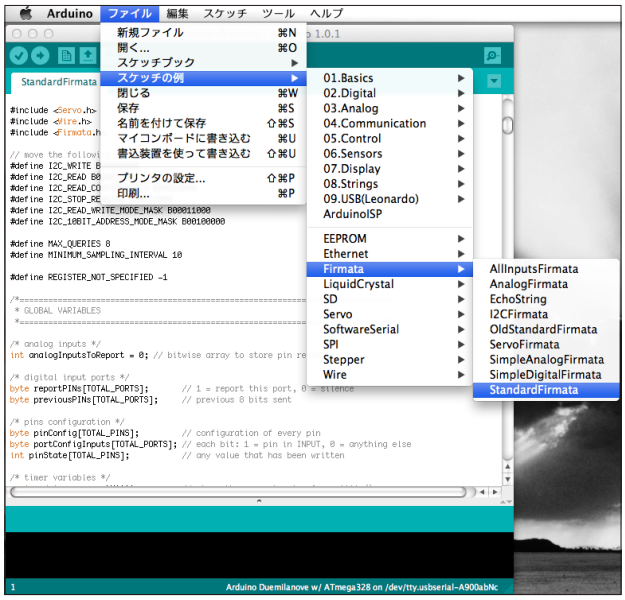
ProcessingとArduinoを組み合わせたシステムには、PC1台＋Arduino1台＋入出力機器（センサやモータ）というシステムが多いのですが、CityCompilerを使うとそうしたシステムを何組も使って空間的に配置したシステムへと進化させやすくなります。ここではProcessingにArduino＋距離センサを組み合わせたシンプルな例を紹介し、それを仮想空間に表示されているProcessingの入力に仮想距離センサを組み合わせた例にしていく過程を紹介します。

N°1

■ ProcessingとArduinoをつなぐFirmata

USB（シリアルポート）につながっているArduinoのセンサからの値をPC側のProcessingに送ったり、PC側のProcessingからArduinoにつながっているモータを動かすためには、シリアルポート通信でデータをやりとりする必要があります。こうしたやり取りは何時誰がやっても同じ処理を書くことになるので、Firmataというライブラリが準備されています。

Arduinoにはあらかじめ準備されたこのやり取りのためのプログラムが最初からExamplesに入っています。Arduino IDEでFile->Examples->Firmata->Standard Firmataを選択して、それをUSBにつないだArduinoにUploadしておきます。



Aduino IDEでFile->Examples->Firmata->Standard Firmata

FirmataのProcessingのライブラリはArduinoのサイトのArduino and Processing (http://playground.arduino.cc/interfacing/processing)からダウンロードできますが、Processing2.0bではエラーが出てしまいます。githubに置いてあるCityCompilerでは、このArduinoのサイトで配布されているArduino.javaをProcessing2.0bで動作するように修正したものをcc.arduinoパッケージに同梱してあります(2013/01現在)。Arduinoのサイトで対応版が公開されたら、そちらを使うことをおススメします。

Firmataを使うことでnew Arduino();としてインスタンスを生成することができます。そのインスタンスにdigitalRead(), digitalWrite(), analogRead(),analogWrite()といったメソッドを呼び出してArduinoとデータをやり取りすることができます。次のコードは、マウスをクリックしたらArduinoのボードにあるLEDをオンするという一番シンプルなProcessingのサンプルです。

FirmataTest.pde

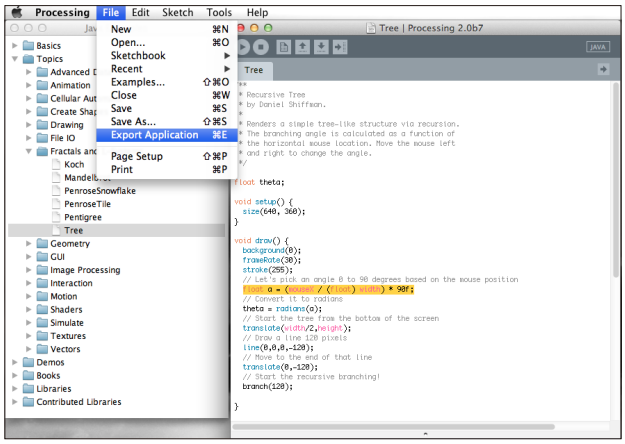
```
1   import cc.arduino.*;
2   import processing.serial.*;
3
4   Arduino arduino;
5   int pin = 13;
6
7   void setup(){
8       println(Arduino.list());
9       arduino = new Arduino(this, Arduino.list()[0], 57600);
10      arduino.pinMode(pin, Arduino.OUTPUT);
11  }
12
13  void draw(){
14      if(mousePressed) {
15          arduino.digitalWrite(pin, Arduino.HIGH);
16      }else{
17          arduino.digitalWrite(pin, Arduino.LOW);
18      }
19  }
```

これをエクポートしたものをtest.p5パッケージのFirmataPApplet.javaとしてありますので、ArduinoやFirmataの動作確認に使って下さい。

N°. 2

ProcessingからArduinoにつないだ距離センサを使う

ここからは、ProcessingのExamples->Topics->Fractals and L-Systemsの中のTreeというサンプルを使っていきます。このPAppletはマウスのX座標を使って、関数を再帰的に呼び出して木のカタチを描きます。Javaにエクスポートして Eclipseに取り込んで、マウスのX座標でカタチを変えている部分をArduinoにつないだ距離センサ（Sharp 2Y0A21）からの値で変えるように変更します。



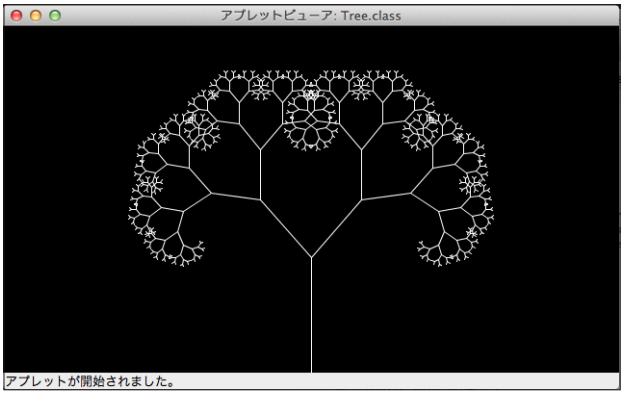
ProcessingのTree.pdeをエクスポート

距離センサをオブジェクトとしてnewできるようにDistanceSensorFirmataというクラスをnet.unitedfield.ccパッケージの中に用意しています。test.p5パッケージにTreeWithDistanceSensorFirmata.javaというコードがありますが、これはTree.pdeをJavaに書き出したものをDistanceSensorFirmataを使うように変更したものです。

TreeWithDistanceSensorFirmata.java

```
1   package test.p5;
2
3   import net.unitedfield.cc.DistanceSensorFirmata;
4   import processing.core.PApplet;
5
6   public class TreeWithDistanceSensorFirmata extends PApplet {
7
8   /**
9    * Recursive Tree
10   * by Daniel Shiffman.
11   *
12   * Renders a simple tree-like structure via recursion.
13   * The branching angle is calculated as a function of
14   * the horizontal mouse location. Move the mouse left
15   * and right to change the angle.
16   */
17
18  float theta;
19  DistanceSensorFirmata distanceSensor; // instance of DistanceSensorFirmata
```

```
20
21      public void setup() {
22        size(640, 360);
23        distanceSensor = new DistanceSensorFirmata(0);
24        distanceSensor.setup();
25      }
26
27      public void draw() {
28        background(0);
29        frameRate(30);
30        stroke(255);
31
32        // Let's pick an angle 0 to 90 degrees based on the mouse position or
          DistanceSensorFirmata
33        //float a = (mouseX / (float) width) * 90f; // original code in Tree.java
34        /* using DistanceSensorFirmata */
35        float a = (distanceSensor.getDistance()/distanceSensor.getSenseMax()) *90f;
36
37        // Convert it to radians
38        theta = radians(a);
39        // Start the tree from the bottom of the screen
40        translate(width/2,height);
41        // Draw a line 120 pixels
42        line(0,0,0,-120);
43        // Move to the end of that line
44        translate(0,-120);
45        // Start the recursive branching!
46        branch(120);
47      }
48
49      public void branch(float h) {} // same as in Tree.java
50  }
```



TreeWithDistanceSensorFirmataの実行画面

元のTree.pdeをエクポートしたTree.javaでは
float a = (mouseX / (float) width) * 90f;
として木のパラメータを変更していたところを
float a = (distanceSensor.getDistance()/distanceSensor.getSenseMax())
*90f;
と変更しました。 またPAppletのsetup()の中でDistanceSensorFirmataのインスタンスをnewした後にsetup()を呼び出しています。
distanceSensor = new DistanceSensorFirmata(0);
distanceSensor.setup();
これはDistanceSensorFirmataもPAppletのサブクラスであるためです。Firmataライブラリを使ってArduinoクラスのインスタンスをnewする際にはその引数としてPAppletを渡しますが、そのためにDistanceSensorFirmataはPAppletのサブクラスとなっています。
コンストラクタに渡す引数は距離センサが接続されているArduinoのピン番号です。現在

は1つのArduinoにひとつの距離センサが繋がっている状態を想定して実装しています。複数の距離センサが1つのArduinoに接続している場合や複数の異なるセンサが1つのArduinoにつながっている場合はまた別の実装をする必要があるため、今後対応してゆく予定です。

N° 3

仮想空間に表示したProcessingからArduinoにつないだ距離センサを使う

このPAppletをnewして仮想ディスプレイに表示すれば、距離センサと連動したコンテンツを表示するディスプレイのサンプルが出来上がります。
CityCompilerでは仮想空間と模型空間を行ったり来たりしながらプロトタイピングを進めることがあります。そうした進め方がやり易いよう、jMEの仮想空間の中で動作する仮想距離センサもDistanceSensorNodeも用意しました。リアルな距離センサDistanceSensorFirmataと仮想の距離センサDistanceSensorNodeで距離を計る時には同じ名前のメソッドを呼び出せるよう実装しています。

Processingでカメラを使う時はnew Capture();としますが、このCaptureもクラスではなくインタフェースです。そのためにハードウェアのカメラにアクセスするライブラリとしてQuickTimeを使ったりGStreamerを使ったりとVideo Libraryを切り替えられる柔軟性がProcessingには備わっています。この柔軟性を活用して、CityCompilerにおける仮想カメラもこのCaptureインタフェースを実装したクラスとして作り、リアルカメラと仮想カメラの切り替えを実現しています。
リアル距離センサであるDistanseSensorFirmataと仮想距離センサであるDistanseSensorNodeはどちらもDistanseSensorというインタフェースをimplementsしているので、同じような仕組みでリアルセンサと仮想センサの切替えができます。

ここではさらに、仮想距離センサにも対応できるようPAppletを少し修正してみしょう。カメラを使ったPAppletと同じようにまず boolean realDeployment; という変数を定義します。これがtrueならArduino+Firmata経由でリアルな距離センサを使うようにして、これがfalseなら仮想距離センサを使うことにします。

TreeWithDistanceSensor.java

```
1   package test.p5;
2
3   import net.unitedfield.cc.DistanceSensor;
4   import net.unitedfield.cc.DistanceSensorFirmata;
5   import processing.core.PApplet;
6
7   public class TreeWithDistanceSensor extends PApplet {
8
9   float theta;
10  DistanceSensor  distanceSensor = null;
11  boolean     realDeployment = false;
```

```
12
13      public void setup() {
14          size(640, 360);
15          if(realDeployment == true){
16              distanceSensor = new DistanceSensorFirmata(0);
17              ((DistanceSensorFirmata)distanceSensor).setup();
18          }
19      }
20
21      public  void setDistanceSensor(DistanceSensor sensor){
22              this.distanceSensor = sensor;
23      }
24
25      public void draw() {
26          background(0);
27          frameRate(30);
28          stroke(255);
29
30          // Let's pick an angle 0 to 90 degrees based on the mouse position or
            DistanceSensorFirmata
31          //float a = (mouseX / (float) width) * 90f;
32          float a =0;
33          if(distanceSensor != null)
34              a = (distanceSensor.getDistance()/distanceSensor.getSenseMax()) *90f;
35
36          theta = radians(a);
37          translate(width/2,height);
38          line(0,0,0,-120);
39          translate(0,-120);
40          branch(120);
41      }
42
43      public void branch(float h) {} // same as in Tree.java
44  }
```

DistanceSensorFirmataのインスタンスとして定義していたインスタンス変数はDistanceSensorをimplmentしたオブジェクトとすることで、リアル距離センサと仮想距離センサのどちらも使えるようにしておきます。
リアル距離センサを使いたい時は、realDeploymentをtrueにしておき、setup()の中でDistanceSensorFirmataのインスタンスをnewします。DistanceSensorFirmataのsetup()も呼び出さないといけないので、キャストをして呼び出します。

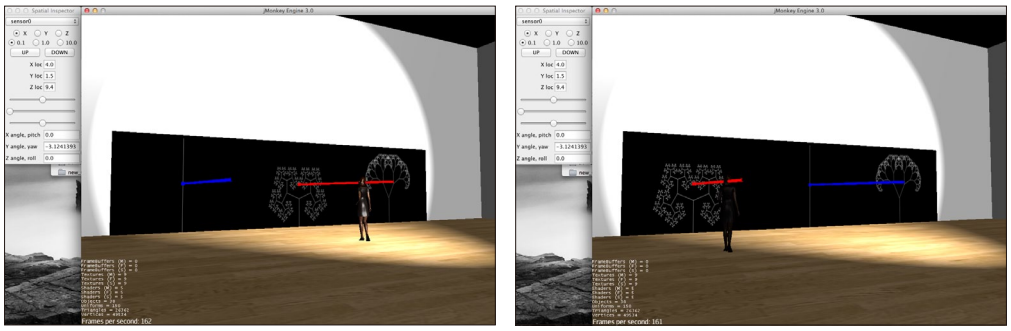
仮想距離センサを使いたい時は、jMEのコードの中でこのアプレットをnewしてから、仮想距離センサのオブジェクトをこのアプレットに接続することになります。なのでそのためのメソッドとして、void setDistanceSensor(DistanceSensor sensor)を追加しました。アプレットはnewされたけど距離センサがまだ渡されていない状態でもNullPointerExceptionが起きないよう、変数distanceSensorを定義するところではnullにしておき、nullでなければdistanceSensor.getDistance()を実行するようにします。これでPAppletがリアル距離センサと仮想距離センサの両方を使えるようになりました。

リアルカメラと仮想カメラを使う場合もPAppletでrealDeploymentという変数を定義した上でsetCapture(Capture capture)というメソッドを準備してありますが、ここでの作り方と同じになっています。

N° 4

仮想空間に表示したProcessingから仮想距離センサを使う

次はこのアプレットに仮想距離センサを接続し、アプレットの表示を仮想ディスプレイにしてみます。今回は複数のセンサとディスプレイを表示するサンプルにしてみましょう。センサとディスプレイの組を3セット用意することにして、特にそのうちの距離センサ1つはリアルセンサにしてみます。



DistanceDisplaysSimulationの実行画面。左と真ん中の距離センサは仮想距離センサ、
右の距離センサはリアル距離センサ（Arduinoの先に距離センサがつながっているので、
リアル距離センサのカタチはjMEの中にはない）。

DistanceDisplaysSimulation.java

```
1   package simulation.cc;
2
3   public class DistanceDisplaysSimulation extends SimpleApplication {
4       Node senseTarget;
5       Spatial girl;
6
7       public void simpleInitApp() {
8           cam.setLocation(new Vector3f(-1f, 1.5f, -3f));
9           cam.lookAt(new Vector3f(0, 1.8f, 5f), Vector3f.UNIT_Y);
10          flyCam.setDragToRotate(true);
11
12          senseTarget = new Node();
13          setupEnvironment();
14          setupDistanceDisplays();
15          setupGirl();
16      }
17
18      private void setupDistanceDisplays() {
19          DistanceSensor      sensors[]   = new DistanceSensor[3];
20          TreeWithDistanceSensor  applets[]   = new TreeWithDistanceSensor[3];
21          PAppletDisplayGeometry  displays[]  = new PAppletDisplayGeometry[3];
22
23          SpatialInspector spatialInspector = SpatialInspector.getInstance();
24          for(int i=0; i<3 i++){
25              if(i<2){
26                  DistanceSensorNode sensorNodeV = new DistanceSensorNode("sensor"+i,
                    5f, assetManager, senseTarget);
27                  sensorNodeV.setLocalTranslation(new Vector3f(4-4*i, 1.5f, 9.4f));
28                  sensorNodeV.rotate(0, FastMath.PI, 0);
29                  rootNode.attachChild(sensorNodeV);
30                  sensorNodeV.addControl(spatialInspector);
31                  sensors[i] = sensorNodeV;
32              }else{
33                  DistanceSensorFirmata sensorNodeR = new DistanceSensorFirmata(0);
34                  sensorNodeR.setup();
35                  sensors[i] = sensorNodeR;
36              }
37              applets[i] = new TreeWithDistanceSensor();
38              applets[i].setDistanceSensor(sensors[i]);
39              displays[i] = new PAppletDisplayGeometry("display"+i,assetManager,4,3,app
                lets[i],640,360,false);
40              displays[i].setLocalTranslation(new Vector3f(4-4*i, 1.5f, 9.4f));
41              rootNode.attachChild(displays[i]);
42              displays[i].addControl(spatialInspector);
43          }
44          spatialInspector.show();
45          this.setPauseOnLostFocus(false);
46      }
47
```

```
48        private void setupEnvironment() {
49        (省略)
50        }
51
52        private void setupGirl(){
53        (省略)
54        }
55
56        // イベントリスナー
57        private AnalogListener analogListener = new AnalogListener() {
58        (省略)
59        };
60
61        public void destroy() {
62        (省略)
63        }
64
65        public static void main(String[] args) {
66            SimpleApplication app = new DistanceDisplaysSimulation();
67            app.setPauseOnLostFocus(false);
68            app.start();
69        }
70    }
```

setupDistanceDisplays();の中で距離センサは２つを仮想距離センサ、１つをリアル距離センサをnewしています。どちらもDistanceSensorインタフェースをimplementしているので、DistanceSensor sensors[];という配列で３つのセンサを保持します。
またPAppletに距離センサを渡す部分では、距離センサが仮想であるかリアルであるかを気にする必要もありません。setup()が呼ばれるタイミングよりも後にsetDistanceSensor()で距離センサをPAppletに渡せるよう、TreeWithDistanceSensor.javaではrealDeployment = false;としてあります。
こうしてサンプルを作ってみると、距離センサは線的な広がりしかないようなセンサではなく、面的な広がりがあるレンジセンサのようなものが欲しくなってきます。その時には、このサンプルやカメラ（仮想／リアル）を参考にしながら仮想レンジセンサとリアルレンジセンサの両方を実装してみて下さい。

細かいノウハウ集 Tips

JMEの設定画面を非表示にするには？

実行時に表示されるJMEの設定画面を非表示にするには、app.setShowSettings(false)を使います。 また、実行画面のサイズを任意に設定したい場合は、AppSettings型のデータを作成した後、app.setSettings() で設定します。

設定画面の非表示と画面サイズの設定

```
public static void main(String[] args) {
    SimpleApplication app = new DisplayTest();

    // 設定画面を非表示にする
    app.setShowSettings(false);

    // 画面サイズの設定
    AppSettings s = new AppSettings(true);
    s.setWidth(1024);
    s.setHeight(768);
    app.setSettings(s);

    app.start();
}
```

ステータス表示をOFFにするには？

左下に表示されるステータス文字列の表示をなくしたい場合は、app.setDisplayStatView(false)とします。 また、FPSの表示をOFFにするにはapp.setDisplayFps(false)とします。

ステータス表示とFPS表示のOFF

```
public static void main(String[] args) {
    SimpleApplication app = new DisplayTest();

    app.setDisplayStatView(false);  // ステータスの表示を OFF
    app.setDisplayFps(false);       // FPS の表示を OFF

    app.start();
}
```

■ 画面をドラッグしたときだけ視点が動くようにするには？

通常simpleApplicationでは画面上でマウスカーソルを動かすだけで視点が動きますが、これをドラッグしたときだけ動くようにするにはsimpleInitApp()の中で flyCam.setDragToRotate(true); とやります。

ドラッグしたときだけ視点が動くようにする

```
1  public void simpleInitApp() {
2      /*
3       * 他の処理
4       */
5      flyCam.setDragToRotate(true);
6  }
```

■ アプレットへのマウス入力等がすぐに仮想空間に反映されるようにするには？

マウスを使った入力をするためにアプレットのウィンドウにアクティブにすると、何も設定せずにいるとJMEのウィンドウでは描画が止まってしまいます。アプレットとJMEの両方が常に描画されるようにするには、JMEのSimpleApplicationのメソッドsetPauseOnLostFocus(false);とやります。

マウスが別のウィンドウに移ってもJMEの描画を止めないようにする

```
1  public static void main(String[] args) {
2      SimpleApplication app = new DisplayTest();
3
4      app.setDisplayStatView(false);  // ステータスの表示を OFF
5      app.setDisplayFps(false);       // FPS の表示を OFF
6      app.setPauseOnLostFocus(false); // JME のウィンドウが一番前でなくても描画を止めない
7      app.start();
8  }
```

とやってもいいですし、

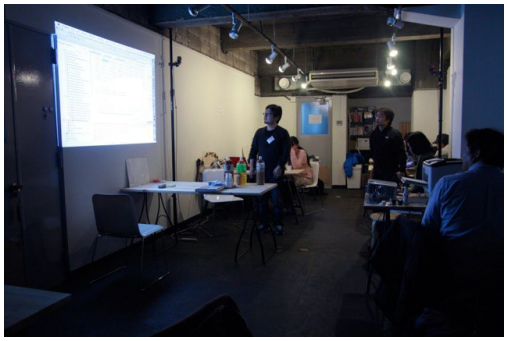
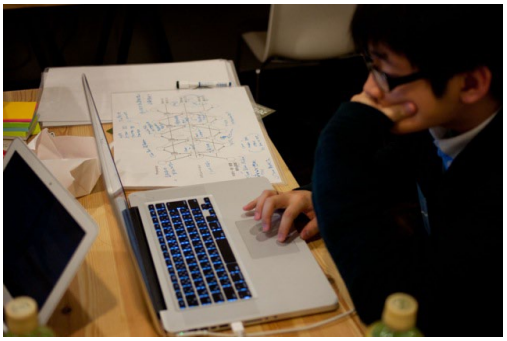
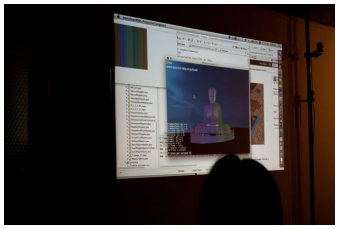
マウスが別のウィンドウに移ってもJMEの描画を止めないようにする

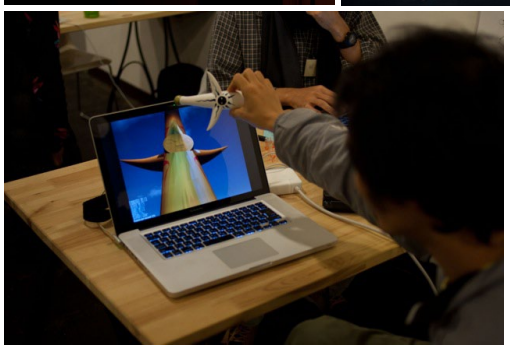
```
1  public void simpleInitApp() {
2      /* 他の処理 */
3      PApplet applet = new DynamicParticlesRetained(); //マウスを使うアプレットを使う
4      this.setPauseOnLostFocus(false); // JME のウィンドウが一番前でなくても描画を止めない
5      /* 他の処理 */
6
7  }
```

とやってもいいです。

■ ワークショップ「空間をプログラミングしよう！」東京都中央区日本橋本町にて開催。

第1回　2012年11月24日
第2回　2012年11月25日
第3回　2013年01月19日


謝辞

CityCompilerの開発およびこのドキュメントの作成は科学技術振興機構における戦略的創造研究推進事業さきがけ研究領域「知の創成と情報社会」における研究課題「空間的な情報システムの設計支援システム」の一環として行われたものです。ここに記して感謝の意を表します。

CityCompilerで
空間をプログラミングしよう!

2013年1月19日

企画・執筆: 中西泰人 (http://unitedfield.net/)
田所　淳 (http://yoppa.org)
橋本　直 (http://kougaku-navi.net)
デザイン: 荒川慎一 (http://d-knots.com/)