Processingプログラミング

会津大学

復興知2022年度

Processingの基礎

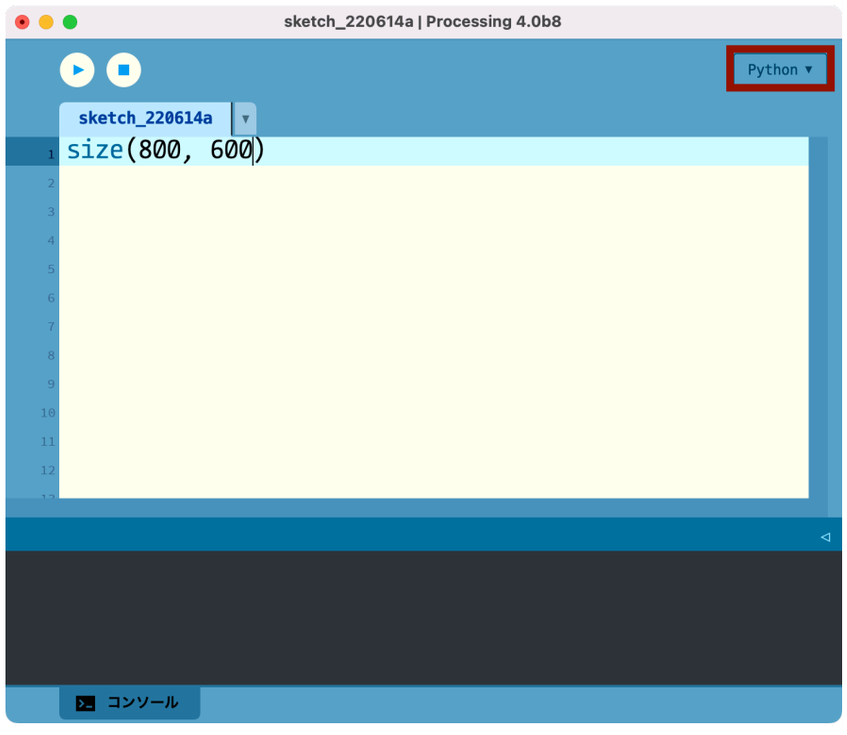
Processing

- マサチューセッツ工科大学メディアラボが開発した,
  電子アートやビジュアルデザインを扱いやすい統合開発環境
  - 可視化が簡単なため，初心者でも学びやすい
  - プログラミング言語であるJavaやPythonなどで使用

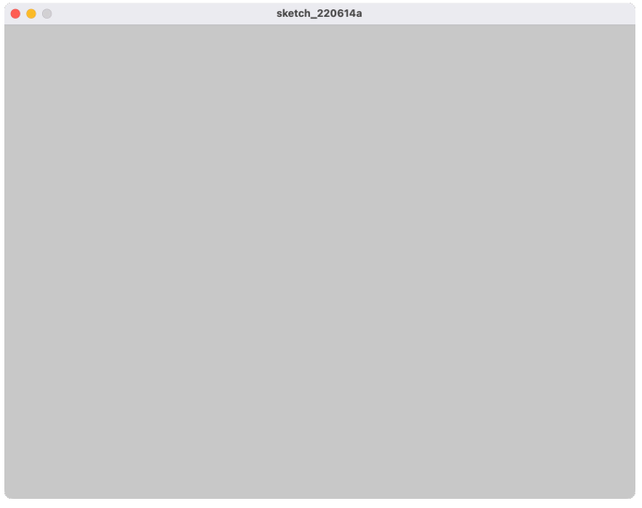
- この講習で行うこと
  - アニメーションの基礎
  - 物理シミュレーション，ゲームプログラミング

Processing

- 基本的にスケッチブックとよばれる統合開発環境に，プログラムを記述して実行する
  - 右上の部分が **Python** であれば，Pythonで実行できる
  - ▶ をクリックすることで，プログラムを実行できる



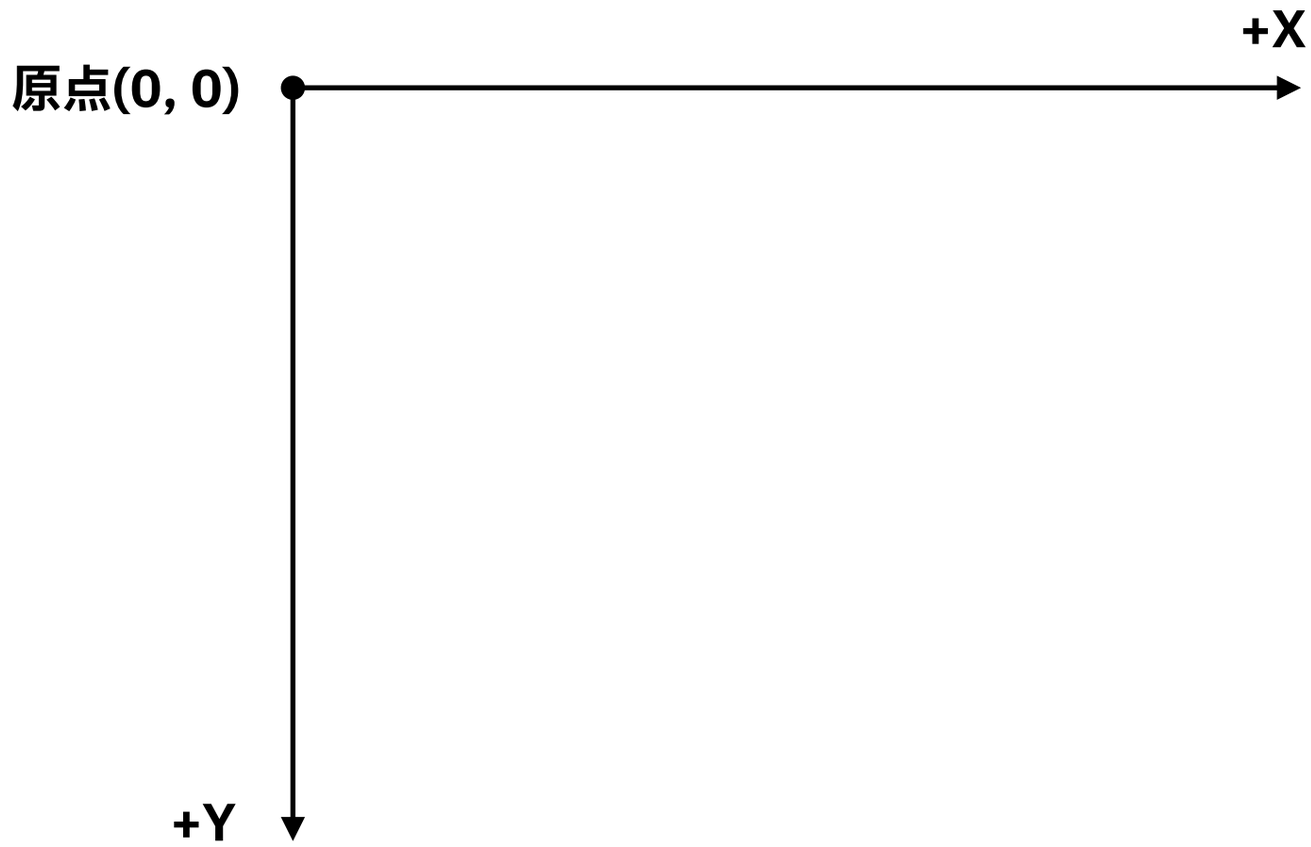
プログラムを実行



幅800，高さ600のウィンドウが表示

ウィンドウを作成

- **`size(w, h)`**: 幅w，高さhのウィンドウを作成する
  - w : 整数値，幅(ピクセル)
  - h : 整数値，高さ(ピクセル)
- Processingの座標系
  - **ウィンドウの左上: 原点**
  - 横方向: X軸方向
    - 右向きが正方向
  - 縦方向: Y軸方向
    - 下向きが正方向



Processingでの描画方法

❖ 関数を実行することで，Processingで描画できる

1. 点を描く，線を描く，図形を描く関数
2. 線の色・太さ，塗りつぶしなどのオプションを決める関数

❖ ウィンドウの大きさを指定して，オプションを決めた後，
描画するプログラムを書くのが，Processingの基本

❖ 線を描くサンプルプログラム

```
size(600, 400)                          # ウィンドウの大きさ
strokeWeight(4.5)                       # 線の太さ
stroke(50.0, 60.0, 200.0)               # 線の色
line(100.0, 100.0, 300.0, 200.0) # 線を描画
```

基本的な描画関数の紹介

- **`strokeWeight(float w)`**: 線（枠線）の太さをwに設定する
- **`noStroke()`**: 線（枠線）を描画しない設定にする
- **`line(float x0, float y0, float x1, float y1)`**: 直線を引く
  - 座標(x0, y0)から，座標(x1, y1) へ線を描画
- **`rect(float x, float y, float w, float h)`**: 長方形を描く
  - 四角形の左上頂点(x, y)とし，幅w，高さhの長方形
- **`ellipse(float x, float y, float w, float h)`**: 楕円を描く
  - 中心座標(x, y)とし，幅w，高さhである楕円を描画
    - 幅と高さを同じ値にすると円となる

基本的な描画関数の紹介

- **`stroke(float r, float g, float b)`**: 線（枠線）の色を設定
- **`fill(float r, float g, float g)`**: 塗りつぶし色を設定
  - 赤成分r，緑成分g，青成分bは0から255の値とする
  - 赤成分r，緑成分g，青成分bの関係（代表例）

	赤成分r	緑成分g	青成分b	色見本
黒	0	0	0	
白	255	255	255	
赤	255	0	0	
緑	0	255	0	
青	0	0	255	
黄	255	255	0	
マゼンタ	255	0	255	
シアン	0	255	255	

基本的な描画関数の紹介

- **background(float r, float g, float b)**: 背景色を設定
  - 赤成分r，緑成分g，青成分bは0から255の値とする
- **background(float w)**: 背景色を設定
  - 1つの値しか指定しない場合，0(黒)～255(白)の範囲で
    グレースケールの背景となる
- 成分wの関係（代表例）

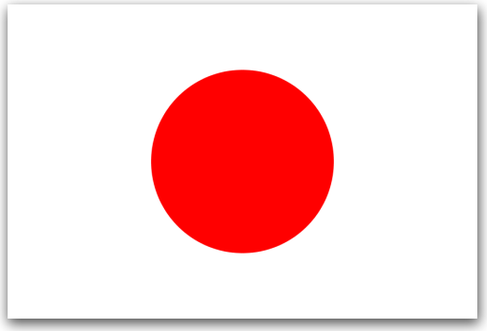
w	色見本
0	
64	
128	
192	
255	

描画の注意

- プログラムは上から下に順番に実行されるため，
  最初に描いた図形が最背面，最後に描いた図形が最前面となる
- 設定した線の太さ，線の色，塗りつぶし色に関しては，
  設定を変更しない限り，状態が継続される
- ウィンドウを作成した後，ウインドウの幅は変数**width**，
  ウインドウの高さは変数**height**として利用できる
  - これらの変数は，Processingが定義している変数なので，
    宣言することなく利用できる
  - ウインドウの中心の座標は(**width/2**, **height/2**)となる

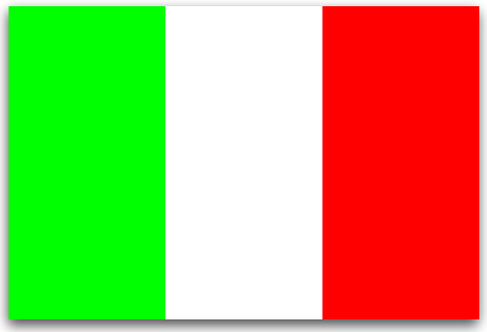
練習問題

1. 日本の国旗を描いてください



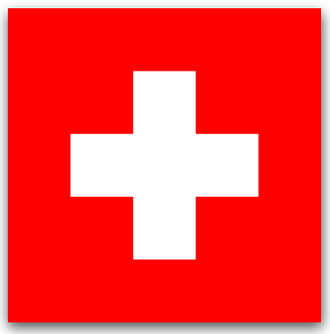
❖ 幅 : 高さ ＝ 3 : 2

2. イタリアの国旗を描いてください



❖ 幅 : 高さ ＝ 3 : 2

3. スイスの国旗を描いてください



❖ 幅 : 高さ ＝ 1 : 1

繰り返し処理との融合

❖ グラデーションの描画例

```
size(600, 300) # 画面の大きさ
noStroke()     # 枠線を消す

# 色を変えながら41個の長方形を描画する
# シアンから黄色へ
for i in range(41):
    fill(i * 255 / 40, 255, 255 - i * 255 / 40)
    rect(i * 20, 0, 20, 300)
```

Processingを使用したアニメーションの基礎

アニメーションの基礎

- 静止画を高速に切り替えることでアニメーションが実現できる
  - アニメーションにおいて，1秒間あたりに処理させる
    静止画像（フレーム）数のことをフレームレートとよぶ
  - 単位: fps（60fpsは1秒間に60フレーム切り替えている）

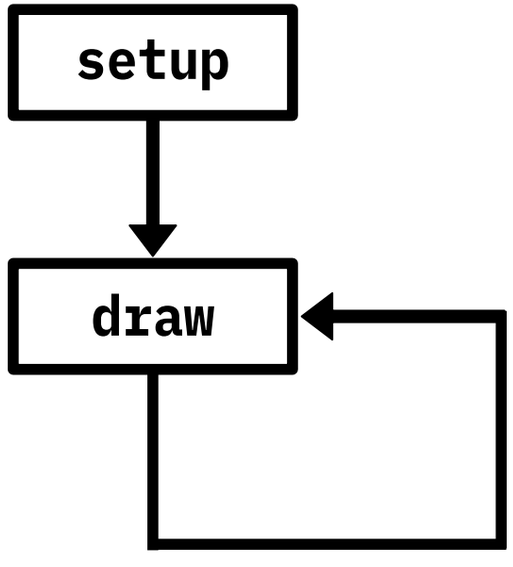
- Processingも同様に，各フレームを高速に切り替えることで，
  アニメーションを実現する
  - 初期化を行う**setup**ブロック（setup関数）と，
    フレームを描画する**draw**ブロック（draw関数）を定義する

アニメーションプログラム概要

- **setup**: 最初に1度だけ実行される
  - ウインドウのサイズ設定など1度だけ必要な処理を
    この中に記述する
- **draw**: フレームを表示するたびに実行される
  - ここで記載していることがアニメーションで表示される
    デフォルトでは1秒間に60回実行される

```
def setup():
  # 初期化処理をここに書く

def draw():
  # フレームごとの処理をここに書く
```



最初の1回だけ実行

1秒間に60回実行

アニメーション例1

```
# 1回だけ実行される
def setup():
  size(255, 255) # ウィンドウのサイズ

# setupを終えた後，繰り返し実行される
def draw():
  background(192)          # 背景色を決める
  z = frameCount % 256
  fill(z, 255 - z, 255)  # 円の塗りつぶし色を決める
  ellipse(z, z, 30, 30)  # 円を描画する
```

- **frameCount**: Processingが用意している特別な変数
  - プログラムが開始されてから表示されたフレーム数を格納
    - 0から始まり，drawが終わると1つ増える
  - 剰余を利用して，変数zの値が0から255となるように設定

アニメーション例2

```
# 1回だけ実行される
def setup():
  size(255, 255) # ウィンドウのサイズ

# setupを終えた後，繰り返し実行される
def draw():
  z = frameCount % 256
  fill(z, 255 - z, 255)  # 円の塗りつぶし色を決める
  ellipse(z, z, 30, 30)  # 円を描画する
```

- アニメーション例1のプログラムから『**background(192)**』を削除して実行した場合，どのようになるか確かめてください，また，どうしてそうなるか，その理由を考えてください

アニメーション失敗例

```
# 1回だけ実行される
def setup():
  size(255, 255) # ウィンドウのサイズ

# setupを終えた後，繰り返し実行される
def draw():
  x = 0
  background(192)          # 背景色を決める
  fill(0, 200, 200)        # 円の塗りつぶし色を決める
  ellipse(x, x, 30, 30)  # 円を描画する
  x += 1
```

- このプログラムは，アニメーションとしては成立しない
  - その理由を考えてみてください

グローバル変数

- setup関数，draw関数の外側で変数を宣言することで，プログラム（setup関数，draw関数）のどこからでも使用可能な変数のこと
  - プログラムが終わるまで，変数は存在する
  - draw関数が終わっとしても，値を維持できる
- 関数の中で宣言した変数はその関数でしか利用できない

```
# グローバル変数の宣言
x = 10

def setup():
  # 初期化処理をここに書く

def draw():
  # フレームごとの処理をここに書く
```

グローバル変数の使い方

- 関数中からグローバル変数の値を変更する場合,
- グローバル変数の値を変更する関数の先頭に **global 変数名** と記述する必要がある

```
# グローバル変数の宣言
x = 10

def setup():
  # 初期化処理をここに書く

def draw():
  # フレームごとの処理をここに書く
  global x # グローバル変数を変更するため
  x += 1   # グローバル変数を変更
```

アニメーション成功例

```
# グローバル変数の宣言
x = 0


# 1回だけ実行される
def setup():
    size(255, 255) # ウィンドウのサイズ

# setupを終えた後，繰り返し実行される
def draw():
    global x
    background(192)          # 背景色を決める
    fill(0, 200, 200)        # 円の塗りつぶし色を決める
    ellipse(x, x, 30, 30)  # 円を描画する
    x += 1
    x = x % 255
```

Processingを使用した物理ゲームの基礎

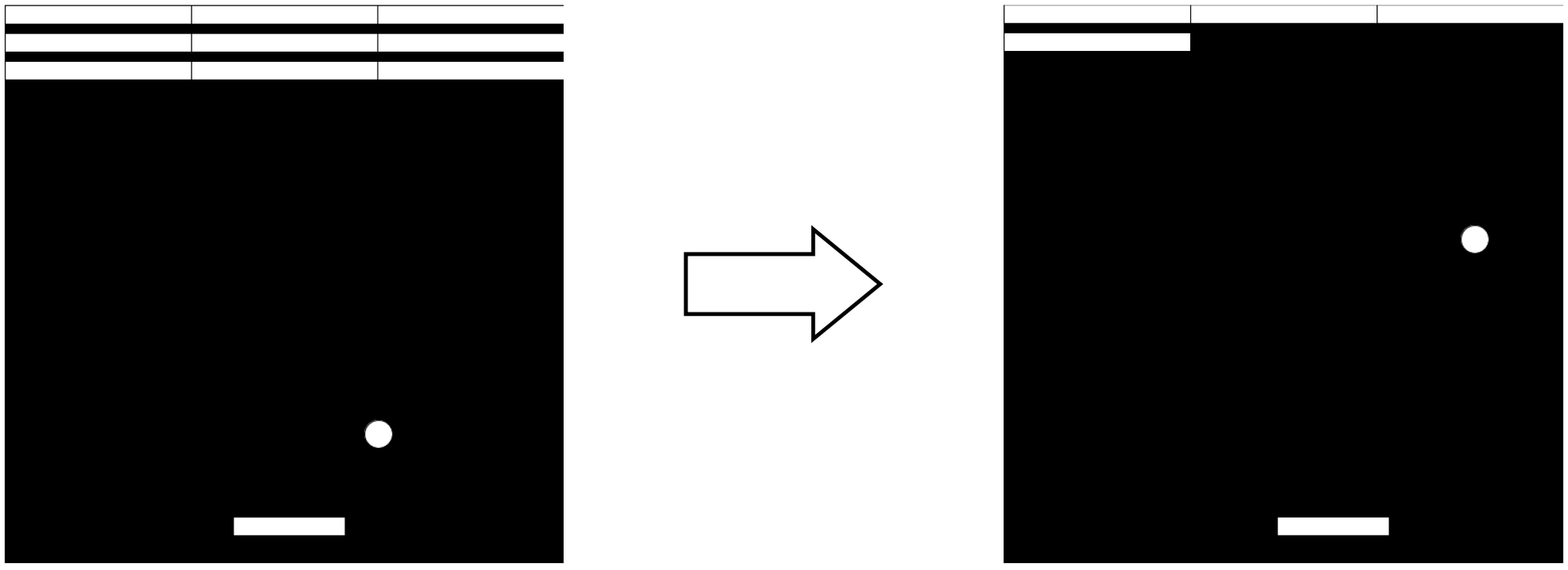
Processingの応用例

- マウス・キーボードとのインタラクション
- メディアファイル（画像，音楽，動画など）との連携
- 電子回路との連携
- スマートフォン，Webとの連携

- 本講義は，ステップバイステップ形式で，
  簡単な物理ゲームである『ブロック崩しゲーム』の基礎を学ぶ

ブロック崩しゲーム

❖ 1970年代後半から1980年代にかけて登場した反射型ゲーム



❖ 本講習では，1ブロックに対するブロック崩しゲームを学ぶ

Step 0 | ウィンドウ設定

- 画面の大きさ，背景色を設定する
  - 以下のプログラムでは，
    - 画面の大きさを幅600，高さ600
    - 背景色を白
  - としている

```
def setup():
    size(600, 600) # ウィンドウのサイズ

def draw():
    background(255) # 背景色を決める
```

Step 1 | バーをマウスの位置で変更

- 長方形バーをマウスの位置で変更できるようする
  - Processingでは，マウスの位置を取得できる特別な変数

    mouseX, **mouseY**という変数が存在する
  - バーの中心位置のx座標を**mouseX**にすれば良い

```
def draw():
    background(255)                                                    # 背景色を決める
    barWidth = 150                                                     # バーの長さ
    barHeight = 20                                                     # バーの高さ
    fill(0, 100, 255)                                                  # バーの色
    rect(mouseX - barWidth / 2, 550, barWidth, barHeight)              # バーを描画する
```

Step 2 | ボールの動作

❖ ボールの位置，速度をグローバル変数として宣言し，

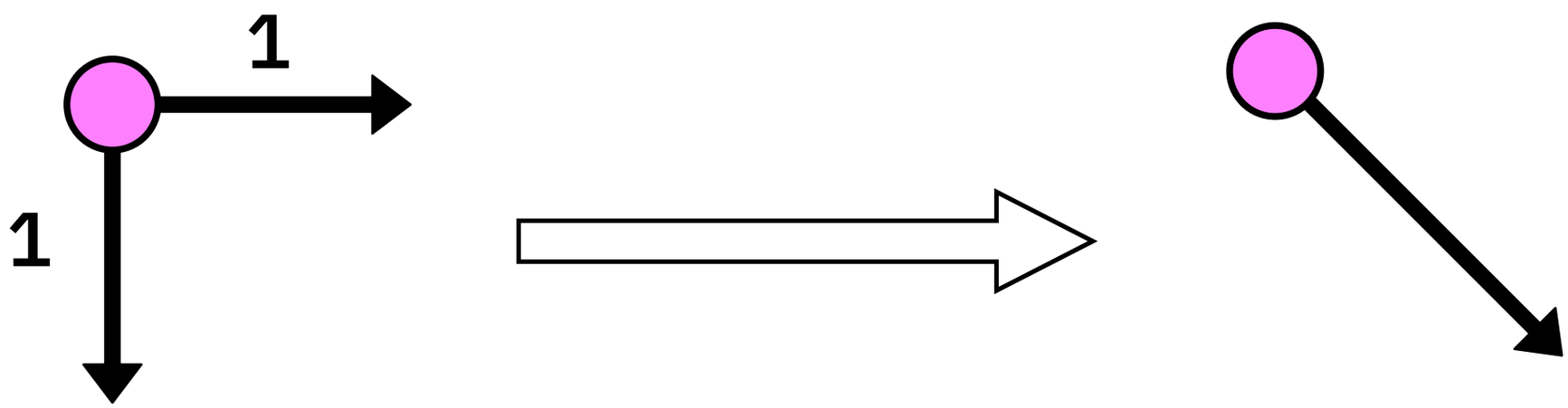
drawメソッドで更新する

❖ 速度は1フレームで動くピクセル量となる

❖ x方向とy方向の変化量を指定することで，平面を1方向に

移動することができる

❖ 三角比，ベクトルなどの考え方で以下は自明



xとyの変化量が同じ場合，45度の方向で進む

Step 2 | ボールの動作

- ボールの位置，速度をグローバル変数として宣言し，drawメソッドで更新する
  - 速度は1フレームで動くピクセル量となる

- グローバール変数の追加箇所

```
ballX = 50 # ボールのx座標
ballY = 50 # ボールのy座標
ballVx = 9 # ボールのx方向に対するの更新量
ballVy = 2 # ボールのy方向に対するの更新量

def setup():
    size(600, 600) # ウィンドウのサイズ
```

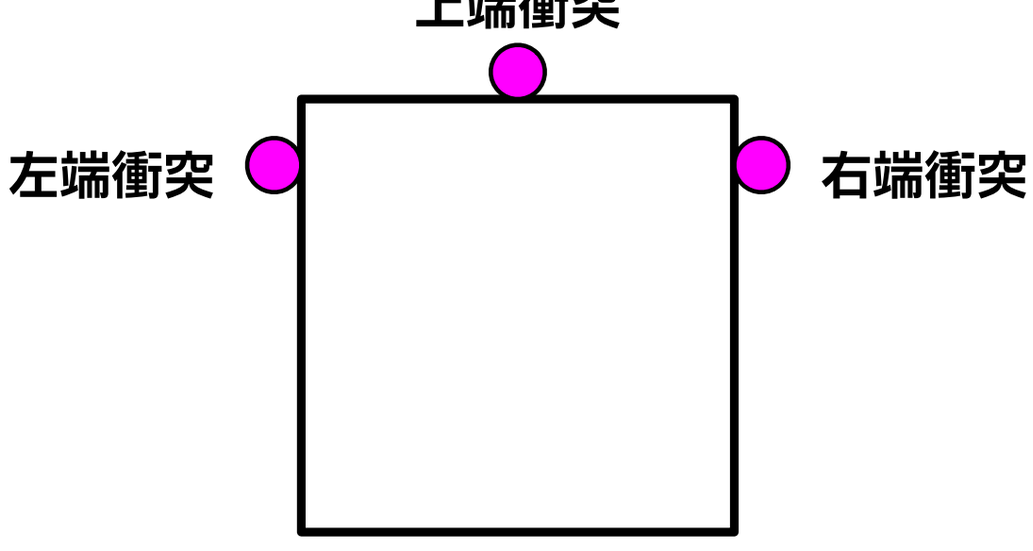
Step 2 | ボールの動作

❖ drawメソッドの追加箇所

```
def draw():
    global ballX, ballY, ballVx, ballVy # グローバル変数を変更する
    background(255)                                    # 背景色を決める
    barWidth = 150                                     # バーの長さ
    barHeight = 20                                     # バーの高さ
    fill(0, 100, 255)                                  # バーの色
    rect(mouseX - barWidth / 2, 550, barWidth, barHeight) # バーを描画する
    fill(200, 0, 255)                  # ボールの色
    ellipse(ballX, ballY, 30, 30) # ボールを描画する
    ballX += ballVx                    # ボールのx座標を更新する
    ballY += ballVy                    # ボールのy座標を更新する
```

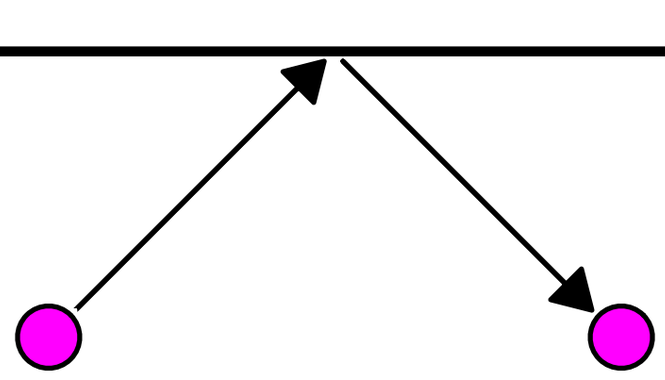
Step 3 | ボールの反射

- 上端，左端，右端にボールが衝突した場合，ボールが反射するようにする
  - 「衝突した場合」と「ボールを反射する」をプログラムで表現する
- プログラムでの考え方
  - 上端に衝突する
    - ボールのy座標が0より小さい場合
  - 左端に衝突する
    - ボールのx座標が0より小さい場合
  - 右端に衝突する
    - ボールのx座標がウィンドウ幅より大きい場合



Step 3 | ボールの反射

- ボールを反射するをプログラムで表現するために
  - 上端に衝突した場合
    - x方向の速度は変化せず，y方向の速度は逆になる



  - 左端に衝突した場合
    - x方向の速度は逆になり，y方向の速度は変化しない
  - 右端に衝突した場合
    - x方向の速度は逆になり，y方向の速度は変化しない

Step 3 | ボールの反射

- まとめると
  - 上端に衝突した場合（ボールのy座標が0より小さい場合）
    - x方向の速度は変化せず，y方向の速度は逆になる
  - 左端・右端に衝突した場合
    （ボールのx座標が0より小さい場合，または，
    ウィンドウ幅より大きい場合）
    - x方向の速度は逆になり，y方向の速度は変化しない

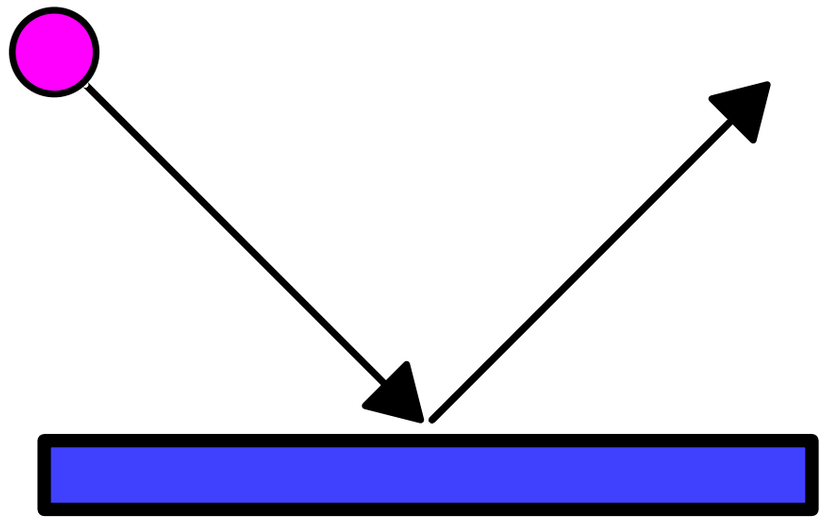
Step 3 | ボールの反射

```
def draw():
    global ballX, ballY, ballVx, ballVy # グローバル変数を変更する
    background(255)                                    # 背景色を決める
    barWidth = 150                                     # バーの長さ
    barHeight = 20                                     # バーの高さ
    fill(0, 100, 255)                                  # バーの色
    rect(mouseX - barWidth / 2, 550, barWidth, barHeight) # バーを描画する
    fill(200, 0, 255)                 # ボールの色
    ellipse(ballX, ballY, 30, 30) # ボールを描画する
    ballX += ballVx                   # ボールのx座標を更新する
    ballY += ballVy                   # ボールのy座標を更新する

    if ballY < 0:                      # 上端衝突
        ballVy *= -1
    if ballX < 0 or ballX > width: # 左右端衝突
        ballVx *= -1
```

Step 4 | ボールとバーの衝突判定

- ボールと上端，左端，右端との衝突と同様にして，
  ボールとバーの衝突を行う
  - ボールのy向き速度が正（下向き）で，かつ，
    四角形バーの内部に存在する場合に，
    ボールのy向き速度を逆（上向き）にする
- バーの左端と右端の座標を求めるとプログラムしやすい



Step 4 | ボールとバーの衝突判定

```
def draw():
    global ballX, ballY, ballVx, ballVy # グローバル変数を変更する
    background(255)                                          # 背景色を決める
    barWidth = 150                                           # バーの長さ
    barHeight = 20                                           # バーの高さ
    fill(0, 100, 255)                                        # バーの色
    rect(mouseX - barWidth / 2, 550, barWidth, barHeight) # バーを描画する
    fill(200, 0, 255)                 # ボールの色
    ellipse(ballX, ballY, 30, 30) # ボールを描画する
    ballX += ballVx                   # ボールのx座標を更新する
    ballY += ballVy                   # ボールのy座標を更新する

    if ballY < 0: # 上端衝突
        ballVy *= -1
    if ballX < 0 or ballX > width: # 左右端衝突
        ballVx *= -1

    leftBar = mouseX - barWidth / 2  # バーの左端座標
    rightBar = mouseX + barWidth / 2 # バーの右端座標

    # ボールとバーの衝突
    if ballVy > 0 and ballX > leftBar and ballX < rightBar and ballY > 550 and ballY < 550 + barHeight:
        ballVy *= -1
```

Step 5 | ブロックの設置

- ブロックの扱い方
  - ブロックにライフを設定して，衝突したらライフが無くなり，ライフが0になったら，ブロックが消えるようにする
- プログラム的な考え方
  - 消える → 選択処理でライフが0の場合，描画をしない
- グローバル変数の追加

```
ballX = 50 # ボールのx座標
ballY = 50 # ボールのy座標
ballVx = 9 # ボールのx方向に対するの更新量
ballVy = 2 # ボールのy方向に対するの更新量

blockLife = 1 # ブロックのライフ
```

Step 5 | ブロックの設置

```
def draw():
    global ballX, ballY, ballVx, ballVy, blockLife          # グローバル変数を変更する
    background(255)                                         # 背景色を決める
    barWidth = 100                                          # バーの長さ
    barHeight = 20                                          # バーの高さ
    fill(0, 100, 255)                                       # バーの色
    rect(mouseX - barWidth / 2, 550, barWidth, barHeight) # バーを描画する

    blockWidth = 70  # ブロックの幅
    blockHeight = 20 # ブロックの高さ
    if blockLife > 0:
        fill(200, 200, 255)                                             # ブロックの色
        rect((width - blockWidth) / 2, 30, blockWidth, blockHeight) # ブロックを描画する

    fill(200, 0, 255)                 # ボールの色
    ellipse(ballX, ballY, 30, 30) # ボールを描画する
    ballX += ballVx                   # ボールのx座標を更新する
    ballY += ballVy                   # ボールのy座標を更新する

    # 以下続く
```

Step 6 | ボールとブロックの衝突判定

- 衝突プログラムを参考にして，ボールとブロックの衝突を行う
- 衝突した場合，
  - ブロックのライフを1つ減らし，ボールのy向き速度を逆にする
    - 上方向，下方向からボールが当たる可能性があるため
      - 左右から当たる可能性もあるが，時間の都合上，割愛

Step 6 | ボールとブロックの衝突判定

```
def draw():
    # 続きから
    leftBar = mouseX - barWidth / 2
    rightBar = mouseX + barWidth / 2

    # ボールとバーの衝突
    if ballVy > 0 and ballX > leftBar and ballX < rightBar and ballY > 550 and ballY < 550 + barHeight:
        ballVy *= -1

leftBlock = (width - blockWidth) / 2
rightBlock = (width + blockWidth) / 2

# ボールとブロックの衝突
if ballX > leftBlock and ballX < rightBlock and ballY > 30 and ballY < 30 + blockHeight:
    ballVy *= -1
    blockLife -= 1
```

Step 7 | 文字列によるゲームインタラクション

- 画面に文字列を出力させ，ゲームの見栄えを良くする
  - ブロックを消したら「Game Clear !」と出力する
  - ボールをバーに当てることができない場合，
    「Game Over … 」と出力する
  - etc･･･

- Processingでは，以下の方法で文字列を画面に出力できる
  - **`textSize(size)`**: 文字の大きさをsizeにする
  - **`text(data, x, y)`**: 文字列dataを座標(x, y)に出力する

Step 7 | 文字列によるゲームインタラクション

```
def draw():
    # 続きから

    blockWidth = 70  # ブロックの幅
    blockHeight = 20 # ブロックの高さ
    if blockLife > 0:
        fill(200, 200, 255)                                          # ブロックの色
        rect((width - blockWidth) / 2, 30, blockWidth, blockHeight) # ブロックを描画する
    elif blockLife == 0: # GameClear処理
        fill(0)
        textSize(30)
        text("Game Clear !", 200, 200)

    fill(200, 0, 255)                # ボールの色
    ellipse(ballX, ballY, 30, 30) # ボールを描画する
    ballX += ballVx                  # ボールのx座標を更新する
    ballY += ballVy                  # ボールのy座標を更新する

    # GameOver処理
    if ballY > height:
        fill(0)
        textSize(30)
        text("Game Over...", 200, 400)

    # 以下続く
```

Step ∞ | 魔改造

- これまでのプログラムを参考にして，ゲームを魔改造する
  - ブロックを増やす
  - ブロックが移動する
  - ボールを増やす
  - 衝突するたびに，ボールのスピードが変わる
  - ブロックのライフを増やして，衝突するたびに，ブロックの色が変わる
  - etc･･･

- **発想次第でいくらでもゲームを魔改造できます**