

知能プログラミング演習II 課題6

グループ8

29114003 青山周平

2020年1月6日

提出物 rep6

グループ グループ8

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 6-1 課題5にやり残した発展課題があれば参考にして拡張しても良いし、全く新しい独自仕様を考案しても構わない。自由に拡張するか、あるいはもし残っていた問題点があれば完成度を高めよ。

2 必須課題 6-1

課題5にやり残した発展課題があれば参考にして拡張しても良いし、全く新しい独自仕様を考案しても構わない。自由に拡張するか、あるいはもし残っていた問題点があれば完成度を高めよ。

私の担当箇所は、発展課題5-7に対してUnityを用いて実装したプログラムの、GUI改善やプランニングへの機能追加である。

2.1 手法

発展課題5-7では、ブロックワールドにおけるプランニングを実現するための過程として、以下のような実装を行った。

1. 空間やプランに関するオブジェクトの生成。
2. プランニングを行うための、オブジェクトの動作等に関するスクリプトの作成。

これに引き続いでは、ブロックワールドにおけるプランニングを実現するために、以下のような方針を立てた。

1. プランに用いるブロックをより正確に生成できるようにする。
 2. ブロックの管理を容易にする。
 3. プランニングの情報を可視化する。
1. に関して、GUIを画面上に表示することで、ブロックを名前や色、形を指定した上で生成できるような仕様とした。
 2. に関して、オブジェクトの情報を一覧で表示して生成したオブジェクトの管理を視覚的に行えるような仕様とした。また、フォーカスしているブロックを視覚的に確認できるように、選択中のオブジェクトのアウトラインが表示されるような仕様とした。
 3. に関して、2. で作成する一覧と連動して、選択したブロックの情報が画面左下のステータスバーで確認できるようにした。

2.2 実装

発展課題5-7で作ったオブジェクトは以下のとおりである。

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.

新しく作った主なオブジェクトは以下のとおりである.

EventSystem GUIにおいてButtonやInputFieldを機能させるための、フォーカス情報等を掌るオブジェクト.

Canvas Generator, Preparator, Staterを子オブジェクトを持つ、GUIの大元となるオブジェクト.

Generator オブジェクトを生成するためのパネル. 子オブジェクトに
InputFieldName,DropdownColor,DropdownShape,ButtonGenを持つ.

Preparator 生成したオブジェクトの一覧を管理するパネル. 子オブジェクトに ScrollView, ButtonRm, ButtonPlanningを持つ.

Stater オブジェクトの情報等を表示するためのパネル. 子オブジェクトに TextStatus, Scrollbarを持つ.

ListObj PreparatorにおけるScrollViewの要素となるプレハブ.

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.
Masterにアタッチされる.

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. Masterにアタッチされる.

Generator Generatorオブジェクトで得た情報に合わせて、ブロックを生成するスクリプト. ButtonGenにアタッチされる.

Destroyer Preparator オブジェクトで選択されたブロックを削除するためのスクリプト. ButtonRm にアタッチされる.

Manager ListObj プレハブにアタッチされる, 各インスタンスが 1 対 1 で対応するブロックを保持するためのスクリプト.

SelectOnList ListObj プレハブにアタッチされる, インスタンスが自身にアウトラインや削除のフォーカスを当てるためのスクリプト.

Starter GUI の一部を非表示にし, プランニングを開始するためのスクリプト. ButtonPlanning にアタッチされる.

CollisionGetter 自身と衝突中のブロックを保持するスクリプト. 各ブロックにアタッチされる.

StateGetter SelectOnList でフォーカスされたブロックと衝突中のブロックを Stater に表示するスクリプト. Master にアタッチされる.

2.2.1 プランに用いるオブジェクトをより正確に生成できるようにする.

Unity[1]において GUI を作成するために, Unity の標準機能に含まれる uGUI[2] を用いた.

まず, GUI 制作を Unity 上で開始するために, Canvas オブジェクトを生成した. Canvas オブジェクトは実行環境によって, そのサイズが動的に変化するため, シーンビューにおける大きさは意味を持たない. その仕様を理解するまでは扱いに戸惑った.

次に, ブロック生成のための GUI である Generator オブジェクト(図 1)を作るために, Generator オブジェクトには Panel を, 名前を入力するためのフォーム (InputFieldName) には InputField を, 色や形の選択 (DropdownColor, DropdownShape) には Dropdown を, 生成ボタン (ButtonGen) には Button をそれぞれ用いた. これら UI に関するパーツはいずれも RectTransform コンポーネントを用いて配置される.

RectTransform の特徴として, アンカーとピボットが挙げられる. これらはブロック等のオブジェクトの座標指定で用いられる Transform コンポーネントには含まれない. 今回ピボットは用いなかったが, アンカーは GUI パーツの配置を行う上では欠かせないものであることが分かった.

アンカーは, 指定した位置を基準として, オブジェクトの相対座標を指定するものである. 図 1 に示すように, Canvas の右上部である位置に

表示されている4つの三角がアンカーの基準点を示すものであり、パネルはアンカーから左下部に相対座標で配置されている。これにより、実行環境により画面サイズが変化しても、図2や図3で示すようにレイアウトを崩さずに表示されることが確認できる。

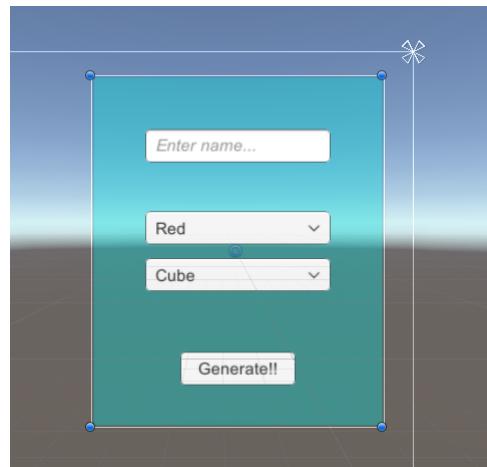


図1: Generator オブジェクト

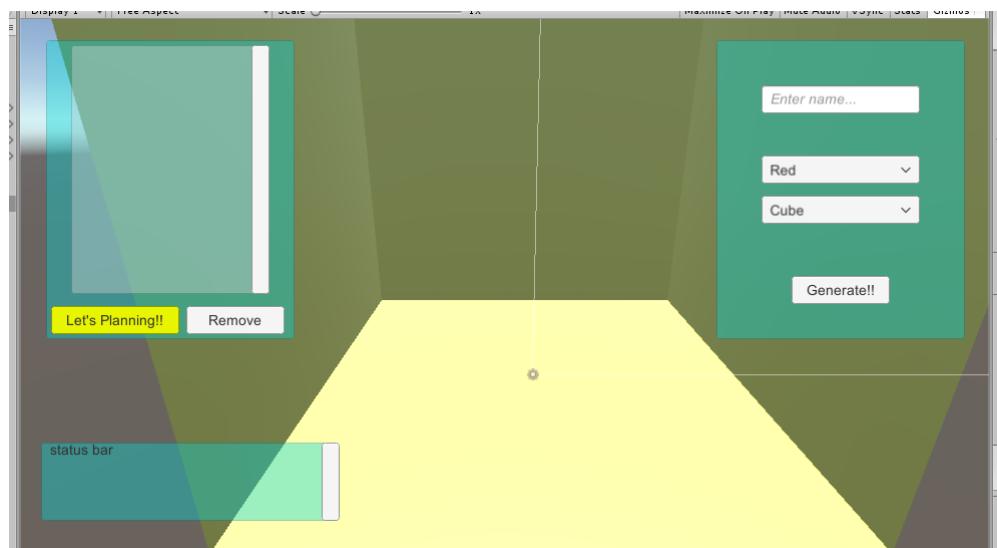


図2: 横長の画面における表示

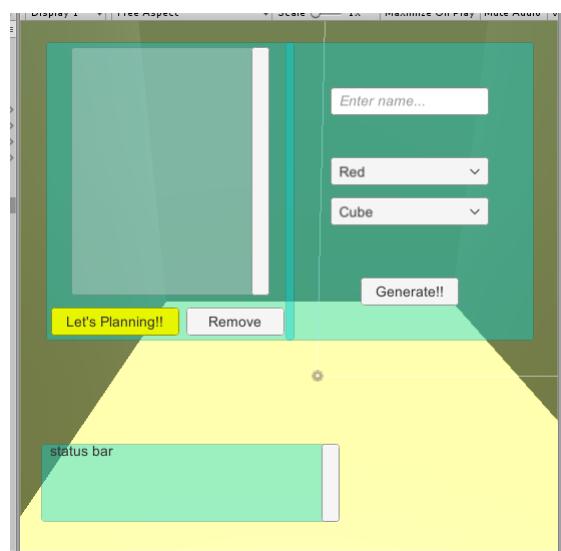


図 3: 縦長の画面における表示

次に、配置した Generator 内のオブジェクトを用いて、ブロックを生成するためのスクリプト (Generator.cs) を生成する。ブロックはソースコード 1 で示すように生成される。

ソースコード 1: Generator クラス

```
1 public class Generator : MonoBehaviour
2 {
3     ...
4     public void Generate()
5     {
6         ...
7         GameObject obj = (GameObject)Resources.Load("Cube
8             ");
9         switch (ddShape.value)
10        {
11            case 0:
12                obj = (GameObject)Resources.Load("Cube");
13                break;
14            case 1:
15                obj = (GameObject)Resources.Load("Sphere
16                    ");
17                break;
18            case 2:
19                obj = (GameObject)Resources.Load("Torus");
20                break;
21        }
22        GameObject target = Instantiate(obj, obj.transform
23             .position, Quaternion.identity);
24        target.name = name;
25        target.GetComponent<Renderer>().material.color =
26            getColor(ddColor.value).color;
27
28
29        ...
30
31        Material getColor(int num)
32        {
```

```
33     Material color = matR;
34     switch(num)
35     {
36         case 0:
37             color = matR;
38             break;
39         case 1:
40             color = matG;
41             break;
42         case 2:
43             color = matB;
44             break;
45     }
46     return color;
47 }
48 }
```

ddShape.value を介して DropdownShape で選択された形を選択し, Resource.Load メソッドでプレハブを取得している。取得したプレハブを Instantiate メソッドで生成し, 生成したインスタンスの名前を InputFieldName で入力した名前に書き換え, 色を ddColor.value を介して DropdownShape で選択された色に書き換えている。

このスクリプトを ButtonGen にアタッチすることで, ボタンをクリックしたときにこのスクリプトは呼び出され, 実行される。

2.2.2 ブロックの管理を容易にする.

ブロックを管理するための GUI である Preparator オブジェクト（図 4）を作るために、Preparator オブジェクトには Panel を、ブロックを表示するリスト (ScrollView) には ScrollView を、リスト内の要素 (ListObj) には Button をプレハブ化したもの、リスト内の要素削除ボタン (ButtonRm) とプランニング開始ボタン (ButtonPlanning) には Button をそれぞれ用いた。

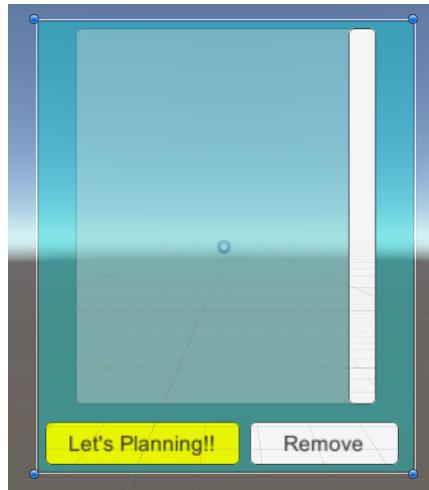


図 4: Preparator オブジェクト

ScrollView を作成する [3] にあたり、ListObj は要素の選択を可能にするために、Button を用いて実装した。

Generator スクリプト (ソースコード 1) の変数 `_text` によって、ListObj からインスタンスが生成されている。この際、スクロールバーに表示するためには適切な位置の子オブジェクトとして格納する必要がある。これは、`Instantiate` の引数に親オブジェクトの `transform` を渡すことで実現した。また、ここで生成したインスタンスに、先程生成したブロックのインスタンスへの参照も、Manager スクリプトの変数を介して渡している。これにより、ListObj のインスタンスから対応するブロックへのアクセスを可能としている。

次に、ScrollView から選択した要素にアウトラインを付けるためのスクリプト `SelectOnList`, `StateGetter` を実装する。

まず、アウトラインは Unity の標準機能に備わっていないため、アセット QuickOutline[4] を導入し、各ブロックに非アクティブ状態でアタッチした。

その上で、引数で渡されたブロックのアウトラインをアクティブにする StateGetter クラス内の FocusOutline メソッドを、ソースコード 2 に示す。

ソースコード 2: StateGetter クラス内の FocusOutline メソッド

```
1     public GameObject target;
2
3     public void FocusOutline (GameObject newObj) {
4         if (target != null) {
5             target.GetComponent<Outline> ().enabled =
6                 false;
7         }
8         if (newObj != null) {
9             newObj.GetComponent<Outline> ().enabled = true
10            ;
11        }
12        target = newObj;
13    }
```

アウトラインをアクティブにしたブロックを target に保持することで、フォーカスが外れた際の非アクティビ化も同時に見えるようになっている。StateGetter は Master に 1 つだけアタッチされることで、フォーカスの管理を一元的に行うこととした。

SelectOnList スクリプトは各 ListObj インスタンスにアタッチされることで、フォーカスが当てられたときに SelectOnList スクリプトから FocusOutline メソッドを呼び出すことで、自身にアウトラインを付けることができる。(図 5,6)

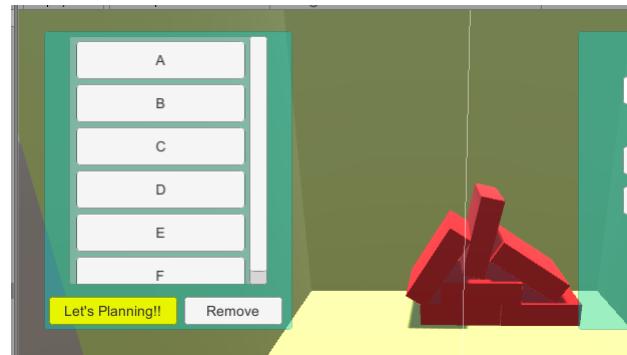


図 5: 非フォーカス時

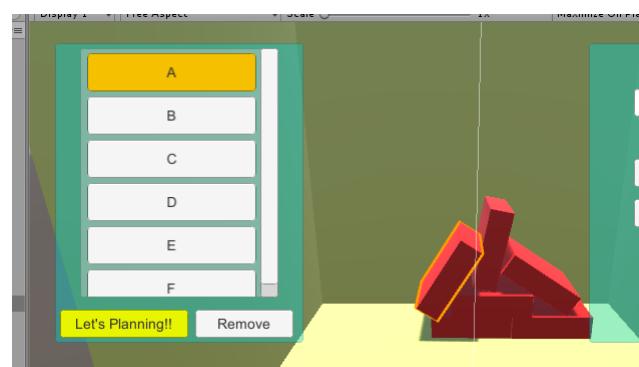


図 6: フォーカス時

2.2.3 プランニングの情報を可視化する.

プランニングの情報表示のための GUI である Stater オブジェクト（図 7）を作るために、Stater オブジェクトには Panel を、情報を表示するテキスト (TextStatus) に Text を用い、一度に全部を表示できないときのために Scrollbar を作成した [5].

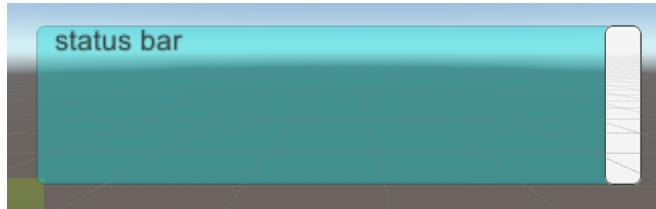


図 7: Stater オブジェクト

フォーカスされたオブジェクトの衝突情報を取得するために、CollisionGetter スクリプトを作成し、各ブロックにアタッチした。それをソースコード 3 に示す。

ソースコード 3: CollisionGetter クラス

```
1 public class CollisionGetter : MonoBehaviour {
2
3     public List<GameObject> colList;
4
5     void Awake () {
6         colList = new List<GameObject> ();
7     }
8
9     void OnCollisionStay (Collision collision) {
10         if (!colList.Contains(collision.gameObject)) {
11             colList.Add (collision.gameObject);
12         }
13     }
14     void OnCollisionExit (Collision collision) {
15         colList.Remove (collision.gameObject);
16     }
17 }
```

衝突相手のブロック名が colList に 1 つだけ格納される。この際、同じブロックに対して複数の衝突判定を持つときに colList に複数同じ要素が

含まれないようにするために、要素の追加は OnCollisionEnter メソッドではなく OnCollisionStay メソッドを用いて常に監視することで、colList の要素が常に正確になるような実装をした。

このリストを StateGetter から取得し、Stater に反映することで、フォーカスしたブロックの衝突判定を、常に可視化することを実現した。

2.3 実行例

task5-7/build/Block World Planning.exe を起動したところ、図 8 のような画面が得られる。

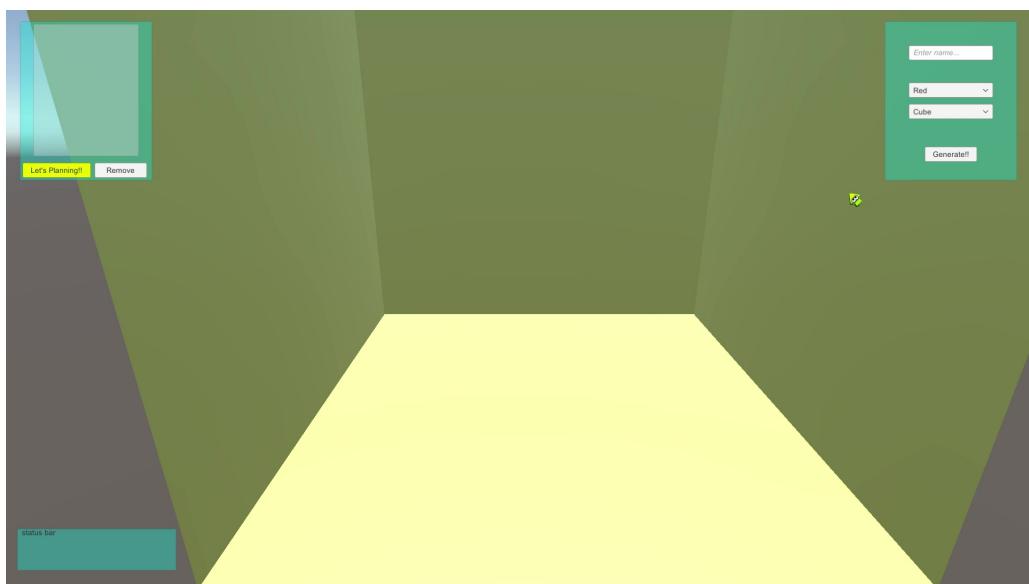


図 8: 起動時の画面

Generator を用いてブロックを生成すると、Preparator にも反映されることが分かる(図 9,10).

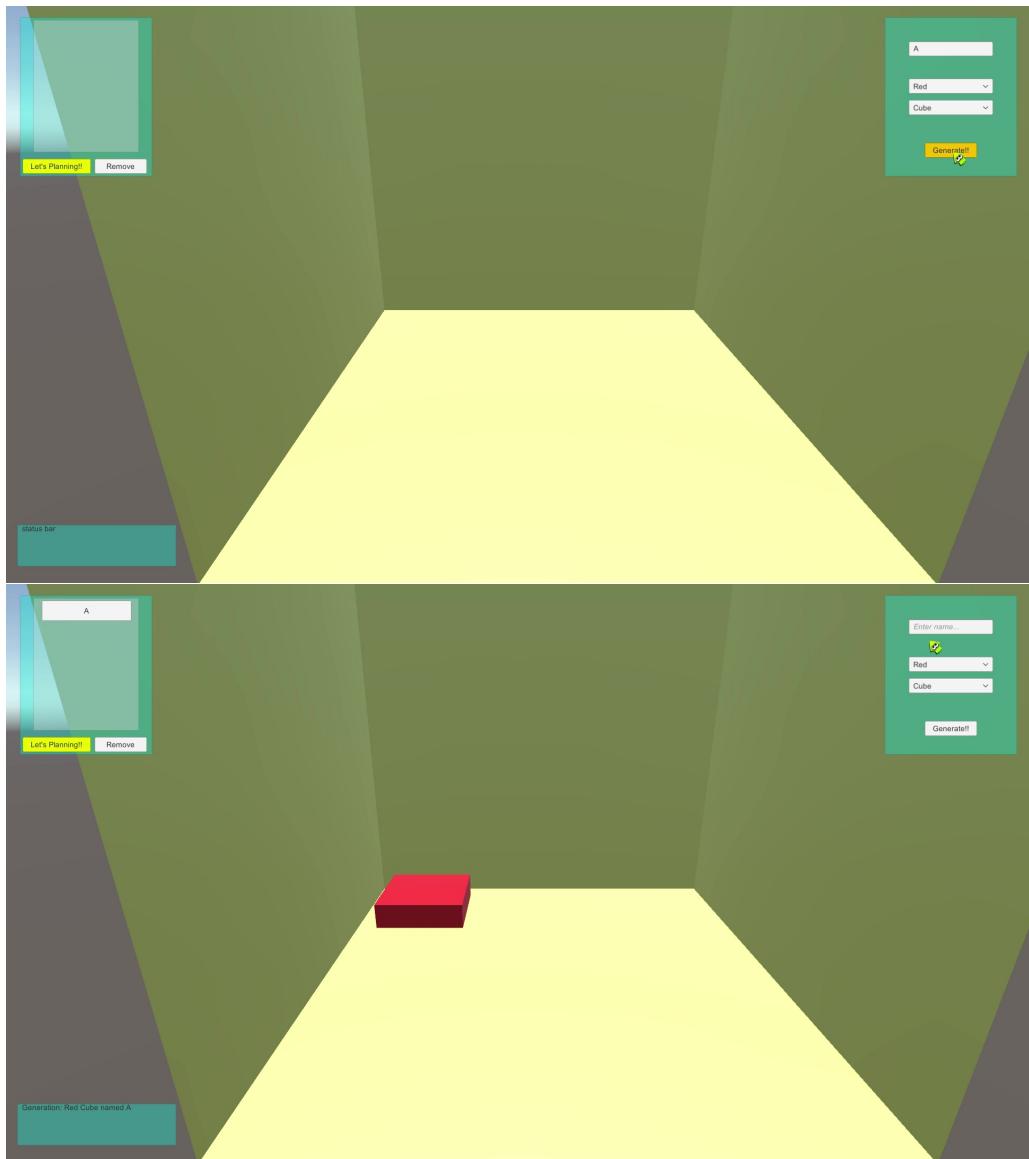


図 9: 赤い球を A という名前で生成

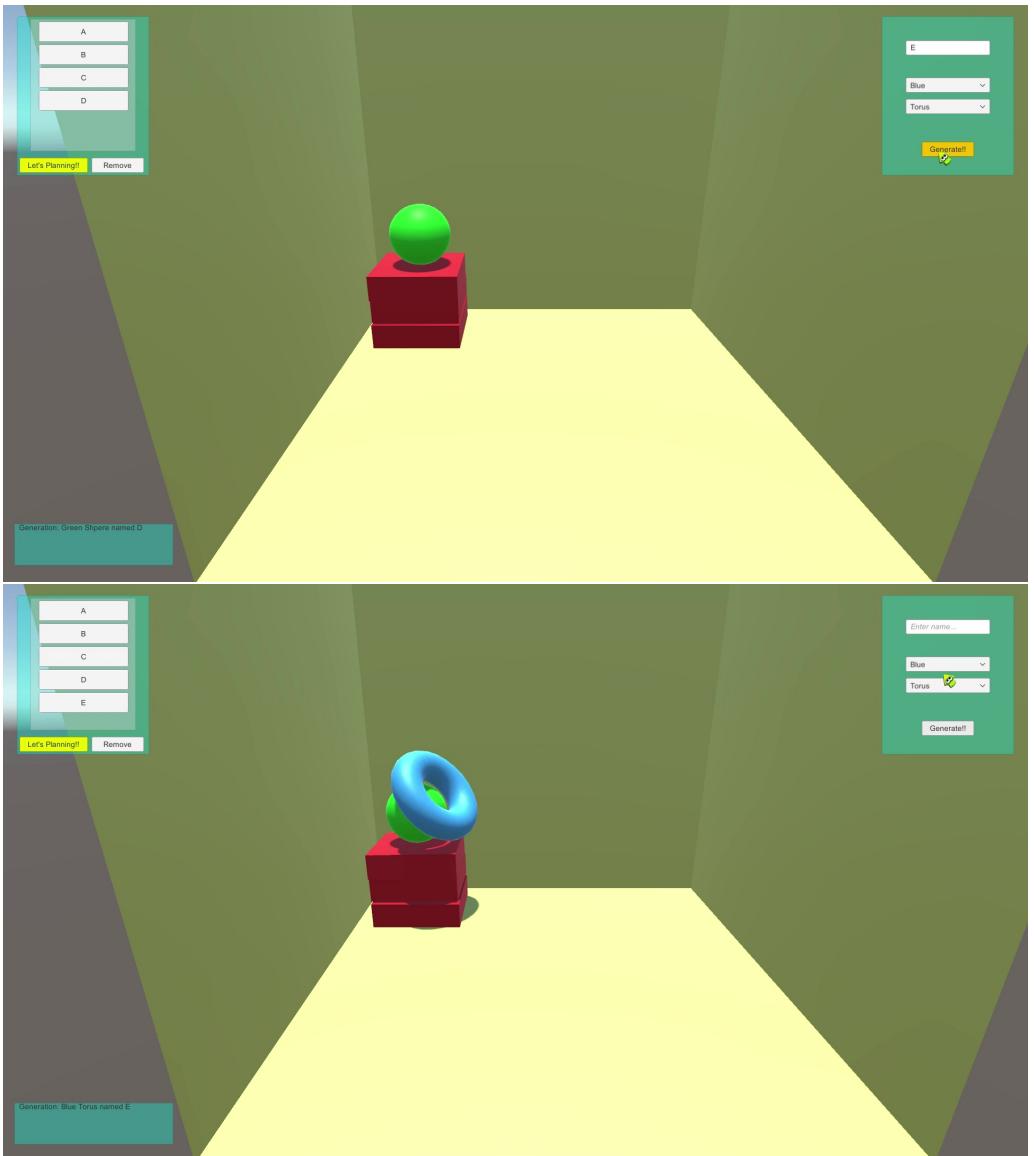


図 10: 青い円環体を E という名前で生成

Preparator で要素を選択すると、対応するブロックのアウトラインが表示される(図 11).

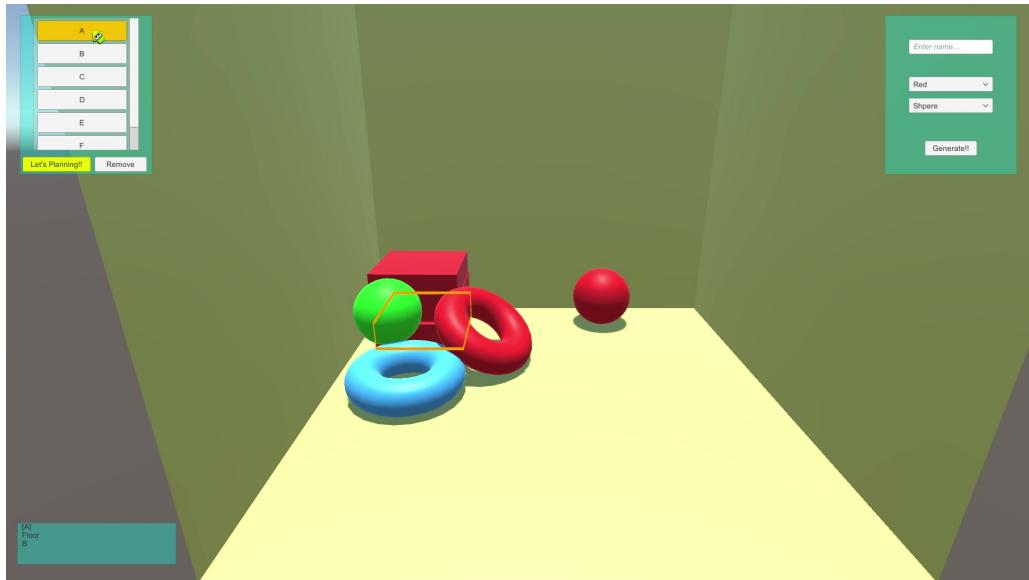


図 11: A にフォーカス

フォーカスしたブロックは Remove ボタンで削除できる (図 12).

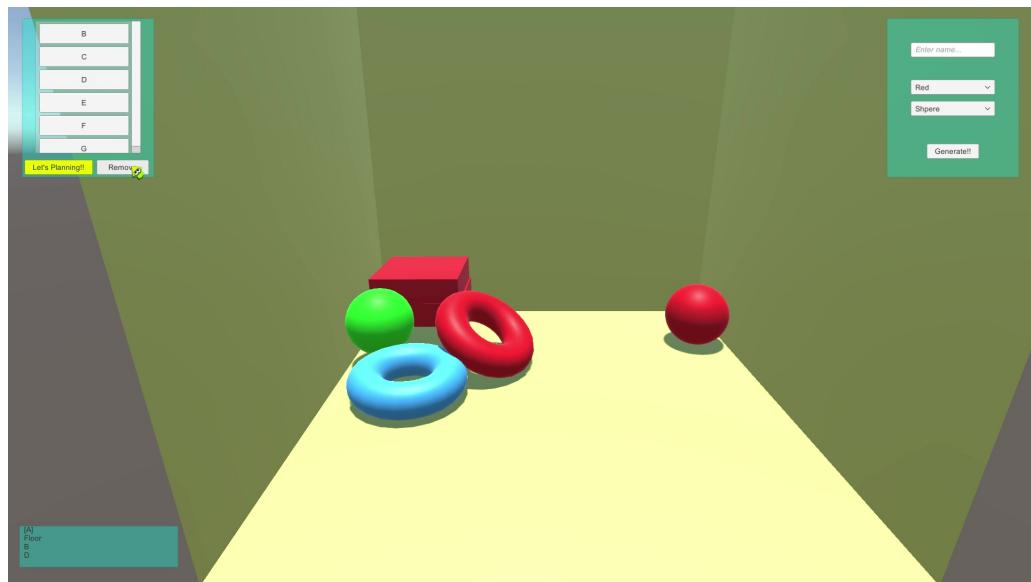


図 12: A を削除

ブロックの生成や削除が完了したら、Let's Planning ボタンでプランニングを開始する(図13)。

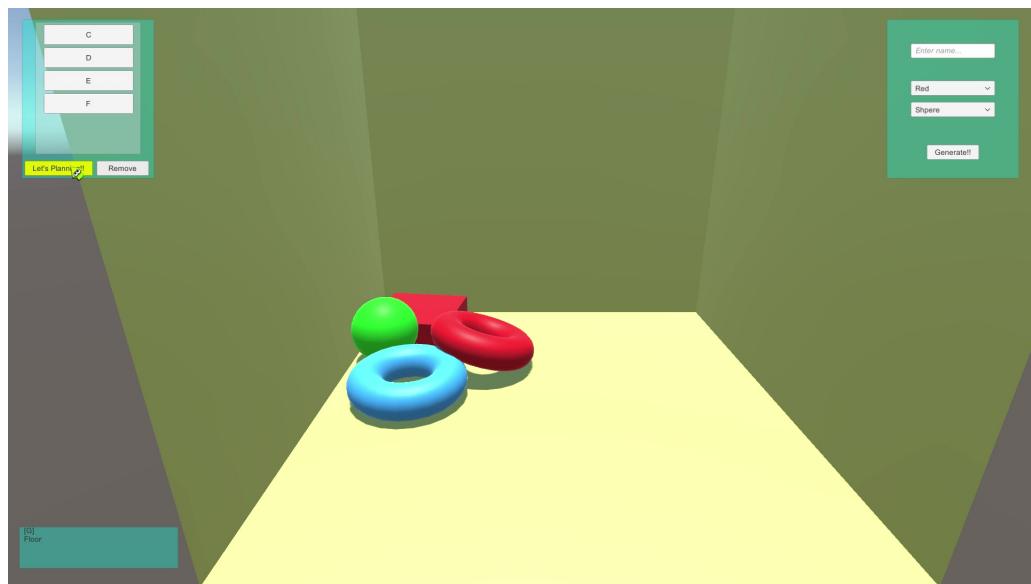


図 13: プランニング初期状態

フォーカスしたブロックと衝突しているブロックの情報がStaterに表示される(図14,15).

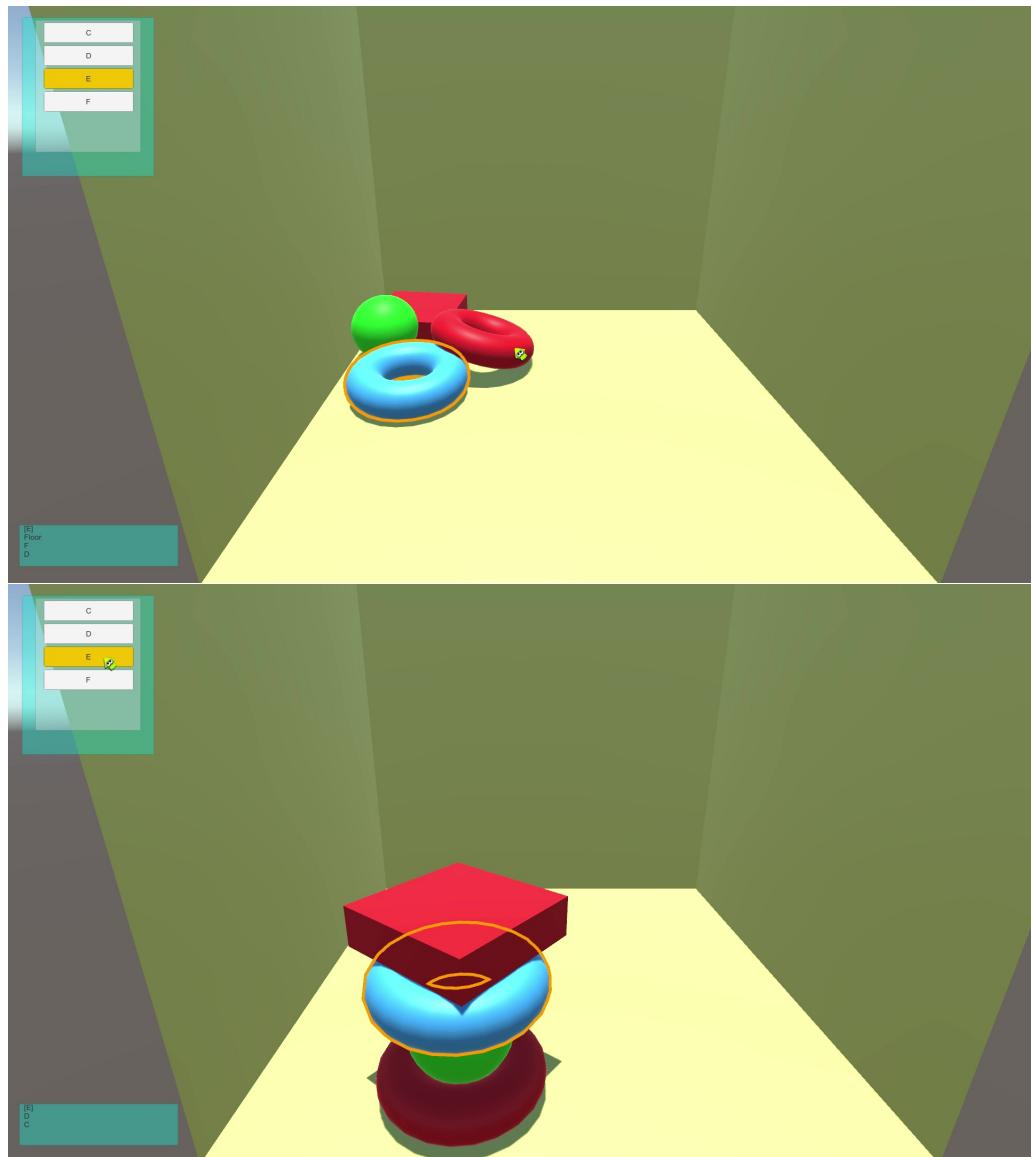


図14: Eにフォーカスを当て、プランニング実行

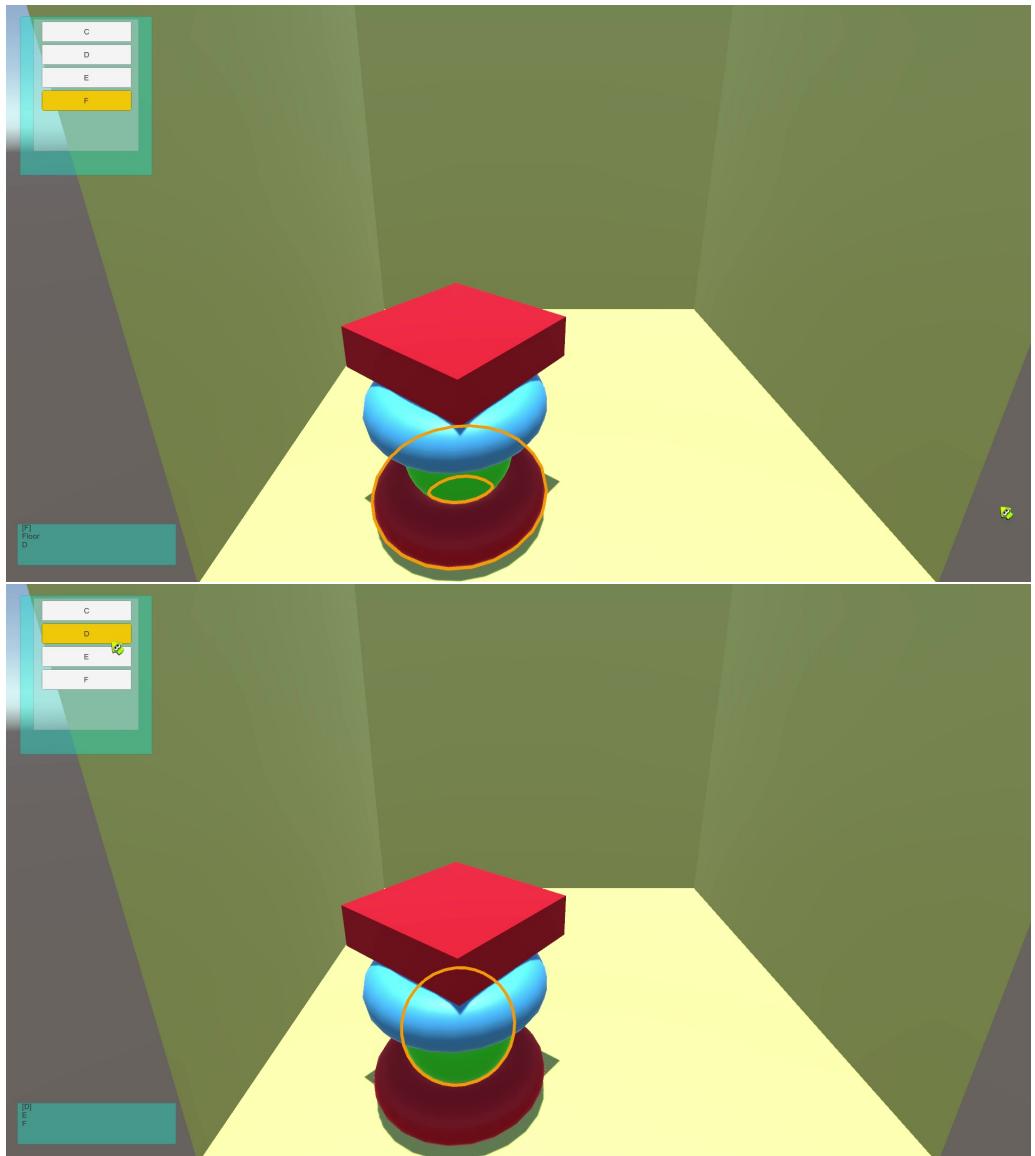


図 15: プランニング完了時の各ブロックの衝突情報

2.4 考察

Generator スクリプト (ソースコード 1) にて, switch を用いているが, ここで用いる変数 obj の宣言の際, 初期化しないとコンパイルエラーになることは C# の特徴として挙げられると考えられる.

GUI を実装し始めた際, 自動で追加された EventSystem というオブジェクトを誤って消してしまい, 実行しても Button や InputField が全く反応しないという事態に陥った. EventSystem を追加し直すことで修正できたが, このように自動で用意されるものに甘んじて各種機能を使うことは, 便利な反面で, 何かトラブルが起こったときに簡単には自力で直せないという事態が起こり得ることに繋がるため, 自分が利用している機能はどのようなものに基づいてどのように動いているのかを探求することは, 今後それを用いてより高度なことをするときのために大切なことだと感じた.

uGUI での GUI 実装と Swing での GUI の実装と比較してみると, Swing よりも並列動作は扱いやすく, Swing よりも変数の管理は面倒だと感じた. Unity での GUI の部品となる各オブジェクトは, 常時独立して動き続けているため, 並列的な動作を行ったときにどのような挙動をとるかをオブジェクト毎に確認できるため挙動の管理がしやすいのだと考えられる. 一方で, 各オブジェクトが独立しているということは, オブジェクト間の変数はスクリプトを用意して共有する必要があるということなので, 並列動作の分かりやすさと変数の管理の楽さはトレードオフの関係にあることが考えられる.

また, uGUI と Swing のいずれも, フォーカスの操作はとっつき辛いと感じた. しかし, uGUI では GUI 全体のフォーカスが 1 つの EventSystem によって明確に行われているため, その点では uGUI の方が把握しやすいと感じた.

しかし, GUI の一番の難点は, やはり実行する環境によって表示のされ方が異なってくることだと考えられる. その点では, Swing も標準の機能としてレイアウトが用意されてはいるが, uGUI でのアンカーを用いた配置の方が, 斷然とっつきやすく, 調整もしやすいと感じた.

以上のことから, 今後 GUI を作る際は, 基本的に Swing よりも uGUI で良いと考えられる. しかし, 軽量化という点において uGUI は Swing に優ることができるか, または優る必要があるかということは今後の課題である.

Unity を使って一番悩まされたことが、プレハブにアタッチしたコンポーネントの変数が public でも、その変数に他のオブジェクトを直接アタッチして参照させることができなかったことである。そのため、GameObject.Find メソッドで毎度目的のオブジェクトを探して参照を作っているが、直接アタッチできる方が効率的だと考えられる。前回も、通常のオブジェクトだと上手くいくのにプレハブだとうまくいかないようなこと (Torus の衝突判定) があったように、プレハブの特性や注意点で理解できていないことがまだ多々あることが考えられる。Unity を使っていく上でプレハブは欠かせない機能なので、今後も積極的に使って学んでゆきたい。

また、Unity ではオブジェクトを消去するという機能があるため、この点も注意して使っていかなければいけないと考えられる。Unity に限らず並列処理を行うようなプログラムでは、先程まで参照していたものが突然なくなるということが往々にして起こり得るため、その点も留意した上でプログラム実装を行ってゆく必要があると考えられる。

また、Unityにおいて、変数をオブジェクト間で共有するために、いちいちスクリプトを作成する必要があると述べたが、共有する変数ごとにスクリプトを作成するのではなく、共有すべき変数をまとめて管理するようなスクリプトと、そのためのオブジェクトを作成して、そこを共有変数の仲介場所とすると改善できることが考えられる。例えば、今回のプログラムでは Stater がその仲介場所として適していたと考えられる。更に、Stater を仲介場所にすればステータスの表示もより自在に行えるようになることが考えられ、一石二鳥である。また、このような方法で共有する変数を管理したほうが、プログラムの保守性にも良いということが考えられる。

今回の実装について、開始時点を決めれるようにしたことで、初期状態の明確化ができるようになった。

最終的に自動でのプランニングを実装するには、その前段階として実装する必要のあるものが多くあったため、その過程の一部を実装しようと思い今回作ったような実装となった。ブロックの属性付与や生成したブロックの管理、衝突情報の管理が行えるようにしたことで、プランニング実装に向けて次に必要なのは、衝突情報を突き詰め、どちらのブロックが上に乗っているかといった相対的な位置関係の取得・管理をするこ

とと、それに基づきどの物体を目標状態に向けてどう動かすかを選択することの実装だと考えられる。しかし、3次元空間における物体間の衝突の仕方は多様であり、どちらのブロックが上に乗っているかを厳密に判別することは困難な場合が多く存在することが考えられる。また、それを解決したところで、物体を動かすことによる状態変化も不定であり、目標状態に向かた行動の選択を一意的に決めるることは難しいと考えられる。このように、実装上の難しさだけでなく、そもそも3次元空間上においてどのようにプランニングを定義するかという実世界的な難しさが数多くあることを改めて痛感した。

3 感想

アウトラインの実装にあたって、今回は既存のアセットを導入して実装したが、初めは自分でアウトラインのシェーダを作ることでの実装を試みていた。しかし、Shaderもそれ自体が1つの言語であるように奥深く、思ったとおりに実装することは難しかった。自在に物体を彩るのには欠かせない知識であるため、いずれシェーダについても勉強してみたいと思う。

これまでLaTeXでレポートを書いてきたが、相互参照が文字化けすることが多くあった。しかし、タイプセットを2回すれば解決できる[6]ということが知れて、嬉しかった。

この講義全体を振り返り、GUIを作る能力や、チームで協力して1つのプログラムを作る能力、GitHubやSlackやLaTeXを使う能力を大きく向上することができたと感じている。特に、これまでチームでプログラムを作ることに苦手意識があったため、この経験を今後にも活かしてゆきたいと思う。

また、チームメンバーにも非常に恵まれていたと思う。互いに支え合い、高め合うことができた。この縁も大切にし、今後にも繋げてゆきたい。

参考文献

- [1] Unity Technologies.:『Unity - Manual: Unity User Manual (2019.2)』
<https://docs.unity3d.com/Manual/index.html> (2020/01/06 アクセス)

- [2] 竹内 大五郎 : 『uGUI チュートリアル - Metal Brage』
<http://www.metalbrage.com/UnityTutorials/uGUI/index.html>
(2020/01/06 アクセス)
- [3] shimoaraiso : 『Unity の ScrollView で一覧表示を作成する』
https://tech.pjin.jp/blog/2016/08/30/unity_skill_3/ (2020/01/06 アクセス)
- [4] chrisnolet : 『chrisnolet/QuickOutline: Unity asset for adding outlines to game objects』 <https://github.com/chrisnolet/QuickOutline>
(2020/01/06 アクセス)
- [5] hiyotama : 『【Unity 開発】uGUI の Scrollbar の使い方 【ひよこエッセンス】 - Unity(C#)初心者・入門者向けチュートリアル ひよこのたまご』 <https://hiyotama.hatenablog.com/entry/2015/07/04/090000>
(2020/01/06 アクセス)
- [6] PukiWiki Developers Team. : 『【LaTeX 入門/相互参照とリンク - TeX Wiki』 <https://texwiki.texjp.org/?LaTeX?LaTeX%2F%20相互参照とリンク#h84e81eb> (2020/01/06 アクセス)