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

2019年11月25日

提出物　work5

グループ　グループ8

メンバー

学生番号	氏名	貢献度比率
29114003	青山周平	20
29114060	後藤拓也	20
29114116	増田大輝	20
29114142	湯浅範子	20
29119016	小中祐希	20

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 5-4

> 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．

私の担当箇所は，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．

2.1　手法

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．

2.2　実装

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，擬似的な各種のゲッタやセッタ，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．

2.2.1　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．

2.2.2　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ったら 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1に示す．

ソースコード 1: 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           break;
18         }
19       }
20     }
21   }
```

2.3 実行例

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 2: 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}
```

2.4 考察

今回の Presenter 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で 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．

3 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．

3.1 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．

3.2 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．

3.2.1　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Pigidbody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，そもそも円環体の当たり判定の方法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持たせた．

SAColliderBuilder で実装した改良前の衝突判定と，Sphere Collider を組み合わせで実装した改良後の衝突判定を図 1 と図 2 に示す．

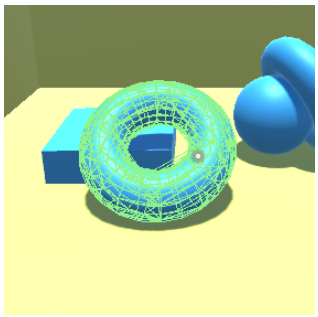

図 1: 改良前

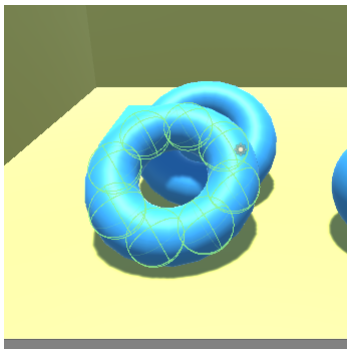

図 2: 改良後

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．

3.2.2　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，操作対象外の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，円環体の当たり判定である子オブジェクトをクリックで検出したとき，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 3: 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.root.gameObject;
                    // 親要素の取得
16              if (!clickedGameObject.CompareTag("Plan"))
17              {
18                  clickedGameObject = null; //
                        Plane は対象外
19              }
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った．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じた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 4: 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．

3.3 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．

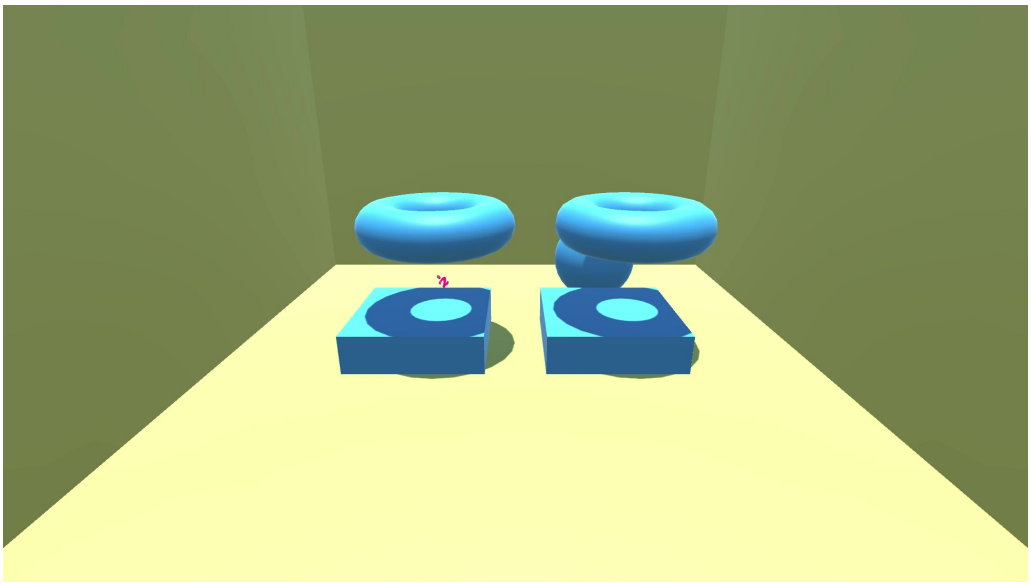

図 3: 起動時の画面

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．

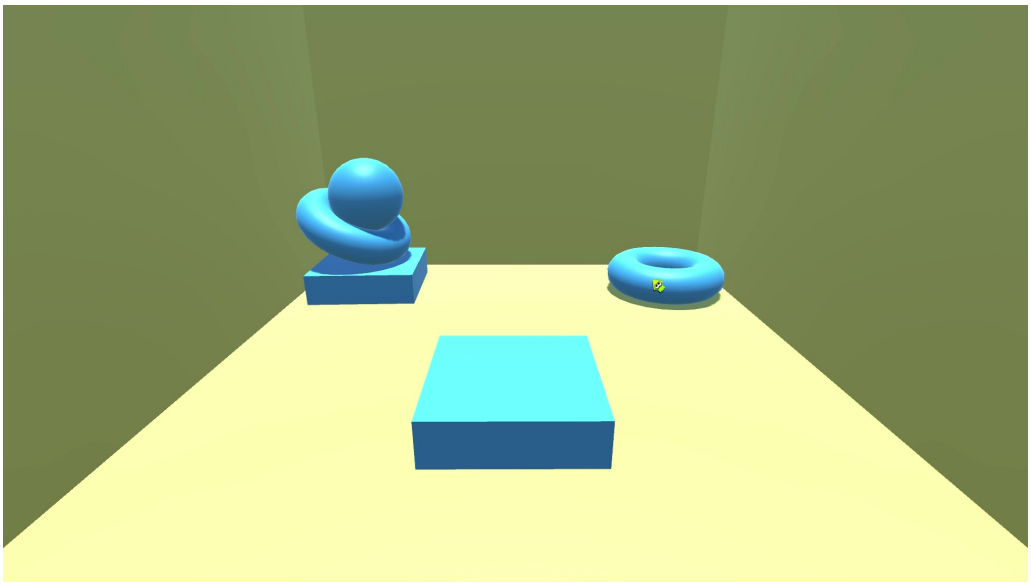

図 4: 初期状態

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．

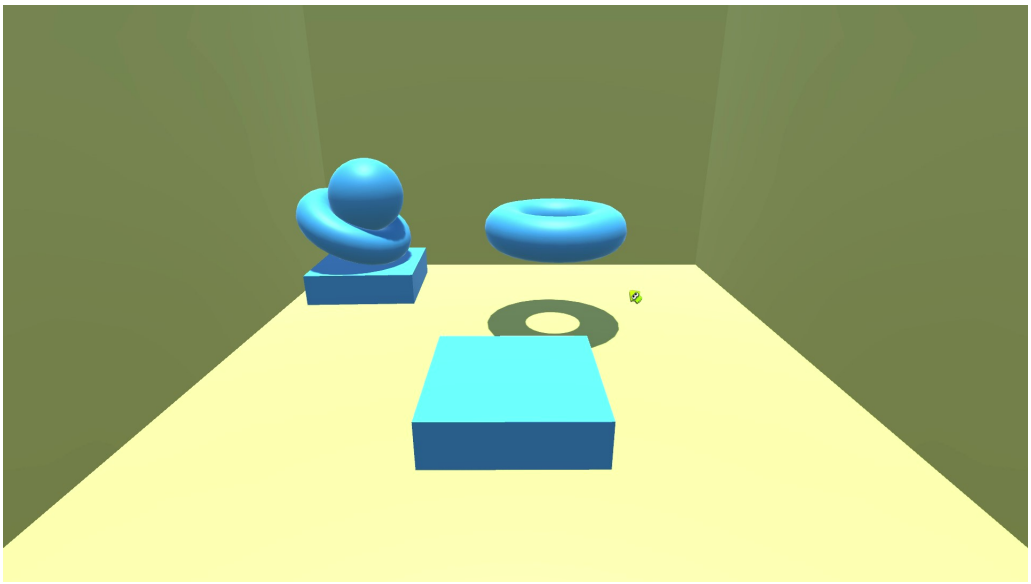

図 5: プランニング開始

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乗せた様子が下図のとおりである．

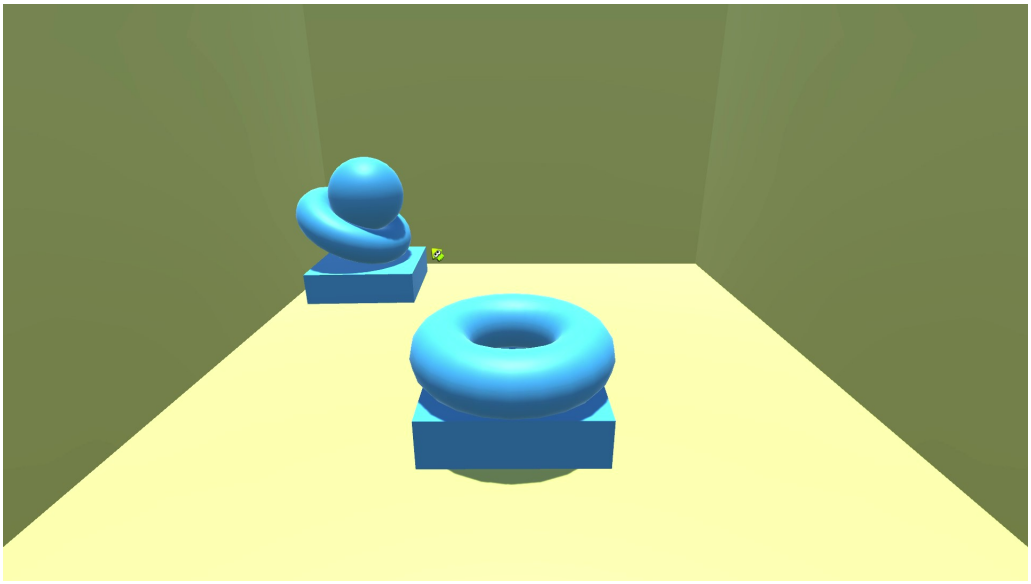

図 6: ステップ 1 完了

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．

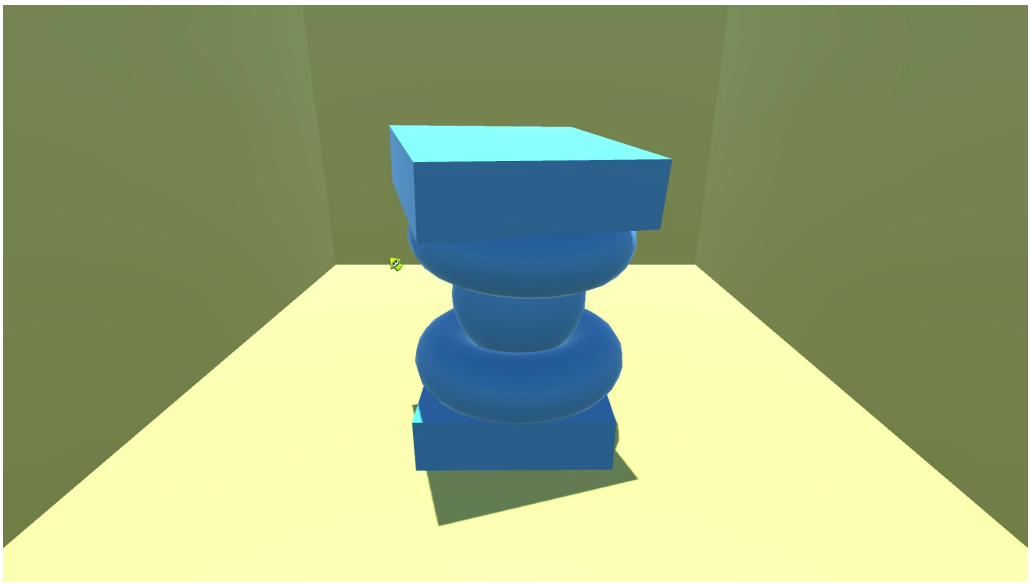

図 7: プランニング完了

3.4 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 のプログラミングならそういうことが起こったことはないので，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．

4 感想

GUI も Swing じゃなくて Unity で作らせてもらえたらなあ．でも Swing の経験も活きた．java プログラムのデータをどう Unity に渡すか．ファイル？

参考文献

Unity Technologies.:『Unity - Manual: Unity User Manual (2019.2)』https://docs.unity3d.com/Manual/index.html（2019/12/09 アクセス）