

知能プログラミング演習 II 課題 5

グループ 8

29114003 青山周平

29114060 後藤拓也

29114116 増田大輝

29114142 湯浅範子

29119016 小中祐希

2019 年 12 月 10 日

■提出物 group08.pdf, group08.zip

■グループ グループ 8

■担当	氏名	担当課題
	青山周平	必須課題 5-4, 発展課題 5-7
	後藤拓也	必須課題 5-2
	増田大輝	必須課題 5-3, 発展課題 5-5
	湯浅範子	必須課題 5-4
	小中祐希	必須課題 5-1, 必須課題 5-2, 発展課題 5-8

1 課題の説明

必須課題 5-1 目標集合を変えてみたときに, 動作が正しくない場合があったかどうか, 実行例を示して考察せよ. また, もしあったならその箇所を修正し, どのように修正したか記せ.

必須課題 5-2 教科書のプログラムでは, オペレータ間の競合解消戦略としてランダムなオペレータ選択を採用している. これを, 効果的な競合解消戦略に改良すべく考察し, 実装せよ. 改良の結果, 性能がどの程度向上したかを定量的に (つまり数字で) 示すこと.

必須課題 5-3 上記のプランニングのプログラムでは、ブロックの属性（たとえば色や形など）を考えていないので、色や形などの属性を扱えるようにせよ。ルールとして表現すること。例えば色と形の両方を扱えるようにする場合、A が青い三角形、B が黄色の四角形、C が緑の台形であったとする。その時、色と形を使ってもゴールを指定できるようにする（"green on blue" や "blue on box" のように）

必須課題 5-4 上記 5-2, 5-3 で改良したプランニングシステムの GUI を実装せよ。ブロック操作の過程をグラフィカルに可視化し、初期状態や目標状態を GUI 上で変更できることが望ましい。

発展課題 5-5 ブロックワールド内における物理的制約条件をルールとして表現せよ。例えば、三角錐（pyramid）の上には他のブロックを乗せられない等、その世界における物理的な制約を実現せよ。

発展課題 5-6 ユーザが自然言語（日本語や英語など）の命令文によってブロックを操作したり、初期状態／目標状態を変更したりできるようにせよ。なお、命令文の動詞や語尾を 1 つの表現に決め打ちするのではなく、多様な表現を許容できることが望ましい。

発展課題 5-7 3 次元空間（実世界）の物理的な挙動を考慮したブロックワールドにおけるプランニングを実現せよ。なお、物理エンジン等を利用する場合、Java 以外の言語のフレームワークを使って実現しても構わない。

発展課題 5-8 教科書 3.3 節のプランニング手法を応用できそうなブロック操作以外のタスクをグループで話し合い、新たなプランニング課題を自由に設定せよ。さらに、もし可能であれば、その自己設定課題を解くプランニングシステムを実装せよ。

2 必須課題 5-3

上記のプランニングのプログラムでは、ブロックの属性（たとえば色や形など）を考えていないので、色や形などの属性を扱えるようにせよ。ルールとして表現すること。例えば色と形の両方を扱えるようにする場合、A が青い三角形、B が黄色の四角形、C が緑の台形であったとする。その時、色と形を使ってもゴールを指定できるようにする（"green on blue" や "blue on box" のように）

2.1 手法

ブロックの属性を扱えるようにするために、まずはルールによってブロックと属性の関係を表現し、それらの関係を保持する必要がある。その上で、属性によって表現されたゴールや、初期状態をブロックによる表現に変換し、後ろ向き推論を行う方針とした。

ここで、ブロックの属性によってゴールを表現する場合、そのまま後ろ向き推論を行うと、同一ブロックの異なる属性による表現がワーキングスペースに含まれる場合などへの対応が非常に複雑となる。したがって、入力されたルールと初期状態はあらかじめ属性表現をブロックに変換することで、もとの後ろ向き推論が正常に機能するような実装を行うものとする。

2.2 実装

まず、属性を扱うクラス `Attributions` を作成した。 `Attributions` クラスの変数として `HashMap<String,String>` 型の `attributions`, `List<String>` 型の `rules` がある。それぞれ、属性とブロックの関係 (属性が主キー) の集合、ルールの集合に対応する。この `Attributions` クラスには以下の3つのコンストラクタが存在し、全て上記変数への操作を行うものであるが、それぞれに次のような用途がある。

`Attributions()` テスト用のデフォルトコンストラクタ。デフォルトのルールが用意されている。ルールは自然言語で記述。

`Attributions(List <String>rules)` 自然言語で記述されたルールをもとに属性とブロックの関係を構築する。

`Attributions(HashMap <String, String>attributions)` あらかじめ、属性とブロックの関係が構築された状態のものを受け取る。

ただし、自然言語で記述されているものは、“ブロック is 属性”の形で表現されているものを扱うものとする。例えば、“A is blue”と記述されたものを受理する。

次に、自然言語から属性とブロックの関係を構築するメソッド `addAttribution(String attributionState)` についての説明を行う。このメソッドは上記コンストラクタ `Attributions()` と `Attributions(List <String>rules)` から呼び出される。これらのコンストラクタからは `rules` の各要素を引数として渡すものとする。 `addAttribution` メソッドは仮引数

attributionState でこれらを受け取り, "ブロック is 属性" を空白で分割したリストの第二要素が "is" である場合に, attributions に関係を加える. この時, 属性を主キーとし, ブロックを紐づけられるオブジェクトとする.

すなわち, "A is blue" とした場合には, 第二要素は "is" であるので "blue" を主キーとして, "A" と紐づけられることとなる.

以上により, 属性を持つブロックを扱うための下準備が整った.

続いて, attributions に格納された属性とブロックの関係をゴールリストや初期状態のリストに適応するためのメソッド editStatementList(ArrayList<String> statementList) についての説明である. このメソッドでは, 仮引数として渡された statementList の各要素について, attributions 内に格納されている属性との照合作業が行われ, 属性がある場合には紐づけられたブロック名に置き換えられる作業が行われる. 以下にメソッドのソースコードを示す.

ソースコード 1 editStatementList メソッドの実装

```
1      ArrayList<String> editStatementList(ArrayList<String> statementList
      ) {
2          System.out.println("++++++ EditStatement ++++++");
3          ArrayList<String> newStatementList = new ArrayList();
4          for (String statement: statementList) {
5              List<String> tokens = Arrays.asList(statement.
                  split(" "));
6              String newStatement = "";
7              for(int tokenNum = 0; tokenNum < tokens.size
                  (); tokenNum++) {
8                  String token = tokens.get(tokenNum);
9                  if(attributions.containsKey(token)) {
10                     token = attributions.get(token
                        );
11                 }
12                 newStatement += token;
13                 if(tokenNum < tokens.size()-1) {
14                     newStatement += " ";
15                 }
16             }
        }
```

```

17             newStatementList.add(newStatement);
18             System.out.println(statement+" =====> "+
                               newStatement);
19         }
20         return newStatementList;
21     }

```

特に,9 11 行目で HashMap の機能を利用して属性からブロック名を導き出す処理を行っている.

以上によって, 自然言語で記述されたルール集合から導出された属性とブロックの関係や与えられた属性とブロックの関係を元にして, ゴールリストや初期状態のリストに含まれる属性表現をブロック名に統一する処理が完成した.

具体的な利用法は, 使用したい任意のコンストラクタによって Attributions クラスを初期化し, 以下のような形で, ゴールリストや初期状態のリストに適用すればよい.

ソースコード 2 editStatementList の利用

```

1    goalList = attributions.editStatementList(initAttributeGoalList());
2    initialState = attributions.editStatementList(
        initAttributeInitialState());

```

2.3 実行例

以下に,CUI 上での実行結果を示す. ただし, コンストラクタはデフォルトコンストラクタを使用するものとする.

ソースコード 3 ゴールと初期状態に属性を指定した場合の実行結果

```

1    ~/Programming2/Work5
2    ●java Planner [ re-fix/attribute-rule ]
3    ===== goal:green on ball =====
4    ===== goal:blue on pyramid =====
5    ++++++ EditStatement ++++++
6    green on ball =====> B on C
7    blue on pyramid =====> A on B
8    ----- initInitialState:clear blue -----
9    ----- initInitialState:clear green -----
10    ----- initInitialState:clear red -----
11    ----- initInitialState:ontable box -----

```

```

12  ----- initInitialState:ontable pyramid -----
13  ----- initInitialState:ontable ball -----
14  ----- initInitialState:handEmpty -----
15  ++++++ EditStatement ++++++
16  clear blue =====> clear A
17  clear green =====> clear B
18  clear red =====> clear C
19  ontable box =====> ontable A
20  ontable pyramid =====> ontable B
21  ontable ball =====> ontable C
22  handEmpty =====> handEmpty
23  *** GOALS ***[B on C, A on B]
24  **B on C
25  Place B on C
26  *** GOALS ***[clear C, holding B]
27  **clear C
28  [clear A, clear B, clear C, ontable A, ontable B, ontable C,
    handEmpty]
29  *** GOALS ***[holding B]
30  **holding B
31  pick up B from the table
32  *** GOALS ***[ontable B, clear B, handEmpty]
33  **ontable B
34  [clear A, clear B, clear C, ontable A, ontable B, ontable C,
    handEmpty]
35  *** GOALS ***[clear B, handEmpty]
36  **clear B
37  [clear A, clear B, clear C, ontable A, ontable B, ontable C,
    handEmpty]
38  *** GOALS ***[handEmpty]
39  **handEmpty
40  pick up B from the table
41  Place B on C
42  [clear A, ontable A, ontable C, B on C, clear B, handEmpty]
43  *** GOALS ***[A on B]
44  **A on B
45  Place A on B
46  *** GOALS ***[clear B, holding A]

```

```

47      **clear B
48      [clear A, ontable A, ontable C, B on C, clear B, handEmpty]
49      *** GOALS ***[holding A]
50      **holding A
51      pick up A from the table
52      *** GOALS ***[ontable A, clear A, handEmpty]
53      **ontable A
54      [clear A, ontable A, ontable C, B on C, clear B, handEmpty]
55      *** GOALS ***[clear A, handEmpty]
56      **clear A
57      [clear A, ontable A, ontable C, B on C, clear B, handEmpty]
58      *** GOALS ***[handEmpty]
59      **handEmpty
60      pick up A from the table
61      Place A on B
62      ***** This is a plan! *****
63      pick up B from the table
64      Place B on C
65      pick up A from the table
66      Place A on B

```

まず,3・4行目で属性によってゴールが表現されていること示している. その次に,5 7 行目でこれらのゴールを属性からブロック名による表現に変換している. 続いて,8 14 行目で属性によって式状態が表現されていることを示している. 先ほどと同様に,15 22 行目では,これらの表現を属性によるものからブロック名のみの形に変更している. したがって,以降の実行結果はブロック名のみを用いた場合と同様となる.

2.4 考察

属性を扱うに当たって,初めは属性のまま推論を行い,プランニングを実行できるようにしようと考えた. しかし,同じブロックの属性であっても,異なる属性同士において,字面だけではマッチングが成功しないことに気が付いた. 例えば”A is blue”と”A is box”がルールとして与えられていたとする. この時,ゴールが”blue on green”で,状態として”box on green”が保持されているものとする. 本来,blue と box は同じブロック A の属性であるが,字面の上では,異なる定数として認識される.

これらを同一のブロック A を指し示すものとして認識するには,属性からブロックその

ものへと変換する機能が必要である。したがって、変換後のゴール”A on B”と状態”A on B”であればマッチングは成功する。

ここで、もう一度属性のまま推論を行う意義を考えることとした。上記の方法では、マッチングにおいて最終的に属性からブロック名への変換処理を行っている。ところが、初めから同様に属性をブロック名に変換する手法にすることでもマッチングは可能である。例えば、あらかじめゴールを”blue on green”から”A on B”に変換し、初期状態を全てブロック名による表現にすることで、プランニング中の状態を”A on B”とする。結果として、即座に”A on B”と”A on B”のマッチングが行われるようにする。

加えて、後者の方がゴールリストと初期状態のリストから属性表現をブロック名に変換するのみであるので、処理も非常にシンプルとなる。後ろ向き推論の特徴と処理の簡潔さから、後者の手法を採用することにした。

複数ある実現方法の中から、既存のプログラムとの相性や論理的な汎用性を考えてもっとも良いと思える方法を選択できたと考えている。

3 必須課題 5-4

上記 5-2, 5-3 で改良したプランニングシステムの GUI を実装せよ。

ブロック操作の過程をグラフィカルに可視化し、初期状態や目標状態を GUI 上で変更できることが望ましい。

湯浅の担当箇所は、得られた結果を基にした GUI 本体の実装である。

青山の担当箇所は、必須課題 5-4 における GUI と Planner.java との間でデータの仲介を行う Presenter の制作である。

3.1 手法

(湯浅担当)

GUI で求められる出力が行えるように、以下のような機能を加えた。

- ブロック操作の過程を示す
- ブロック操作の過程をグラフィカルに可視化する
- 初期状態と目標状態を GUI 上から画面入力で変更できる
- 属性の決定を GUI 上から行える

- 操作に必要なオペレータと属性のデータを表示する
- 禁止制約によってブロック操作が停止した場合にそれを表示する

これらのプログラムを実装するためのデータ受け渡し部分は青山君が作成してくれるため、ここでは受け取ったデータを基にしてどのような方法で表示を行うか考えながら実装を行った。実装は Swing によって行った。

また今回は初めての GUI 作成であったため、第一週では受け取るデータを既已取得したと仮定し、自ら作成した仮データを基にして GUI についての検討を行った。第二週では実際にデータを受け取り、そのデータを基に正しく描画やプランニングが行えるようにプログラムを改良した。

(青山担当)

Presenter を実装するにあたり、以下のような方針を立てた。

1. Planner.java のデータを、外部から取り出し、セットできるように改良する。
2. 導かれたプランの導出過程を渡せるようにする。

1. に関して、MVP アーキテクチャを導入し、GUI と Planner 間のデータのやり取りを、Presenter によって緩衝することで、拡張性の向上と GUI 担当者の負担軽減を図った。

2. に関して、GUI 担当者が受け取りやすい渡し方を相談して決めることで、柔軟に対応した。

3.2 実装

(湯浅担当)

作成した GUI プログラムの中の主部分についての実装を以下に示す。

まず、GUI プログラムの main メソッドをソースコード 4 に示す。

ソースコード 4 main メソッド

```

1 public static void main(String[] args){
2     // 画像の定義
3     images = new String[5][4];
4     images[0][0] = "image/squBt.png";
5     ...

```

```

6         images[4][3] = "image/daiDeft.png";
7
8         // 対応番号の指定 (色・形それぞれで定義)
9         imageMapC = new HashMap<>();
10        imageMapC.put("Blue", 0);
11        ...
12        imageMapC.put("Default", 4);
13        imageMapS = new HashMap<>();
14        ...
15
16        // 属性指定での入出力に対応 (default は予め弾く)
17        Attribution = new HashMap<>();
18        Attribution.put("Blue", 1);
19        ...
20        Attribution.put("Yellow", 1);
21        Attribution.put("box", 2);
22        ...
23        Attribution.put("trapezoid", 2);
24
25        PGUI frame = new PGUI();
26        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27        frame.setBounds(10, 10, 650, 450);
28        frame.setTitle("プランニングシステム");
29        frame.setVisible(true);
30    }

```

ここでフレームの大きさと表示位置・タイトルや、フレームを閉じると同時にプログラムを終了させるなどの動作を定めた。さらに、プログラム中で使用する変数であらかじめ値を設定しておくものの定義を行った。

次に、GUI プログラムのコンストラクタをソースコード 5 に示す。

ソースコード 5 コンストラクタ

```

1 PGUI(){
2     // プレゼンターとの連結
3     presenter = new Presenter();
4     // 結果の格納 (メソッド呼び出し)
5     ArrayList<String> result = presenter.getPlan();

```

```

6      results = new ArrayList<>(result);
7      // 初期状態の追加
8      results.add(0, "default position");
9      // 結果ステップデータの取得
10     pUR = presenter.getStepList();
11     // 入力デフォルト値の格納 (メソッド呼び出し)
12     initialState = presenter.getInitialState();
13     // 出力デフォルト値の格納 (メソッド呼び出し)
14     goalList = presenter.getGoalList();
15     // 初期状態の格納
16     String[] initialName = {"A", "B", "C"};
17     String[] initialAColor = {"Default", "Default", "Default"};
18     String[] initialAShape = {"default", "default", "default"};
19     for (int i = 0; i < initialName.length; i++) {
20         modelName.addElement(new String(initialName[i]));
21         modelColor.addElement(new String(initialAColor[i]));
22         modelShape.addElement(new String(initialAShape[i]));
23     }
24     // 禁止制約の格納
25     prohibitRules.add("box on pyramid");
26     ...
27     prohibitRules.add("trapezoid on ball");
28
29     // 2ページ目以降のカード作成用メソッド
30     createResultPage(pUR);
31     // ボタンの作成メソッド
32     createButton();
33     // 最終処理メソッド
34     finishData();
35 }

```

PGUI コンストラクタで Presenter プログラムを起動し、プランニングを行った結果を取得する。また同時に初期状態を GUI で表示するため、これらの情報も格納した。

ここから、先に述べた各機能の実装について詳しく記述していく。
まず始めにブロック操作の過程を示すプログラムをソースコード 6 に示す。

ソースコード 6 過程表示

```
1 JPanel toString2 = new JPanel();
2 toString2.setLayout(new BoxLayout(toString2, BoxLayout.PAGE_AXIS));
3 toString2.setBackground(Color.WHITE);
4 ArrayList<String> printResult = presenter.getPlan();
5 toString2.add(new JLabel("***** This is a plan! *****"));
6 for (String printR : printResult) {
7     toString2.add(new JLabel(printR));
8 }
9 JScrollPane scrollpane2 = new JScrollPane(toString2);
10 scrollpane2.setPreferredSize(new Dimension(200, 310));
11 BevelBorder border2 = new BevelBorder(BevelBorder.LOWERED);
12 scrollpane2.setBorder(border2);
13 JPanel Plan = new JPanel();
14 Plan.setLayout(new BoxLayout(Plan, BoxLayout.PAGE_AXIS));
15 Plan.add(new JLabel("Plan "));
16 Plan.add(scrollpane2);
```

ブロック操作の過程を示すため、新しく作成したパネルに受け取ったリストデータをラベルとして加え、そのパネルをフレームに追加することで表示を行った。また経路が長くなったときを考え、必要に応じてスクロールバーを追加できるようにした。

次にブロック操作の過程をグラフィカルに可視化するためのプログラムをソースコード 7, 8 に示す。このとき初期状態は別で取得するため、初期状態のみを描画するプログラムと、以降の過程を描画するプログラムを分けて作成した。

ソースコード 7 グラフィカル表示 1(一部抜粋)

```
1 // 変数決定, 属性名の初期化, 配置用・アーム用座標配列を定義
2 ...
3 // 2ページ目の設定
4 JPanel page2 = new JPanel();
5 JLabel[] [] p2Label = new JLabel[row][col];
6 GridLayout page2layout = new GridLayout();
7 page2layout.setRows(row); // 行数
8 page2layout.setColumns(col); // 列数
9 page2.setLayout(page2layout);
10 // テーブルの上に乗っているブロックの初期化
```

```

11 int next = 0;
12 for (String dataS : dataTable) {
13     String[] state = dataS.split(" ", 0);
14     for (int i = 0; i < blocks.size(); i++) {
15         // 初めに入手したブロック名が何番目のものかcheck
16         if (state[1].equals(blocks.get(i))) {
17             // 名称一致のとき
18             iconX[i] = row - 1;
19             iconY[i] = next;
20             next++;
21         }
22     }
23 }
24 // 他ブロックの上に乗っているブロックの初期化
25 for (String dataS : dataOn) {
26     String[] state = dataS.split(" ", 0);
27     for (int i = 0; i < blocks.size(); i++) {
28         // それぞれの属性の番号を取得
29         if (state[0].equals(blocks.get(i))) {
30             ue = i;
31         } else if (state[2].equals(blocks.get(i))) {
32             sita = i;
33         }
34     }
35     // 上部分の座標の確定
36     iconX[ue] = iconX[sita] - 1;
37     iconY[ue] = iconY[sita];
38 }
39 // block とアームの上書き
40 for (int i = 0; i < blocks.size(); i++) {
41     p2Label[iconX[i]][iconY[i]] = new JLabel(icon[i]);
42     p2Label[iconX[i]][iconY[i]].setText(iconName[i]);
43 }
44 p2Label[armX][armY] = new JLabel(arm);
45 p2Label[armX][armY].setText(armname);
46 // アイコンの挿入
47 for (int i = 0; i < row; i++) {
48     for (int j = 0; j < col; j++) {

```

```

49         page2.add(p2Label[i][j]);
50     }
51 }

```

ソースコード 8 グラフィカル表示 2(一部抜粋)

```

1 // 3ページ目以降
2 for (int i = 0; i < cardPage; i++) {
3     // 初期化 (2ページ目作成と同様の操作)
4     ...
5     String hatenaX = pUR.get(i).getBindings().get("?x");
6     String hatenaY = pUR.get(i).getBindings().get("?y");
7     int hXz = blocks.indexOf(hatenaX);
8     int hYz = blocks.indexOf(hatenaY);
9     if (pUR.get(i).getName().equals("Place ?x on ?y")) {
10         // x の操作
11         iconX[hXz] = iconX[hYz] - 1;
12         iconY[hXz] = iconY[hYz];
13         // アームの操作
14         armX = iconX[hXz] - 1;
15         armY = iconY[hYz];
16     } else if (pUR.get(i).getName().equals("remove ?x from on top
        ?y")) {
17         // x の操作
18         iconX[hXz] = iconX[hXz] - 1;
19         iconY[hXz] = iconY[hYz];
20         // アームの操作
21         armX = iconX[hXz] - 1;
22         armY = iconY[hYz];
23     } else if (pUR.get(i).getName().equals("pick up ?x from the
        table")) {
24         // x の操作
25         iconX[hXz] = iconX[hXz] - 1;
26         iconY[hXz] = iconY[hXz];
27         // アームの操作
28         armX = iconX[hXz] - 1;
29         armY = iconY[hXz];
30     } else if (pUR.get(i).getName().equals("put ?x down on the

```

```

        table")) {
31         // x の操作, アームの操作
32         iconX[hXz] = row - 1;
33         armX = iconX[hXz] - 1;
34         boolean umu;
35         for (int j = 0; j < col; j++) {
36             umu = true;
37             for (int k = 0; k < iconY.length; k++) {
38                 if (j == iconY[k]) {
39                     umu = false;
40                     break;
41                 }
42             }
43             if (umu == true) {
44                 iconY[hXz] = j;
45                 armY = j;
46                 break;
47             }
48         }
49     }
50     // block・アームの上書き, アイコンの挿入
51     ...
52 }

```

上のように、初期状態は別のメソッドから受け取り予め描画を行い、そのデータを基にしてブロック操作の過程を描画した。また描画は座標で管理を行った。

さらに、初期状態と目標状態・各ブロックの属性を GUI 上から変更、表示するためのプログラムをソースコード 9 と 10 に示す。

ソースコード 9 初期状態と目標状態の変更

```

1 // 手動入力用パネル
2 JPanel natural = new JPanel();
3 natural.setLayout(new BoxLayout(natural, BoxLayout.PAGE_AXIS));
4 // 入力 (setInitialState)
5 JPanel sI = new JPanel();
6 iArea = new JTextArea(9, 20);
7 JScrollPane iScroll = new JScrollPane(iArea);

```

```

8 String ii = "";
9 for(String i : initialState) {
10     ii += i + "\n";
11 }
12 iArea.setText(ii);
13 sI.add(iScroll);
14 // 入力 (setGoal)
15 JPanel sG = new JPanel();
16 gArea = new JTextArea(4, 20);
17 JScrollPane gScroll = new JScrollPane(gArea);
18 String gg = "";
19 for(String g : goalList) {
20     gg += g + "\n";
21 }
22 gArea.setText(gg);
23 sG.add(gScroll);
24 natural.add(sI);
25 natural.add(sG);

```

JTextArea を用いて文字列の入力が GUI 上で行えるようにした。また、入力文が増えた場合のためスクロールバーを必要に応じて表示した。

ソースコード 10 各ブロックの属性変更

```

1 JPanel allRadio = new JPanel();
2 // 色選択
3 JPanel p2 = new JPanel();
4 radio = new JRadioButton[4];
5 radio[0] = new JRadioButton("Blue");
6 ...
7 // ボタンのグループ化
8 ButtonGroup group = new ButtonGroup();
9 group.add(radio[0]);
10 ...
11 p2.add(new JLabel("Select Color"));
12 p2.add(radio[0]);
13 ...
14 // 形状選択 (色選択と同様の操作)

```



```

15 ...
16
17 // 属性編集用ボタンの作成
18 JPanel ADS = new JPanel();
19 ADS.add(new JLabel("new Name "));
20 newNameText = new JTextField(5);
21 ADS.add(newNameText);
22 JButton add = new JButton("追加");
23 add.addActionListener(this);
24 add.setActionCommand("addButton");
25 ADS.add(add);
26 ...
27 allRadio.add(ADS);
28 allRadio.add(p2);
29
30 // 属性入力用パネルの作成
31 JPanel attribution = new JPanel();
32 JPanel p4 = new JPanel();
33 // 属性の決定用パネル
34 JPanel namedD = new JPanel();
35 namedD.add(new JLabel("Determine Attribution "));
36 p4.add(namedD);
37 JPanel attribute = new JPanel();
38 JPanel NAME = new JPanel();
39 ...
40 // リストで実現
41 namelist = new JList(modelName);
42 JScrollPane namesp = new JScrollPane();
43 namesp.getViewPort().setView(namelist);
44 NAME.add(namesp);
45 attribute.add(NAME);
46 // 色選択リストパネル
47 colorlist = new JList(modelColor);
48 JScrollPane colorsps = new JScrollPane();
49 colorsps.getViewPort().setView(colorlist);
50 // リストを選択不可にする
51 colorlist.setEnabled(false);
52 colorsps.setBorder(borderC);

```

```

53 JPanel COLOR = new JPanel();
54 ...
55 COLOR.add(colorsp);
56 attribute.add(COLOR);
57 // 形状選択リストパネル（色選択と同様の操作）
58 ...
59 p4.add(attribute);

```

今回の実装では色と形状は任意に選択することが出来ないため、これらをラジオボタンによりユーザーが選択するようにした。さらに、これらのデータの追加・削除・編集も行えるよう、属性データは JList を用いて管理を行った。また、新しい属性名の追加のためテキストフィールドを作成した。

また、操作に必要なオペレータを表示するプログラムをソースコード 11 に示す。

ソースコード 11 オペレータ表示プログラム

```

1 JPanel toString = new JPanel();
2 toString.setLayout(new BoxLayout(toString, BoxLayout.PAGE_AXIS));
3 for (Operator operator : operators) {
4     toString.add(new JLabel("●Operator" + i));
5     toString.add(new JLabel("NAME: " + operator.getName()));
6     toString.add(new JLabel("ADD: " + operator.getAddList()));
7     toString.add(new JLabel("DELETE: " + operator.getDeleteList
8         ()));
9 }
10 JScrollPane scrollpane = new JScrollPane(toString);
11 scrollpane.setBorder(border);

```

最後に、禁止制約によってブロック操作が停止した場合の処理を行うプログラムをソースコード 12 に示す。

ソースコード 12 ブロックの状態を表すプログラム

```

1 JPanel prohibit = new JPanel();
2 JPanel hosoku = new JPanel();
3 LineBorder inborder = new LineBorder(Color.red, 2);
4 TitledBorder border = new TitledBorder(inborder, "Warning!!",
5     TitledBorder.LEFT, TitledBorder.TOP);

```

```

5 JLabel setumei = new JLabel(" This Goal is not allowed by
    ProhibitRules");
6 hosoku.add(setumei);
7 ...
8 hosoku.setBorder(border);
9 JPanel kari = new JPanel();
10 kari.add(hosoku, BorderLayout.CENTER);
11 prohibit.add(kari);
12 // 禁止制約のパネルの表示
13 // 属性名
14 JPanel prohibit2_1 = new JPanel();
15 for (int i = 0; i < modelName.size(); i++) {
16     StringBuilder buf_1 = new StringBuilder();
17     buf_1.append((String)modelName.get(i));
18     buf_1.append(" is ");
19     ...
20     buf_1.append((String)modelShape.get(i));
21     prohibit2_1.add( new JLabel( buf_1.toString()) );
22 }
23 JScrollPane scrollpane1 = new JScrollPane(prohibit2_1);
24 JPanel AN = new JPanel();
25 JPanel an = new JPanel();
26 an.add( new JLabel("Attribution ") );
27 ...
28 AN.add(an);
29 AN.add(scrollpane1);
30 // 禁止制約と目標状態（属性名と同様の操作を行う）
31 ...
32 // 目標状態・禁止制約・属性名の設定
33 JPanel details = new JPanel();
34 details.add(AN);
35 details.add(PR);
36 details.add(GL);
37 prohibit.add(details);

```

目標状態が禁止制約に当たる場合は、目標状態の実現が不可能であるため、その目標状態が禁止制約に当たることを知らせる必要がある。今回はそのために新しくページを追加し、設定した属性名と禁止制約・目標状態を表示させ、この状態が禁止制約に当たること

を知らせることでブロック操作が停止した場合の処理を行った。

(青山担当)

Presenter.java には以下のクラスが含まれる。

Presenter テスト用の main メソッド，ゲッター，加工後のデータを受け取れるセッター，導出過程を Planner から取得するメソッド等を実装したクラス。

3.2.1 Planner.java のデータを，外部から取り出し，セットできるように改良する。

初期化をコンストラクタで行わせたり，クラスフィールドに値を移行したりすることで実現した。

3.2.2 導かれたプランの導出過程を渡せるようにする。

得られた plan を渡すために，Planner クラス内に Operator クラスの planUnifiedResult フィールドを追加し，Operator クラス内に HashMap クラスの bindings フィールドを追加した。bindings は変数束縛を示すものであり，getBindings により取得できるようにした。

具体的な渡し方としては，Planner のプランニングにより得られた plan を，全オペレータと unify して，当てはまるものがあつたら unify により明らかになった変数束縛を引数に，新たな Operator インスタンス生成し，planUnifiedResult に追加して planUnifiedResult を渡すというものである。

この挙動をソースコード 13 に示す。

ソースコード 13 Planner クラスの start メソッドの一部

```
1 public void start() {
2     ...
3     planning(goalList, initialState, theBinding);
4
5     System.out.println("***** This is a plan! *****");
6     planResult = new ArrayList<>();
7     planUnifiedResult = new ArrayList<>();
8     for (int i = 0; i < plan.size(); i++) {
9         Operator op = (Operator) plan.get(i);
10        Operator result = (op.instantiate(theBinding));
11        System.out.println(result.name);
```

```
12         planResult.add(result.name);
13         for(Operator initOp : operators) {
14             Unifier unifier = new Unifier();
15             if(unifier.unify(result.name, initOp.getName())) {
16                 planUnifiedResult.add(new Operator(initOp, unifier.getVars
17                     ()));
18             }
19         }
20     }
21 }
```

3.3 実行例

(湯浅担当)

作成したプログラムの実行結果を順に示す.

実行すると初めに初期状態・目標状態・属性の決定画面になる (図 1). デフォルトでは ABC の 3 つのブロックが存在し, その属性は全てデフォルト (黒色の四角形) で定義されている.

プランニングシステム

First Prev Next Last

Determine Attribution

Name	Color	Shape
A	Default	default
B	Default	default
C	Default	default

is &

new Name
[]

追加
削除
編集

Select Color

☐ Blue
☐ Green
☐ Red
☐ Yellow

Select Shape

☐ box
☐ pyramid
☐ ball
☐ trapezoid

SetInitial

clear A
clear B
clear C
ontable A
ontable B
ontable C
handEmpty

SetGoal

B on C
A on B

Planning

図 1 初期状態と目標状態, 属性の決定画面

初めに実行を行った段階で、デフォルト状態でのプランニングが行われているため、ページ上部の Next ボタンを押すと次の画面に遷移し、ブロック操作の過程の描画画面となる。Next ボタンを押すと次の状態へ、Prev ボタンを押すと一つ前の状態へ遷移する様子が確認できる。(図 2, 図 3, 図 4).

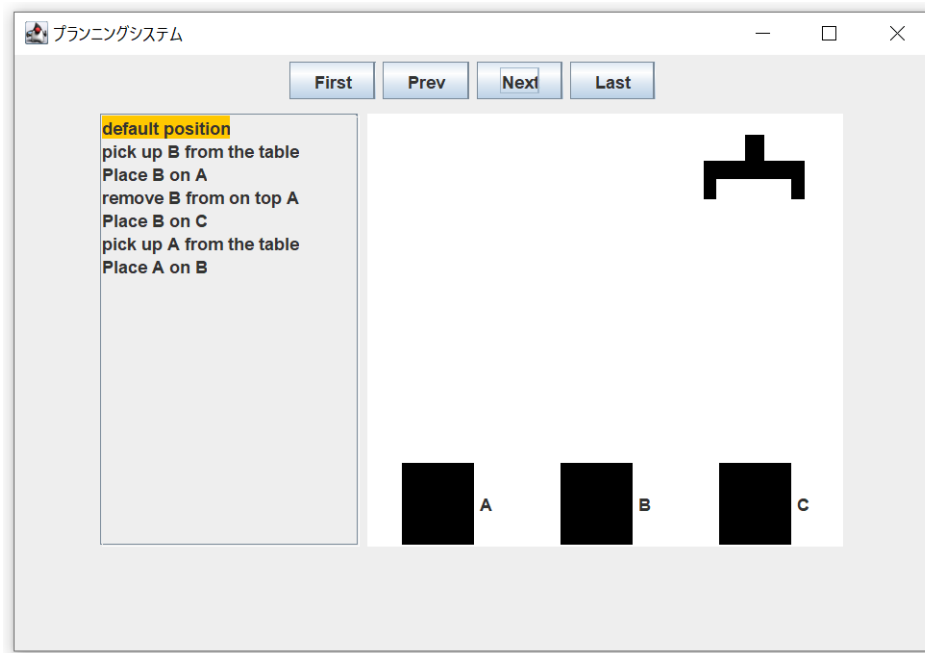


図 2 ブロック操作の過程 1

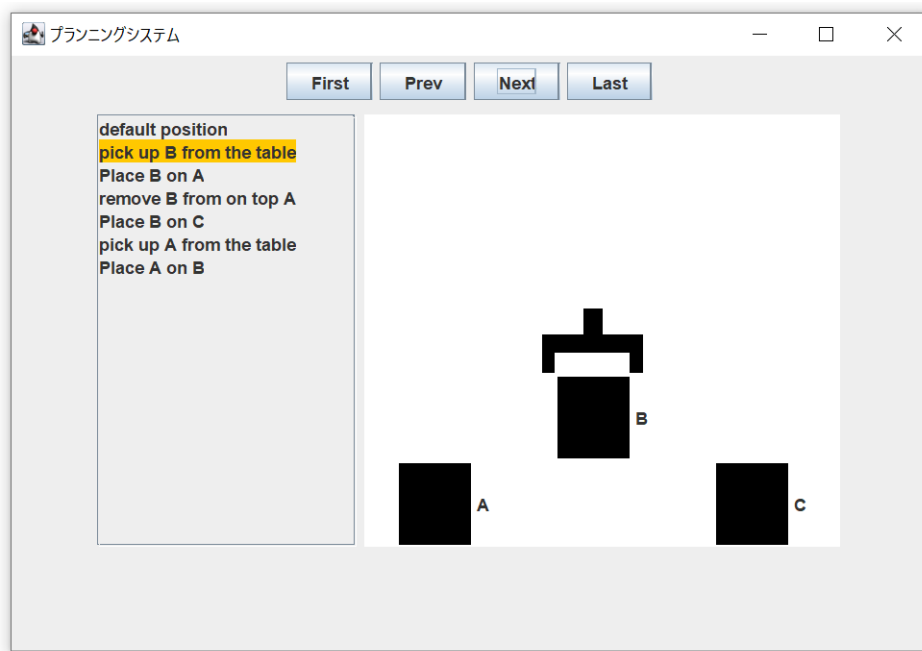


図 3 ブロック操作の過程 2

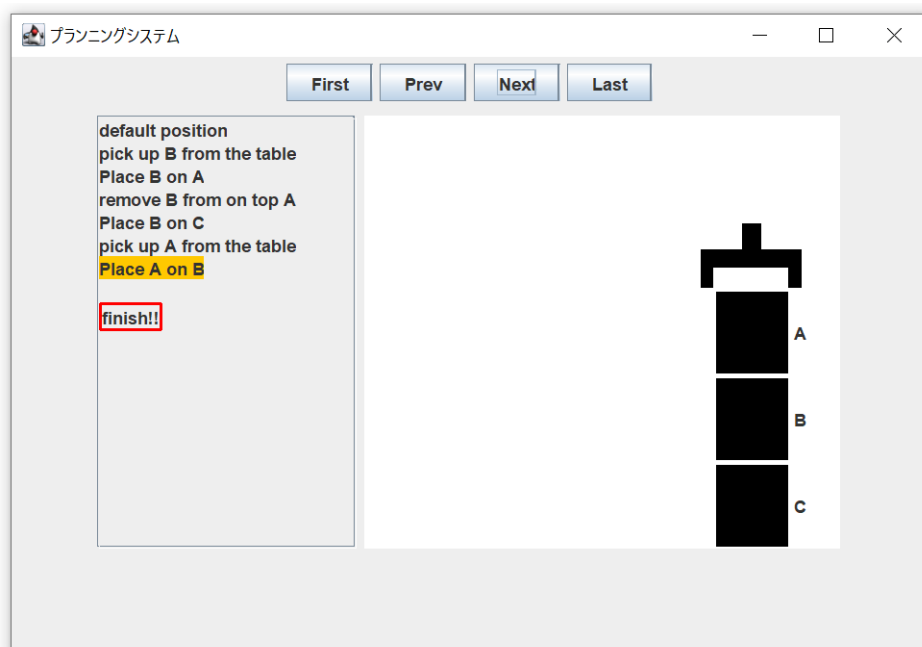


図 4 ブロック操作の過程 3

さらに，First ボタンを押すと一番初めの設定画面へ，Last ボタンを押すと一番最後の画面であるオペレータとプランニング結果を表す画面へと遷移する (図 5)。

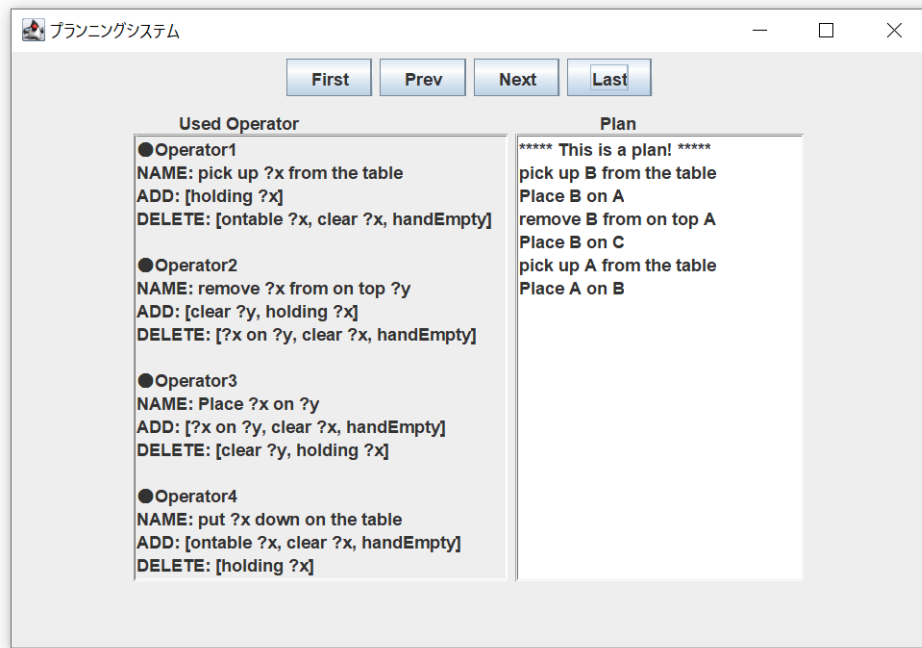


図 5 オペレータとプランニング結果の情報

次に、初期状態、目標状態、属性情報を変更してプランニングを行う。まず初期画面の左上に現在の属性情報があり、これを左下のラジオボタンを追加・削除・編集ボタンで変更する (図 6)。

The screenshot shows a window titled 'プランニングシステム' (Planning System). At the top, there are four buttons: 'First', 'Prev', 'Next', and 'Last'. The main area is divided into several sections. On the left, under 'Determine Attribution', there are three columns: 'Name', 'Color', and 'Shape'. The 'Name' column has a list with items A, B, C, and D, where D is selected. Below this is a 'new Name' input field and three buttons: '追加' (Add), '削除' (Delete), and '編集' (Edit). The 'Color' column has a list with Red, Green, Default, and Yellow. The 'Shape' column has a list with pyramid, trapezoid, default, and box. In the center, there are labels 'is' and '&'. To the right, there are two text areas: 'SetInitial' containing 'clear A', 'clear B', 'clear C', 'ontable A', 'ontable B', 'ontable C', and 'handEmpty'; and 'SetGoal' containing 'B on C' and 'A on B'. At the bottom center, there is a 'Planning' button.

図 6 属性情報の変更

また、ブロック追加時には左下にある 'new Name' 欄に新規ブロックの名前を入力する必要があるが、無記入で追加をした場合や既にある名前を入力した際にはエラーメッセージがダイアログウィンドウで表示されるようになっている (図 7, 図 8)。編集・削除ボタンを押した際に属性リストの名前を選択していなかった場合にも同様にエラーメッセージが表示される。

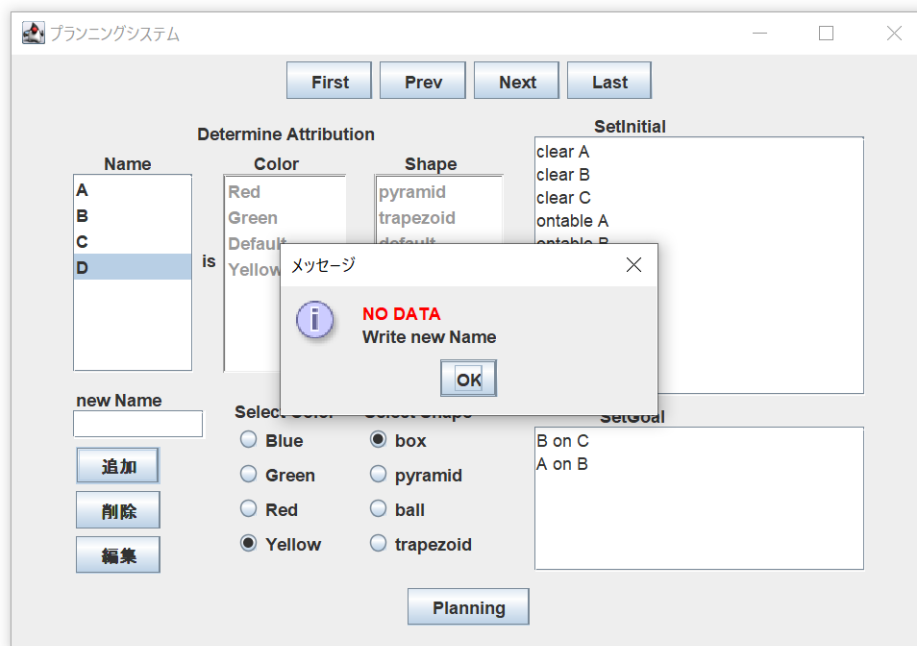


図 7 エラーメッセージの表示 1

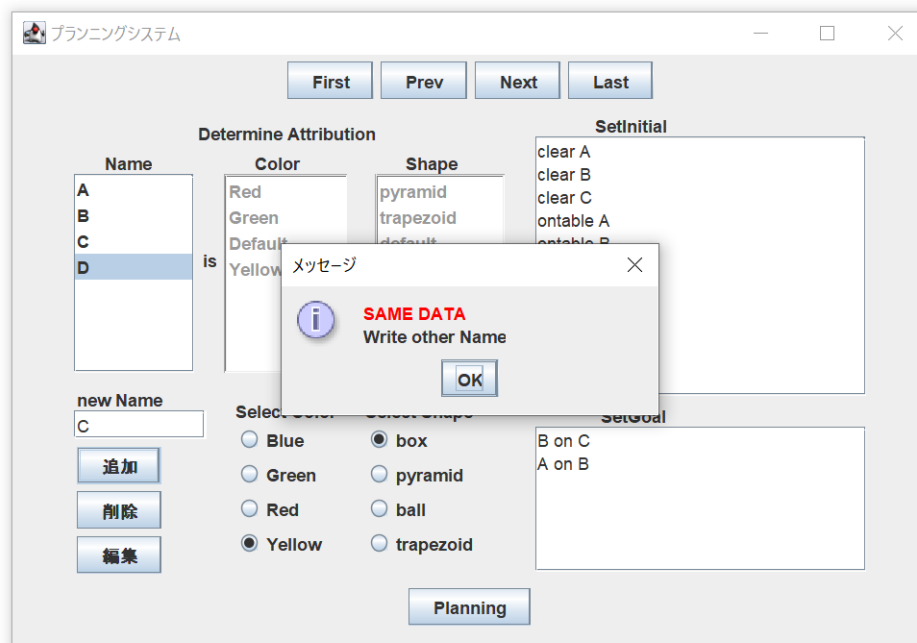


図 8 エラーメッセージの表示 2

初期状態と目標状態の変更は初期画面の右側のテキストボックスに記述する形で行う。このとき、範囲を超えた記述をする場合はスクロールバーが表示される (図 9)。また、属性情報変更時と同様に初期状態と目標状態に何も記入せず Planning ボタンを押すとエラーメッセージが表示される。

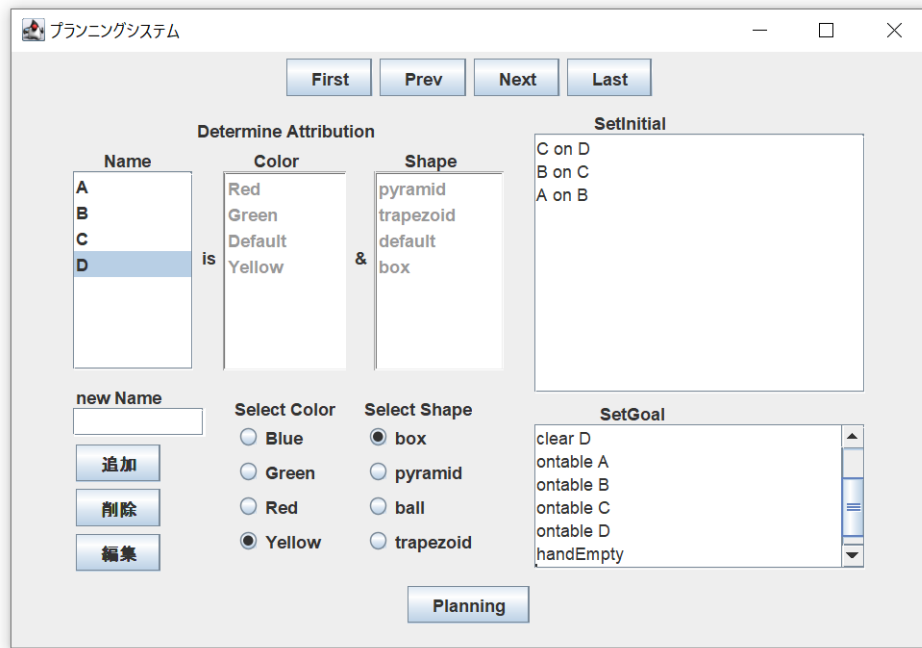


図 9 初期状態と目標状態の変更

変更を行った状態で Planning ボタンを押すと、定義した状態を基にプランニングが行われる。すると、探索が完了したことを伝えるメッセージが表示され、ブロック操作の過程の画面が新しく得られた結果が表示される。今回は例としてブロックを 4 つに増やし、属性情報を変えた (図 10, 図 11, 図 12)。

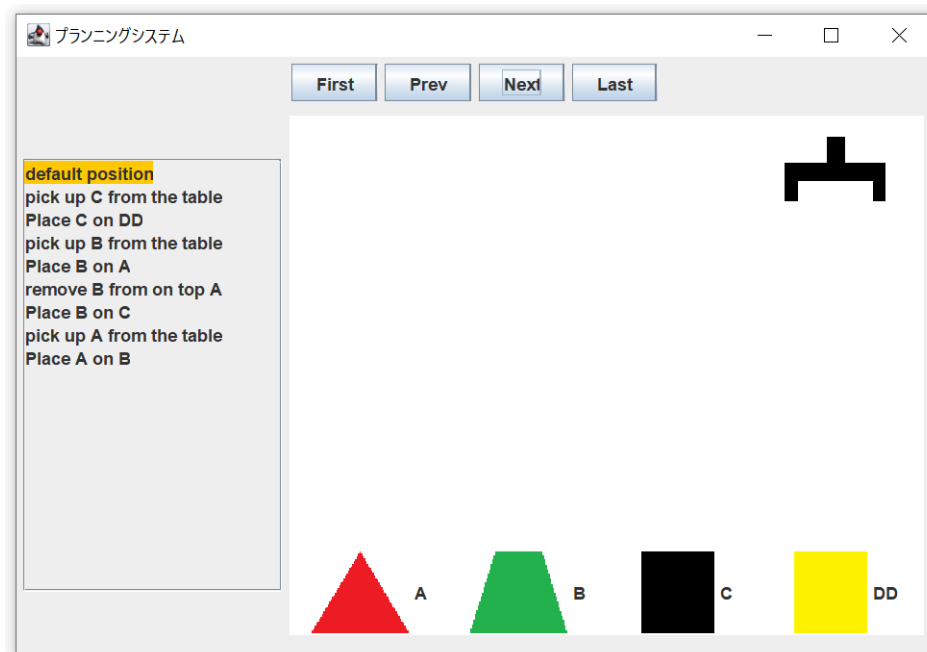


図 10 再実行時のブロック操作の過程 1

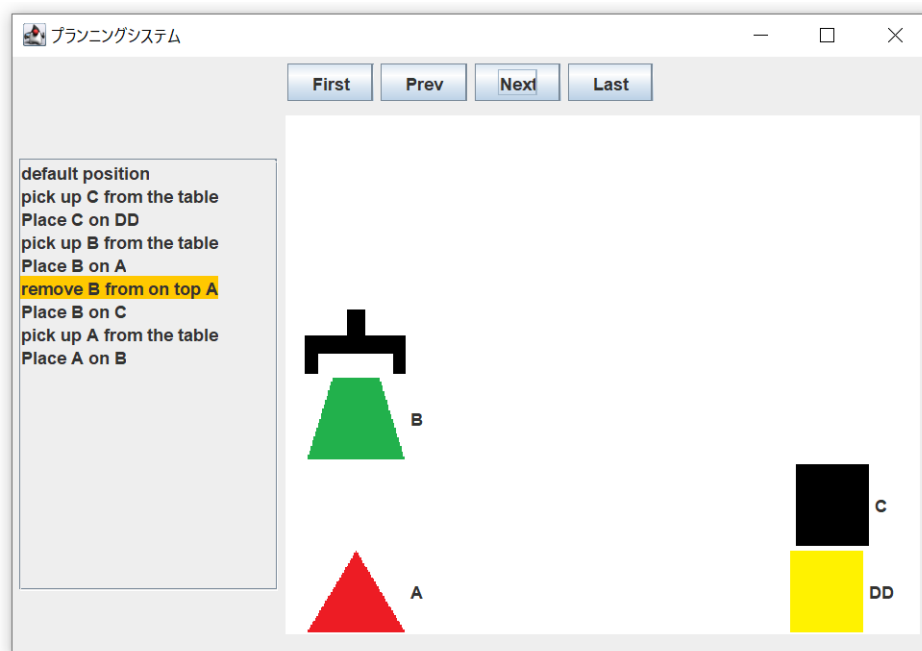


図 11 再実行時のブロック操作の過程 2

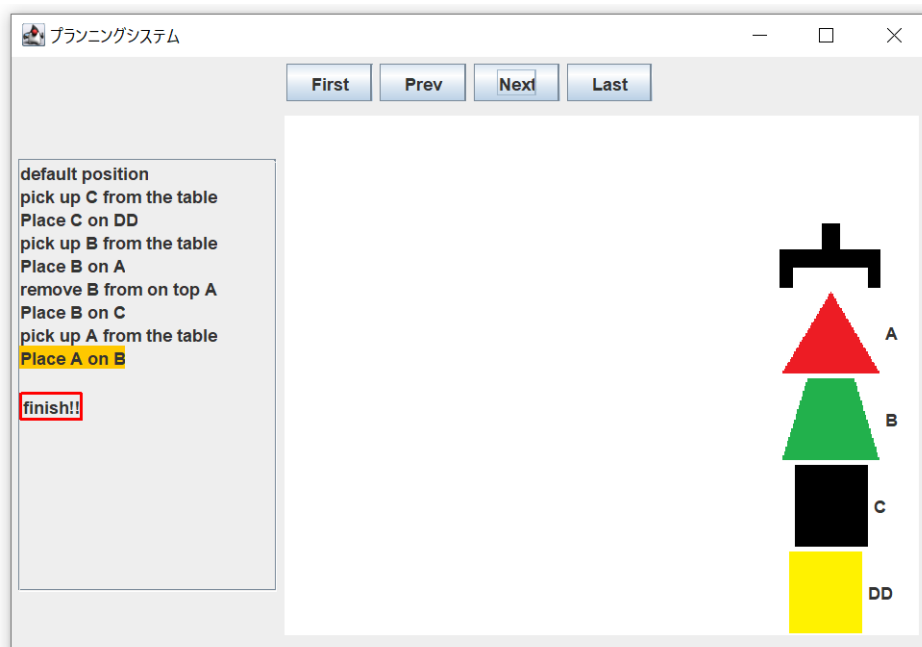


図 12 再実行時のブロック操作の過程 3

またブロックを積み上げる動作だけでなく、ブロックをおろす場合もグラフィカルな可視化を行うことが出来る (図 13, 図 14).

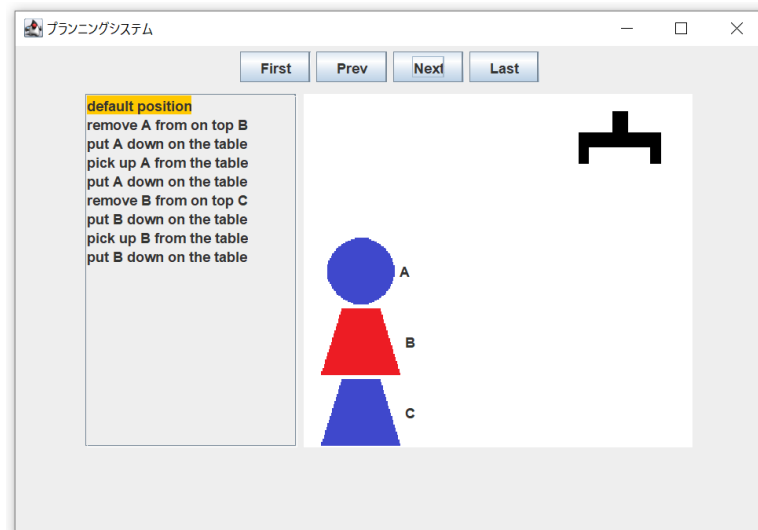


図 13 積み下ろしのブロック操作の過程 1

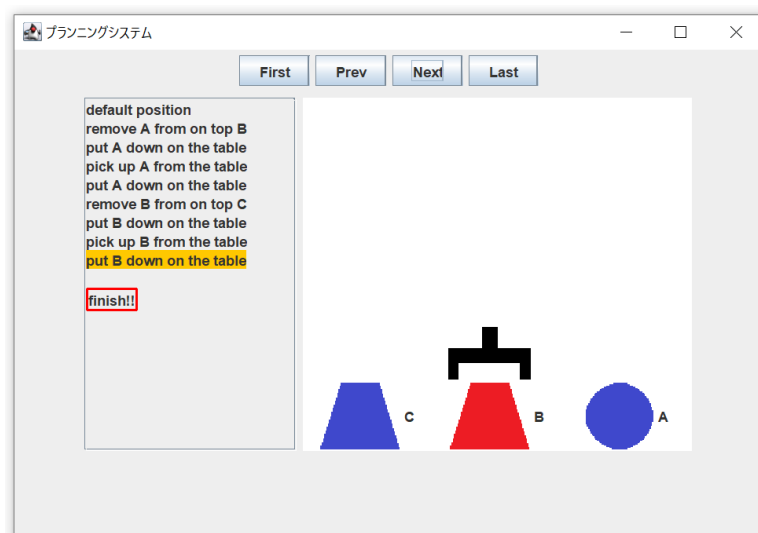


図 14 積み下ろしのブロック操作の過程 2

次に，初期状態と目標状態が同一であった場合の動作を示す．初期状態を示す画面の後に以下の表示が現れ，既に目標状態にあることが示される (図 15)．

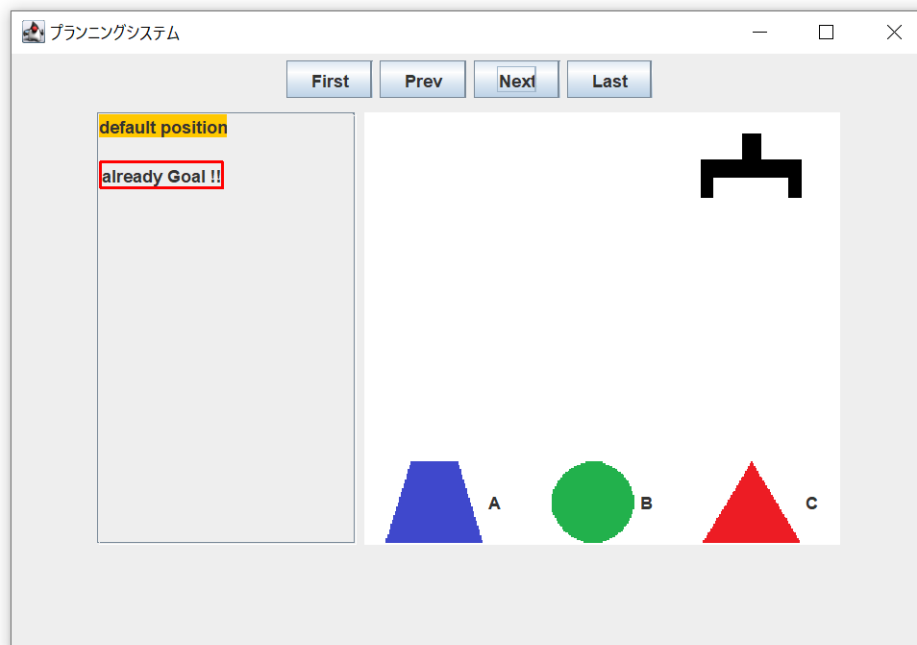


図 15 初期状態と目標状態が同一の場合

さらに，禁止制約によってプランニングが行えなかった場合の動作を示す．Planning ボタンを押すと，プランニングが行えなかったことを示すエラーメッセージが表示され，ユーザーが定めた属性情報・禁止制約・ユーザーが定めた目標状態が表示される (図 16, 図 17)．

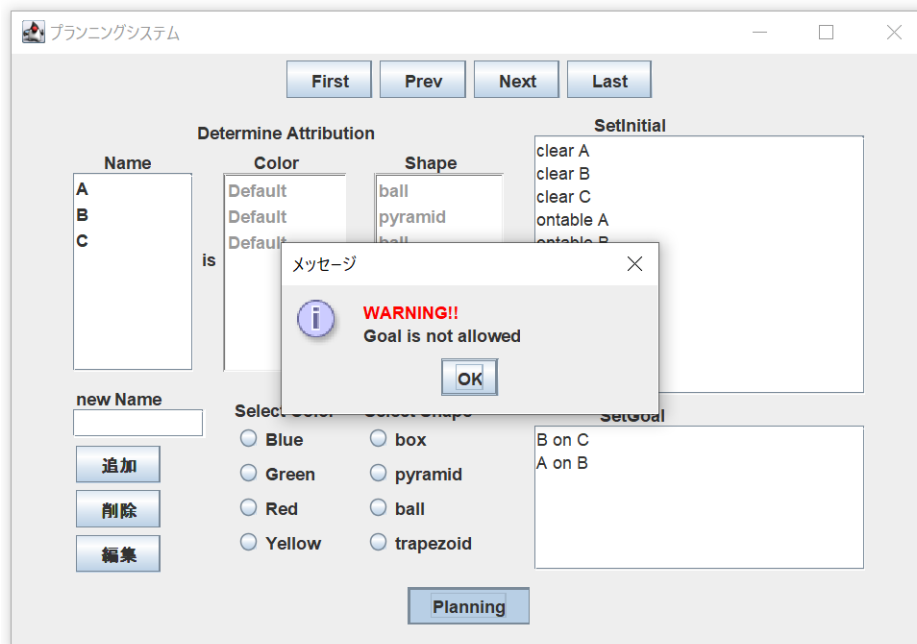


図 16 禁止制約を示すエラーメッセージの表示

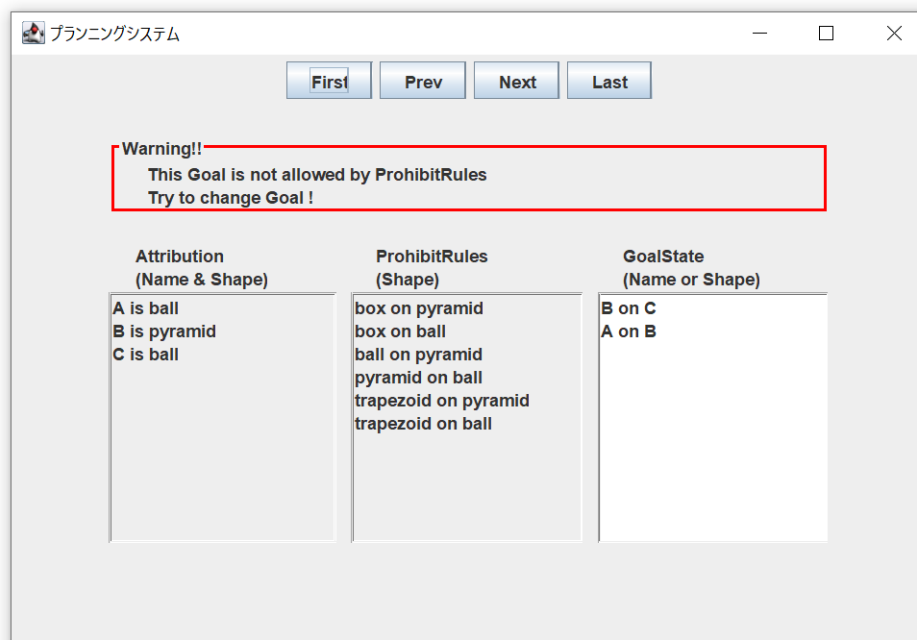


図 17 属性情報・禁止制約・目標状態の表示

(青山担当)

Presenter.java の main メソッドを実行したところ，以下のような結果が得られ，値が正しく取得できていることが分かる．

ソースコード 14 Presenter の実行

```
1 ...
2 ***** This is a plan! *****
3 pick up B from the table
4 ...
5
6 -----on Presenter-----
7 [clear blue, clear green, clear red, ontable ball, ontable trapezoid,
   ontable box, ontable pyramid, handEmpty]
8 [trapezoid on box, ball on trapezoid]
9 pick up ?x from the table
10 {?x=B}
```

3.4 考察

(湯浅担当)

今回は新しい知識について学ぶ必要があったため，まず初めにプログラムを二週間という期間の中でどのように実装するか計画を出来るだけ細かく立てた．これによっていつまでに何を終わらせておけば良いかが明確になり，スケジュール調整がし易くなったため，初めに分かる範囲で詳細な計画を立てることが重要であると分かった．

今回は状態遷移をグラフィカルに可視化するため，どのようにそれを描画するかがまず大きな課題であった．初めは一枚のフレームに状態遷移を全て表示することを考えていたが，それでは状態遷移が何回行われるか分からない今回の課題では表示方法に問題がある．そこで異なる方法を検討した．GUI について調べる中で，CardLayout を用いればカードの枚数を増やしていくことで状態の遷移回数の変化にも対応できると分かり，この実装でプログラムを考えた．

次に描画をどのように行うかについて考えた．GUI 上での図形の描画は座標指定で行う必要がある．しかしこれではレイアウトマネージャーを無効にする必要がある．この場合描画を全て自身の座標管理で行わなければならないため，描画の難度が高いと考えた．

そこで今回は図形を画像として予め作成しておき、これを ImageIcon として取り込み、BoxLayout で等分に分割した座標空間に対して割り当てていくことで実装を行うことを考えた。BoxLayout を用いることで、空間の行と列を指定した数に等分に分割し、それぞれに画像を割り当てることが出来る。この方法を用いることで、座標指定をすることなく図形を配置でき、比較的容易に状態遷移を描画出来ると考えた。

しかしこの方法を用いると、取り込んだ図に対して操作を行うことは出来ないことが分かった。そのため、画像の上に重なるように図形の名称 (A,B,C 等) を入れることや、取り込んだ画像の色を上書きすることが出来ないため、今回は図形の横に名称を記載し、画像は色違いのものをはじめから用意しておくことでこの問題を解決した。

予め図形を作成しておくことから、図形や色の任意指定も出来なくなってしまった。しかし座標指定を行う場合も図形の任意指定はできないと考えられるため、図形の任意指定を行えるようにするためには異なる方法を検討する必要があると感じた。

また、表示範囲は図形の大きさと個数から定めることで、ブロックの個数の変化に対応できるようにした。この方法を用いることで、表示する個数が変化しても描画内容に大きな変化が起きないようにした。しかし初めに設定したフレームの大きさの関係で、5 個以上のブロックの描画を行うと規定のフレームサイズを描画が超えてしまう。これにより、フレームをユーザー自身が大きくする必要が出てしまった。これについてはスクロールバーをつけることで対応しようと考えたが、実装が難しく実現できなかった。そのため描画は正しく行えたものの、表示方法に課題が残る結果となった。同時に、初めに作成するフレームのサイズも考えて決める必要があると分かった。

さらに属性情報の更新方法をどのようにするか考えた。属性情報では、名前・色・形を連動させ変更しなければならない。加えて任意での色や形の設定が出来ないため、これをラジオボタンで操作することとした。ここで項目を選択して編集を行えるとより使いやすくなると考え、JList を用いることを考えた。さらに名前・色・形をそれぞれ別の JList で表示することで、見やすい表示を心掛けた。

しかし名前・色・形をそれぞれ JList で管理すると、それぞれの項目を選択出来てしまうため、ユーザーが現在どの項目を編集しようとしているのかが分かりにくくなってしまった。これを解決するため色と形の JList は、プログラム側で編集を行う時以外は無効にすることで、ユーザーが操作できないようにすることを考えた。

また今回はユーザー視点の操作について考えた実装を心掛けた。

まず、ユーザーが任意で記述できる初期状態や目標状態・属性情報は記述スペースを超えた場合はスクロールバーが表示されるようにした。また出力される経路についても、表示画面よりも長くなってしまった場合はスクロールバーでの表示が出来るようにした。さらに状態遷移の描画を行う際の経路の表示では、自身がどの状態にあるのかを分かりやすくするため、現在の状態を表す一文に背景色としてオレンジ色を付けた。

経路が長くなってしまった場合はスクロールバーが表示されるものの、そのままでは常にスクロールバーが一番上にある状態でページが作成される。これでは表示範囲を超えた状態遷移のページでは、自身の現在の状態を確認するためにはスクロールバーを毎回下げる必要がある。これではユーザーにとって使い難い GUI だと考え、常に現在の状態が見える位置に来るように表示する座標位置を順に下げ、ユーザーがスクロールバーを動かさずに現在の状態を確認できるよう実装を行った。

さらに禁止されている入力を行った場合などはメッセージを表示することで、何が問題でどのようにすれば良いのかを分かりやすく示せるようにした。

(青山担当)

今回、Presenter 側では特にやることがなく、GUI 側の負担が大きいように感じた。しかし、GUI 制作経験のある身として、可能な限り GUI が扱いやすいような Presenter となるよう心がけた。例えば、プランの過程の渡し方は、Operator を少し書き加えて実現したことで、変数束縛も簡単に一緒に取得できるようにしたことや、String 型でも過程を受け取れるようにしたこと、GUI 側で行うべき処理を減らせるようにしたことである。

しかし、今回の課題も前回同様最初はどのような形で値を渡すかで悩むこととなった。前回よりも早く意識のすり合わせと進捗の説明を担当者間で共有できたため、今回はよりスムーズに実装に漕ぎ着けたのだと考えられる。

また、GUI と Presenter の分担を交代してやってみることで、GUI の面ではレイアウトの仕方や表示のための工夫、Presenter の面では Planner 等の元のプログラムから途中のデータを引き継いだり引き渡すことの難しさやすべき考慮等を学ぶことができた。このように、ある程度経験がついた状態で役割を大胆に変えてみることは、お互いのプログラミングスキル向上を促進し合える一つの手だと考えられる。

4 発展課題 5-5

ブロックワールド内における物理的制約条件をルールとして表現せよ。例えば、三角錐（pyramid）の上には他のブロックを乗せられない等、その世界における物理的な制約を実現せよ。

4.1 手法

ブロックワールド内における物理的制約条件（以下、禁止制約と表現する）をルールとして表現し、プランニングに反映するために、課題 5-3 で作成した属性に関する処理を扱うクラス `Attributions` 内に禁止制約のルールを保持させる方法を考案した。例えば、”box on pyramid”（三角錐の上に正四面体がある）のようなルールが物理法則に反する禁止制約に当たる。

これらの禁止制約のルール集合を、課題 5-3 で結びつけた「属性とブロックの関係」をもとに、属性を用いた表現からブロックによる表現に置き換える。

以上により、ゴールリストやオペレーター適用後の状態とブロックによって表現される禁止制約を比較することで、物理法則に反する状態を削除することができる。また、ゴールリストに物理法則に反するものが含まれる場合には、ゴール不成立とする。

以下に、今回実装した禁止制約の例を示す。

4.2 実装

まず、属性を扱う `Attributions` クラス内に `ArrayList<String>` 型の禁止制約のルール集合 `prohibitRules` を保持させる。この `prohibitRules` はコンストラクタから呼び出されるメソッド `addProhibitRules()` により禁止制約のルールが追加される。

加えて、属性によって表現された禁止制約のルールをブロックによる表現に置き換えたルール集合 `prohibitBlockStates` も定義する。`prohibitRules` から `prohibitBlockStates` に変換を行うに当たっては、課題 5-3 で作成したメソッド `editStatementList` を用いる。

次に、状態と禁止制約を比較するメソッド `checkProhibitBlockState(String state)` を実装した。このメソッドを以下に示す。

物理制約によって禁止される制約の例

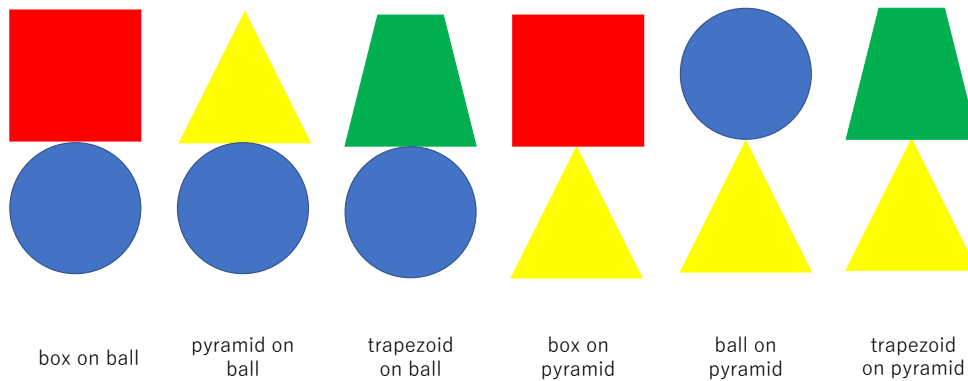


図 18 物理制約によって禁止される制約の例

ソースコード 15 checkProhibitBlockState メソッドの実装

```
1 private Boolean checkProhibitBlockState(String state) {
2     for(String prohibitBlockState: prohibitBlockStates) {
3         if(prohibitBlockState.equals(state)) {
4             System.out.println("【Warning!:状態"+
5                 state+"は禁止制約です!!】");
6             return false;
7         }
8     }
9     return true;
}
```

27 行目に渡って、全ての禁止制約のルール集合の要素と仮引数で受け取った状態を比較する処理を行なっている。ここで、一つでも禁止制約と一致すると、false が返却される。反対に、全ての禁止制約と一致しなかった場合には、true が返却される。

続いて、この checkProhibitBlockState メソッドを利用する checkStates(ArrayList<String> states) メソッドの説明を行う。このメソッドについても以下に示す。

ソースコード 16 checkStates メソッドの実装

```
1 ArrayList<String> checkStates(ArrayList<String> states) {
```

```

2          ArrayList<String> checkedStates = new ArrayList<String>
              >();
3          for(String state: states) {
4              if(checkProhibitBlockState(state)) {
5                  checkedStates.add(state);
6              }
7          }
8          return checkedStates;
9      }

```

3 7 行目に渡って、仮引数で受け取った状態集合の各要素を checkProhibitBlockState の引数として渡すことで、禁止制約のチェックを行なっている。結果として true が帰ってきた場合にはチェック済みの状態集合 checkedStates に格納し、false の場合には格納されない。

全ての要素のチェックが終わると、checkedStates を返却する。

checkedStates メソッドを、変化直後の状態集合に対して使用することによって、物理法則に反する状態を削除することができる。以上の実装により、禁止制約のブロックワールドへの反映を実現した。

4.3 実行例

以下に、CUI 上での実行結果を示す。ただし、コンストラクタはデフォルトコンストラクタを使用するものとする。

ソースコード 17 禁止制約によりゴールが成立しなくなった場合

```

1      ~/Programming2/Work5
2      ●java Planner 【 feature/prohibit-rules 】
3      ##### Add prohibitRule #####
4      ***** ProhibitRule:box on pyramid *****
5      ***** ProhibitRule:box on ball *****
6      ***** ProhibitRule:ball on pyramid *****
7      ***** ProhibitRule:pyramid on ball *****
8      ***** ProhibitRule:trapezoid on pyramid *****
9      ***** ProhibitRule:trapezoid on ball *****
10     +++++++ EditStatement +++++++
11     box on pyramid ==> A on B
12     box on ball ==> A on C

```

```

13    ball on pyramid =====> C on B
14    pyramid on ball =====> B on C
15    trapezoid on pyramid =====> trapezoid on B
16    trapezoid on ball =====> trapezoid on C
17    ===== goal:green on ball =====
18    ===== goal:blue on pyramid =====
19    ++++++ EditStatement ++++++
20    green on ball =====> B on C
21    blue on pyramid =====> A on B
22    【Warning!:状態B on Cは禁止制約です!!】
23    【Warning!:状態A on Bは禁止制約です!!】
24    ----- initInitialState:clear blue -----
25    ----- initInitialState:clear green -----
26    ----- initInitialState:clear red -----
27    ----- initInitialState:ontable box -----
28    ----- initInitialState:ontable pyramid -----
29    ----- initInitialState:ontable ball -----
30    ----- initInitialState:handEmpty -----
31    ++++++ EditStatement ++++++
32    clear blue =====> clear A
33    clear green =====> clear B
34    clear red =====> clear C
35    ontable box =====> ontable A
36    ontable pyramid =====> ontable B
37    ontable ball =====> ontable C
38    handEmpty =====> handEmpty
39    禁止制約によってゴールが成立しなくなりました

```

はじめに,39行目で禁止制約のルールが `prohibitRules` に追加されている。次に,1016行目で `editStatementList` によって属性による表現からブロックによる表現に変換されている。17,18行目で属性表現によって指定されたゴールが1921行目でブロックによる表現に変換されている。

そして,22,23行目で禁止制約がゴールに使用されていることが検出されている。詳しく見てみると,22行目の検出は14行目と20行目の変換結果が一致することによるものであり,23行目の検出は11行目と21行目の変換結果が一致することによるものである。これらによって,ゴールリストに物理法則に反するものが含まれていることが判明したため,39行目においてメッセージが表示され,プランニングが終了している。

4.4 考察

課題 5-3 の実装をもとに禁止制約をブロックワールドに反映する実装を行なった。特に, `editStatementList` のように再利用することのできるメソッドを作成したことは大きなプラスとなった。また, 課題 5-3 で取り決めた, 属性をブロック名に置き換えてからプランニングを行う方針は, 課題 5-5 を進めるに当たって大きな助けにもなった。具体的には, "A is red", "A is pyramid", "B is blue", "B is ball" のようなルールのもと, "pyramid on ball" のような禁止制約があるとする。このとき, "red on blue" が状態に含まれるとき, 属性表現のままプランニングを行うと見逃してしまうことになるが, 先に状態と禁止制約の属性表現をブロック名に変換して, "A on B" とすることによって, 物理法則に反する状態であると判定することができる。

したがって, 課題 5-3 で取り決めた方針によって, 課題 5-5 のプログラムの質を向上させることができたと考えている。

以上より, より良い実装を目指すに当たって, あらかじめ頭の中でプログラムの構造や処理の手順を考えておくことは非常に効果的であると感じた。

5 発展課題 5-6

ユーザが自然言語（日本語や英語など）の命令文によってブロックを操作したり, 初期状態／目標状態を変更したりできるようにせよ。なお, 命令文の動詞や語尾を 1 つの表現に決め打ちするのではなく, 多様な表現を許容できることが望ましい。

6 発展課題 5-7

3次元空間（実世界）の物理的な挙動を考慮したブロックワールドにおけるプランニングを実現せよ。なお, 物理エンジン等を利用する場合, Java 以外の言語のフレームワークを使って実現しても構わない。

私の担当箇所は, 発展課題 5-7 におけるプランニングの, Unity を用いた実装である。

6.1 手法

3次元空間の物理的な挙動を考慮したブロックワールドにおけるプランニングを実現するにあたり、以下のような方針を立てた。

1. 空間やプランに関するオブジェクトを生成する。
2. プランニングを行えるようにスクリプトを作成する。

1. に関して、物体を生成し、コンポーネントを付与することで物理的な挙動を行えるようにした。また、3種類のプランを実装することで、物理制約の確認を容易に行えるような仕様とした。

2. に関して、C#スクリプトを用いてキーボードやマウスの入力を受け付けられるように実装した。Master オブジェクトを作ってそこにアタッチすることで、それらの操作を一括的に管理できるような仕様とした。

6.2 実装

Main シーンに含まれるオブジェクトには以下のものが含まれる。

Main Camera 主カメラに関するオブジェクト。Room 全体をやや見下ろし気味に映す。

Directional Light オブジェクト全体を照らす照明。

Master スクリプトをアタッチするための空オブジェクト。

Room 6個の Plane オブジェクトを子に持つ、立方体の部屋を構成するオブジェクト。

Cube 直方体のブロックを生成するプレハブ。

Sphere 球のブロックを生成するプレハブ。

Torus 円環体のブロックを生成するプレハブ。

C#スクリプトでは以下のものが実装されている。

Clicked クリックされたオブジェクトにフォーカスを当てるスクリプト。

Operationg Clicked でフォーカスされたオブジェクトにキーボード入力を反映するスクリプト。

Generator キーボード入力に合わせてブロックを生成するスクリプト。

Destroyer クリックされたオブジェクトを削除するためのスクリプト。

6.2.1 空間やプランに関するオブジェクトを生成する.

まず、オブジェクトの受け皿となる部屋の実装を行った。オブジェクトには Plane を用い、コンポーネント Mesh Collider をアタッチすることで、ブロックとの衝突判定を実現した。Plane オブジェクトの特徴として、裏面からはオブジェクトが透明に見えることが挙げられる。これにより、外部から可視化した状態のまま密閉空間を実現できた。

次に、Cube や Sphere のオブジェクトを用いてプレハブの元となる直方体と球のブロックを実装した。直方体には Box Collider、球には Sphere Collider をアタッチすることで衝突判定を実現し、更にいずれに対しても Rigidbody をアタッチすることで、質量や重力に関する挙動を実現し、より物理的な挙動を考慮したプランニングが実現できるようになった。また、Physic Material を作成し各ブロックの Collider にアタッチすることで、摩擦も考慮した挙動が実現できた。そしてこれらのオブジェクトはプレハブにすることで、同オブジェクトの複製が容易に行えるようになった。

次に、球ブロックの受け皿として最適だと思い、円環体のブロックの実装を試みた。しかし直方体や球とは異なり、プリミティブなゲームオブジェクトからでは実装は難しかったため、PackageManager に含まれる機能である、ProBuilder を用いて実装を試みた。これにより円環体の生成はできたものの、形が複雑であるため、衝突判定には Mesh Collider を用いる必要があった。

しかし、Mesh Collider は他のオブジェクトとの衝突判定を無視したり、設定を弄って衝突判定が行われるようになっても内側の穴が穴としての判定がなされなかったりと、その挙動に悩まされることとなった。

そこで、円環体の衝突判定には Asset Store から SAColliderBuilder をインポートし、それを用いることにした。SAColliderBuilder のコンポーネント SAMeshColliderBuilder から Split Polygon Normal をオンにし、Shape Type を Mesh に、Mesh Type を Convex Hull にすることで、Rigidbody としての挙動を残したまま、穴を含む正確な衝突判定を実現できた。

しかしこれをプレハブ化すると、SAColliderBuilder が正常な挙動がなされず、再度床をすり抜けるようになってしまった。設定を弄っていると”Non-convex MeshCollider with non-kinematic Rigidbody is no longer supported since Unity 5.” といったエラーが出たことから、MeshCollider と Rigidbody は相性が悪いと考えられたため、そもそもの円環体の当たり判定の仕様を変えることにした。

判定には Sphere Collider を組み合わせて行い、円環体の子オブジェクトとして衝突用のオブジェクトを作成した。

SAColliderBuilder で実装した改良前の衝突判定と，Sphere Collider を組み合わせて実装した改良後の衝突判定を図 19 と図 20 に示す．

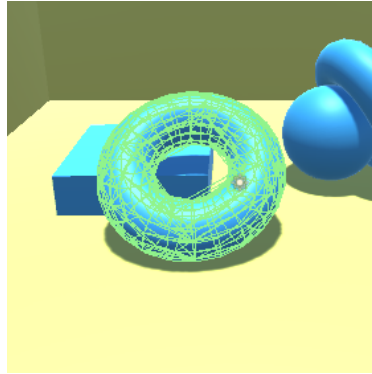


図 19 改良前

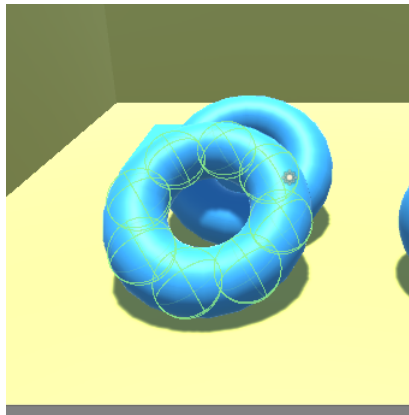


図 20 改良後

このように，円環体の当たり判定が簡略されながらも充分に実現されていることが分かる．Sphere Collider で実現したことにより，プレハブ化しても Rigidbody との併用が問題なく行うことができた．

6.2.2 プランニングを行えるようにスクリプトを作成する．

Clicked.cs について，クリックされたときにまずフォーカス対象のオブジェクトを null にしている．次にクリックしたオブジェクトの取得を Raycast を用いて行うが，ここで操作対象外である床等のオブジェクトをフォーカス対象外にする必要があった．これはフォーカス対象のブロックに Plan というタグを統一して付けることで，CompareTag か

らその判定を行えるようにした。また、クリック時に円環体の当たり判定である子オブジェクトを検出したとき、親オブジェクトを取り扱うように、root フィールドを用いてその値を取得した。

クリックしたときに実行される Update メソッドは以下のとおりである。

ソースコード 18 Clicked クラスの Update メソッド

```
1    void Update()
2    {
3        if (Input.GetMouseButtonDown(0))
4        {
5            if(clickedGameObject != null)
6            {
7                clickedGameObject = null;
8            }
9
10           Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
11           RaycastHit hit = new RaycastHit();
12
13           if (Physics.Raycast(ray, out hit))
14           {
15               clickedGameObject = hit.collider.gameObject.transform.
16                   root.gameObject; // 親要素の取得
17               if (!clickedGameObject.CompareTag("Plan"))
18               {
19                   clickedGameObject = null; // Plane は対象外
20               }
21           }
22           Debug.Log(hit.collider.gameObject);
23       }
24   }
```

Operating.cs では、Clicked クラスで取得したオブジェクトの移動のために、GetKey メソッドを用いてキーボードの入力判定を行う。初めはスクリプト内で一時的に対象のオブジェクトの Rigidbody の isKinematic を true にして transform の position に Vector3 を足し合わせることで移動させていたが、それだと摩擦の判定が行われなかったり、衝突

判定を貫通して移動できてしまったりするという問題があった。これは isKinematic を true にしたことが原因であると分かった。そこで、position に足し合わせるのではなく AddForce で Vector3 を足し合わせるようにしたことで、isKinematic を false のまま物理的制約に準ずる物体の移動を実現することが出来た。

キーボード入力時に挙動を示す Operating クラスは以下の様になっている。

ソースコード 19 Operating クラス

```
1 public class Operating : MonoBehaviour
2 {
3     private Vector3 velocity;
4     private float moveSpeed = 1000.0f;
5
6     void Update()
7     {
8         velocity = Vector3.zero;
9         if (Input.GetKey(KeyCode.W))
10             velocity.z += 1;
11         if (Input.GetKey(KeyCode.A))
12             velocity.x -= 1;
13         if (Input.GetKey(KeyCode.S))
14             velocity.z -= 1;
15         if (Input.GetKey(KeyCode.D))
16             velocity.x += 1;
17         if (Input.GetKey(KeyCode.E))
18             velocity.y += 1;
19         if (Input.GetKey(KeyCode.Q))
20             velocity.y -= 1;
21         // 速度ベクトルの長さを 1秒でmoveSpeed だけ進むように調整します
22         velocity = velocity.normalized * moveSpeed * Time.deltaTime;
23
24         if (velocity.magnitude > 0)
25         {
26             // プレイヤーの位置 (transform.position)の更新
27             // 移動方向ベクトル (velocity)を足し込みます
28             GameObject go = Clicked.clickedGameObject;
29             if (go != null)
30             {
```

```
31             go.GetComponent<Rigidbody>().AddForce(velocity);  
32         }  
33     }  
34 }  
35 }
```

Generator.cs では、Resources クラスの Load メソッドからプレハブを取得し、Instantiate で生成するようにすることで、任意のタイミングで好きなだけブロックの生成ができるようにした。

Destroyer.cs では、Clicked.cs 同様にタグを用いて判定することで、ブロック以外のオブジェクトが誤って消えることのないようにした。

また、これらのスクリプトは各オブジェクトにアタッチせずとも、Master オブジェクトにさえアタッチすれば全体の挙動が管理できるような仕様としたことで、各オブジェクトに対する負荷を減らし、効率的なプログラムを実装することができた。

6.3 実行例

Block World Planning.exe を起動したところ，下図のような画面が得られる．

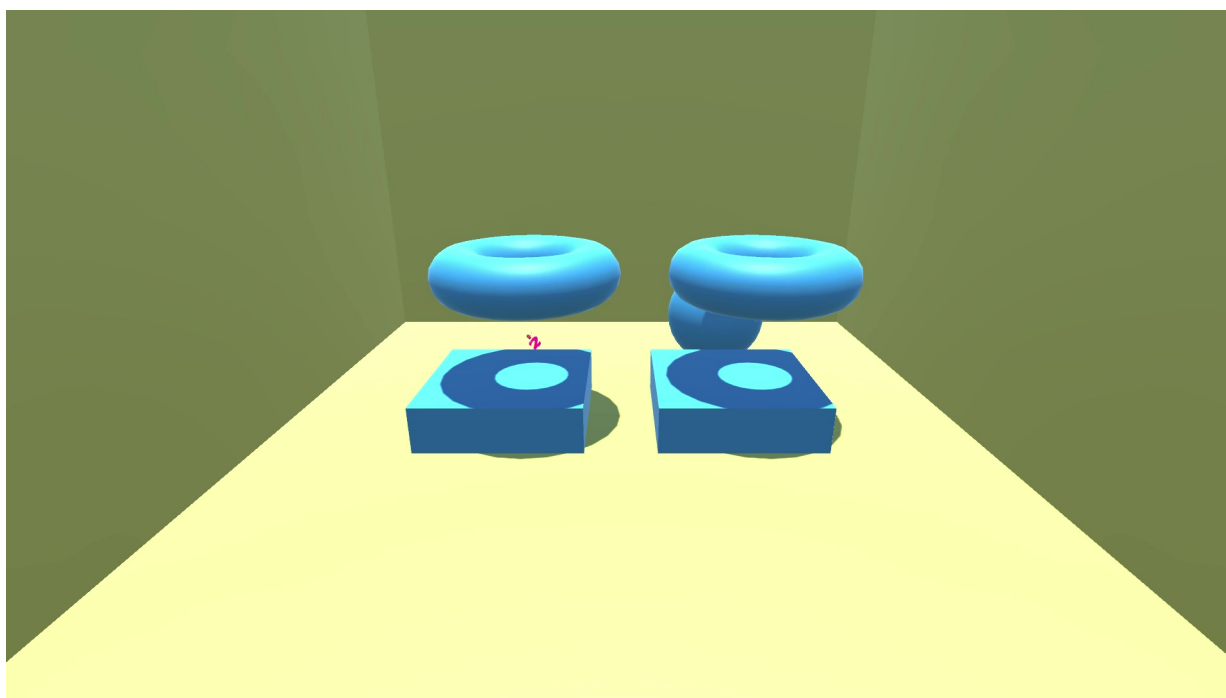


図 21 起動時の画面

右クリックでオブジェクトを削除でき，キー“1”で直方体，キー“2”で円環体，キー“3”で球のブロックを生成できる．これによって構成した下図をプランニングの初期状態とする．

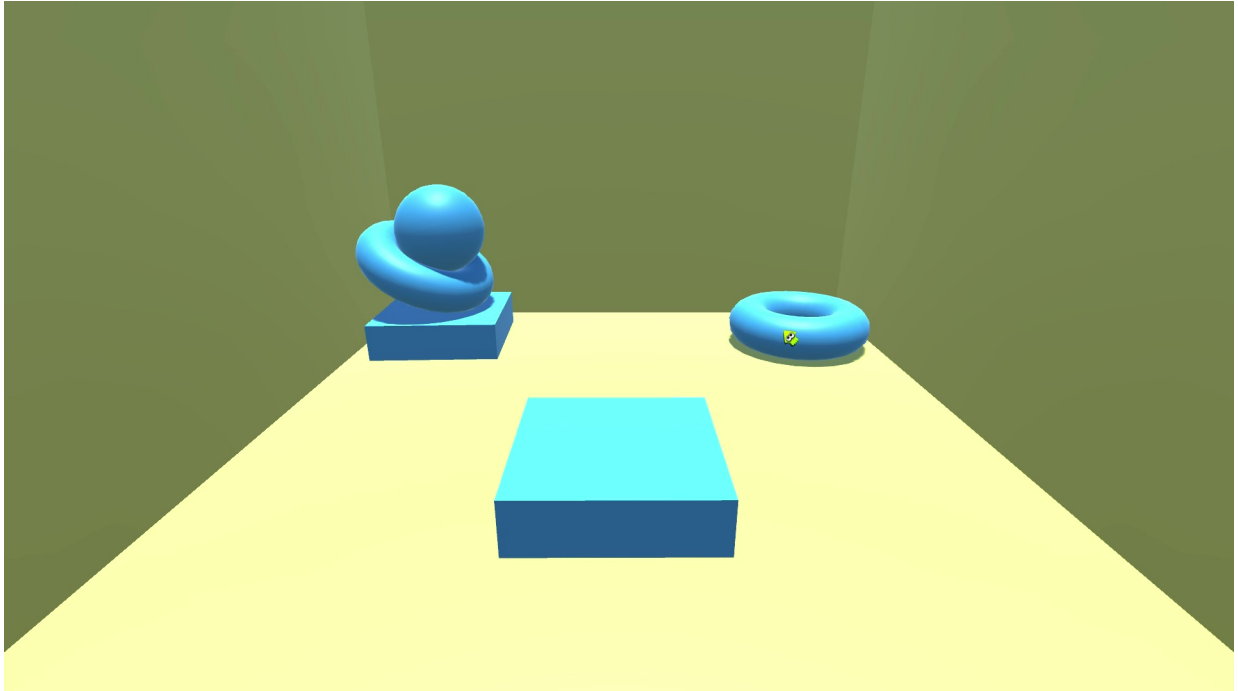


図 22 初期状態

左クリックで移動するオブジェクトを選択してから，WASD キーで上下左右，EQ キーで昇降の移動が行える．手前の直方体に乗せるために右奥の円環体を持ち上げた様子が下図のとおりである．

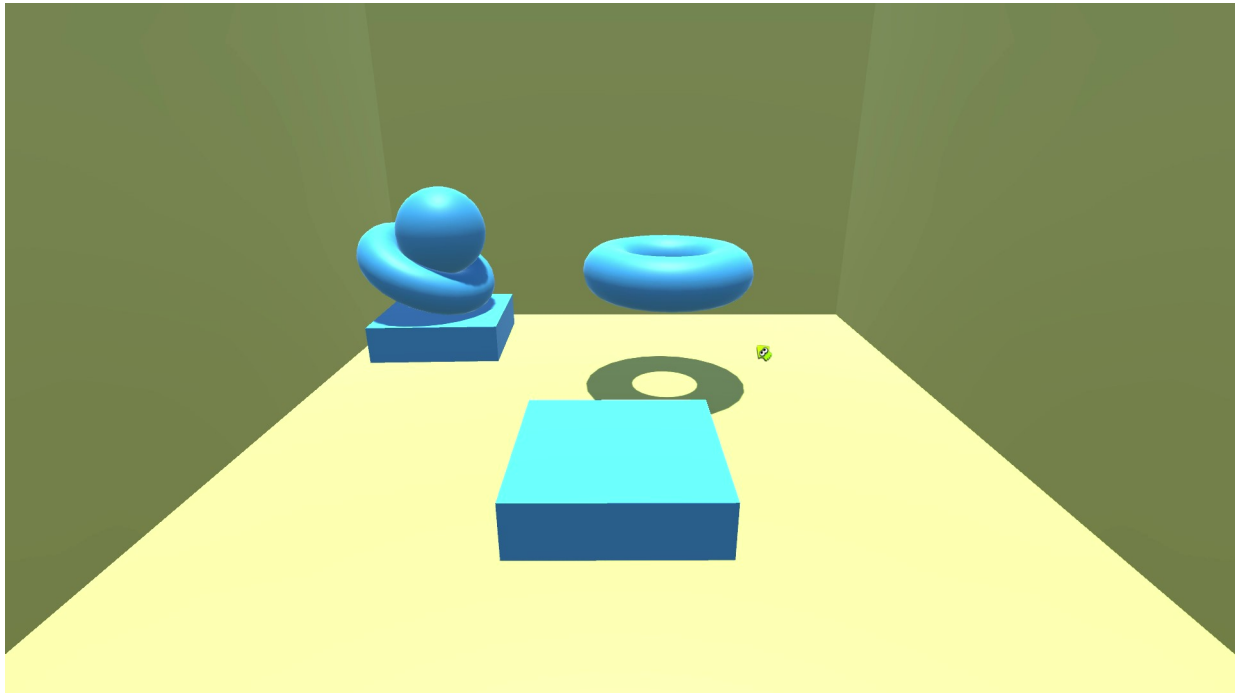


図 23 プランニング開始

そうして，物理的な挙動を考慮した上で円環体を直方体の上に乘せた様子が下図のとおりである．

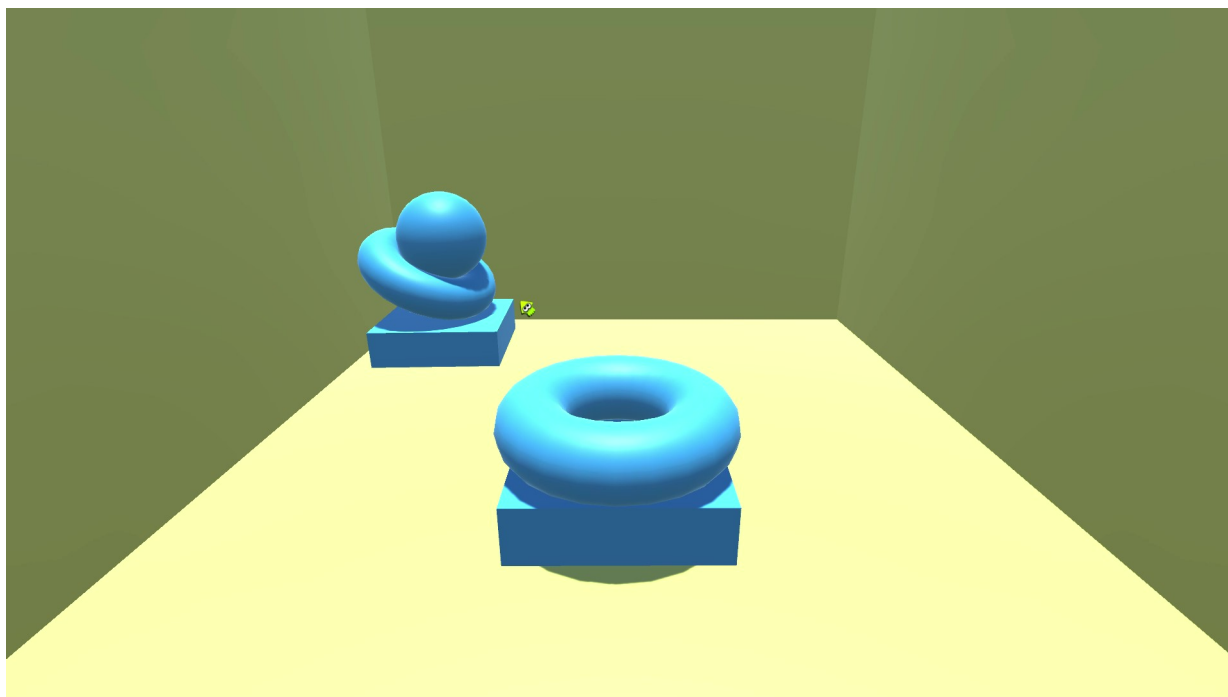


図 24 ステップ 1 完了

物理的な挙動を考慮して完了したプランニングが下図のとおりである.

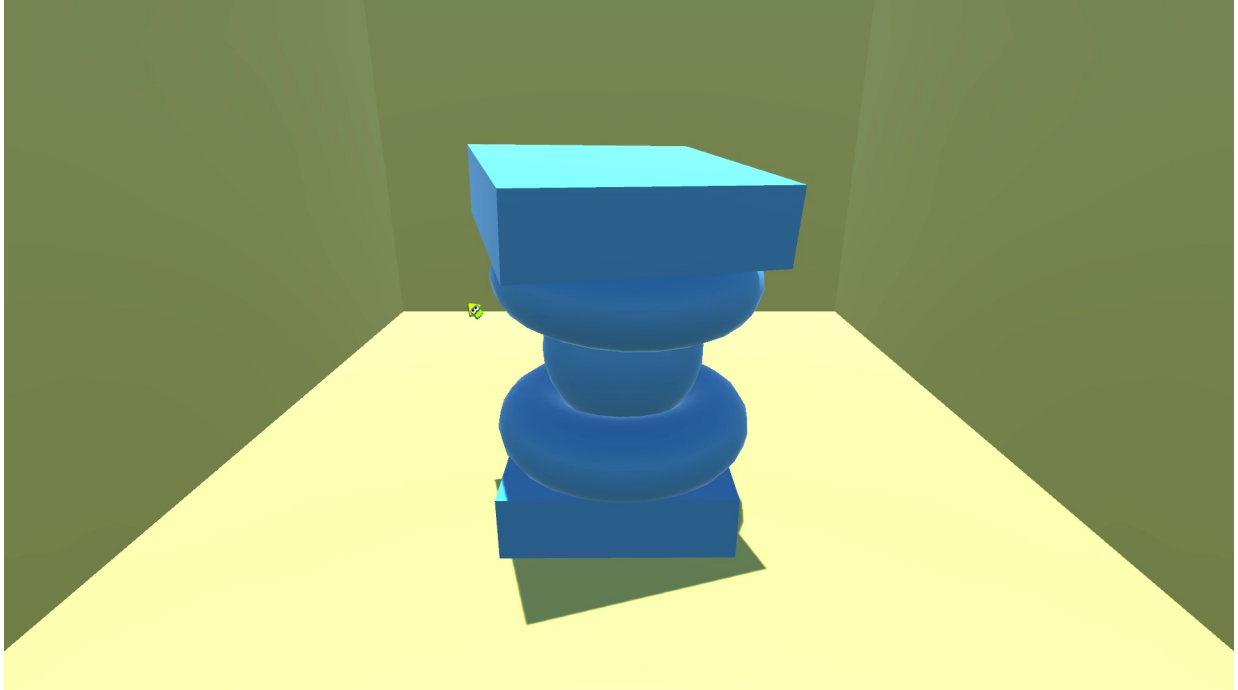


図 25 プランニング完了

6.4 考察

物理的な挙動を考慮したプランニングを実装するために Unity を用いてみて、衝突判定や重力の考慮などを付けるだけなら容易に出来たが、円環体の穴の当たり判定が正しく設定できないなどで数多く躓いた。この問題は SphereCollider を工夫して用いることで解決したが、このように、実現したいものをマイナーなアセットを探して実現するのではなく、今ある機能を工夫して用いて実現する、という考え方は大切にすべきだと思った。

厳密な当たり判定が必要であったわけではなく、円環体の穴に自然な感じで球をはめたいというのが元々の目的であったことに気づいたおかげで解決できたように、本当に実現したいものは何か、本当に必要なものは何かを立ち返って考え、問題を簡略化することが、プログラミングの1つのテクニックであると考えられる。

また、子オブジェクトという概念に悩まされたように、Unity も Java と同じ様に、ポインタが何を指しているかを考え、微妙な違いを見落とさないようにすることが重要であることを痛感した。Unity は Java よりも並列的な処理がしやすい一方で、カプセル化に関しては大雑把な印象を受けたため、パブリックな変数がどのように遷移しているのか、Debug.Log をこまめに用いて確認することが大切だと思った。

また、Unity を使っていてフリーズやクラッシュが起こって、データが飛んだことがあった。Java のプログラミングではそういうことが起こったことはなかったので、ゲームエンジンならではの重い処理の影響だと考えられる。このような普段と違うフレームワークを用いるときは、そのフレームワークに合わせてこまめにセーブをするといった対応や環境構築を行うことも大切だと考えられた。

また、作ったアプリをチームメンバにやってみてもらったところ、処理が重いという報告を受けた。このことを受け、フレームルートやポリゴンの制限を行い、自分とは違う環境でも動作しやすいアプリケーション作りを今後は意識してゆく必要があると思った。

7 発展課題 5-8

教科書 3.3 節のプランニング手法を応用できそうなブロック操作以外のタスクをグループで話し合い，新たなプランニング課題を自由に設定せよ．さらに，もし可能であれば，その自己設定課題を解くプランニングシステムを実装せよ．

参考文献

- [1] Java による知能プログラミング入門 ー著：新谷 虎松
- [2] Let's プログラミング Swing を使ってみよう, <https://www.javadrive.jp/tutorial/> (2019 年 12 月 8 日アクセス) .