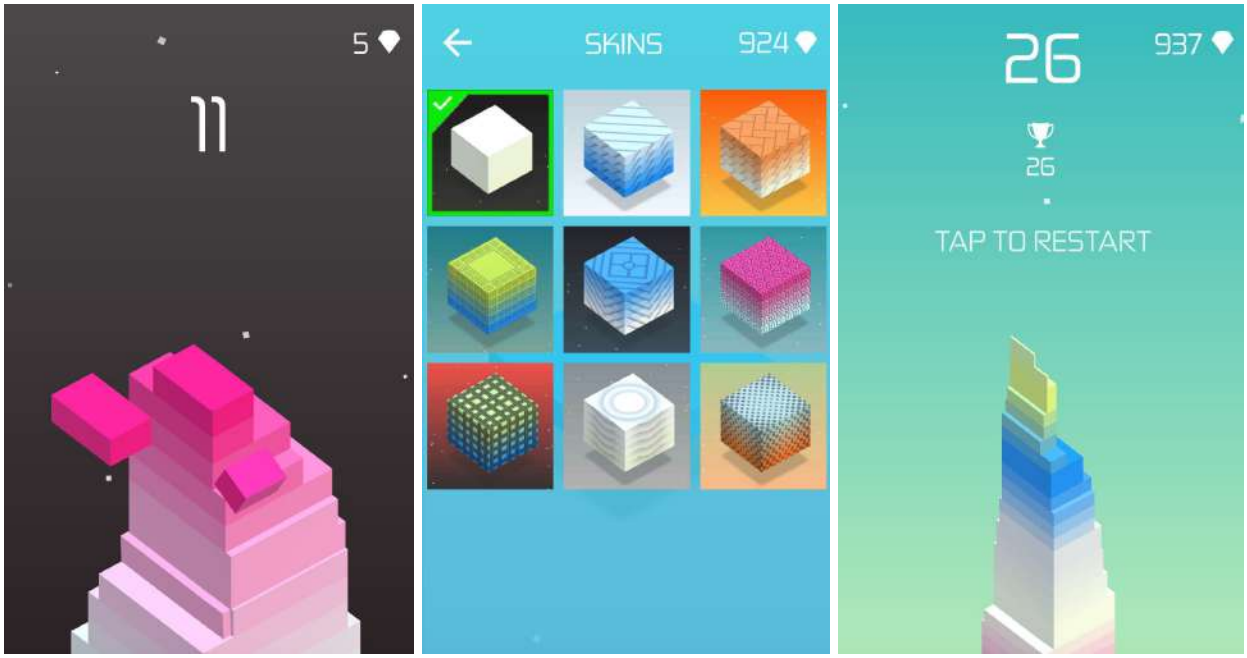
POLY&CODE ;

Stack the Blocks

Poly&Code

1.Introduction



The game consists of block moving back and forth on top of another block and Player 's goal is to stop the block exactly on top of the underlying block to create a tower of blocks as high as possible. If the incoming block does not overlap with the underlying block at all, it's game over.

If stopped when the block is exactly on top of the other, the block is stacked with exactly the same size as the underlying one. If failed to do so, the part that is left outside of the underlying block will be chopped off and fall down. The next block will be of this new block's size. As a result, the blocks will get smaller increasing the difficulty as the game progresses.

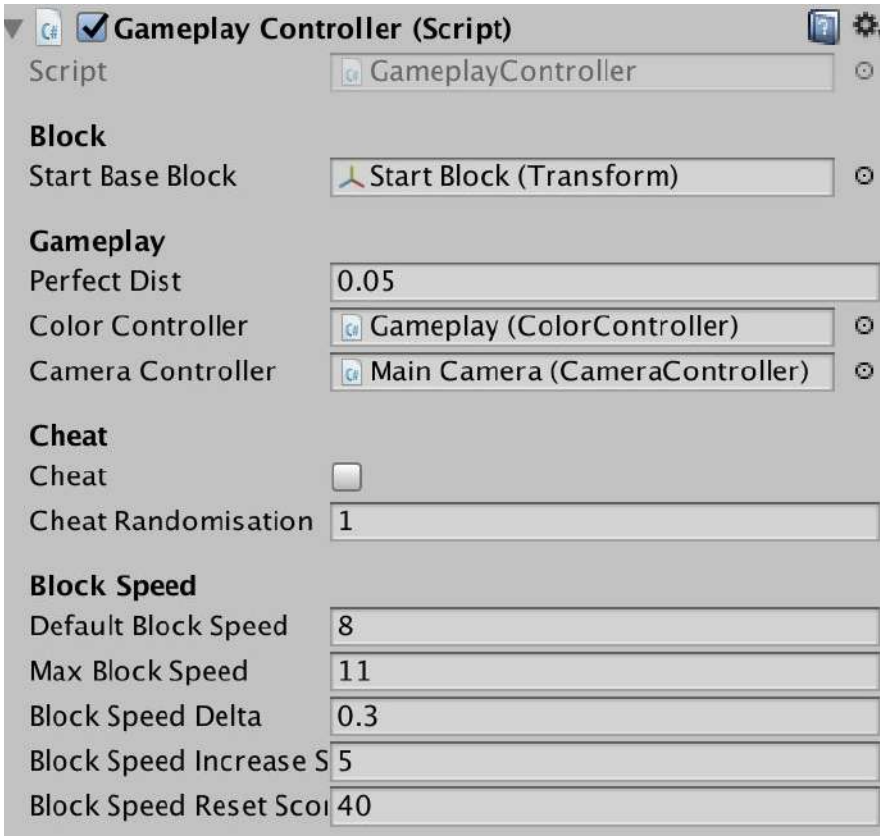
Every stacked block adds to the Player's score. A particular number of scores will give one diamond. Diamonds can be used to buy skins which can be applied to the blocks.

2. Customizing the Game

2.1 Tweaking the gameplay

2.1.1 Gameplay Controller

You can find the *Gameplay Controller* on *Main → Gameplay* " GameObject. This script controls the blocks and the respective stacking, chopping etc. Tweak these values according to your liking.

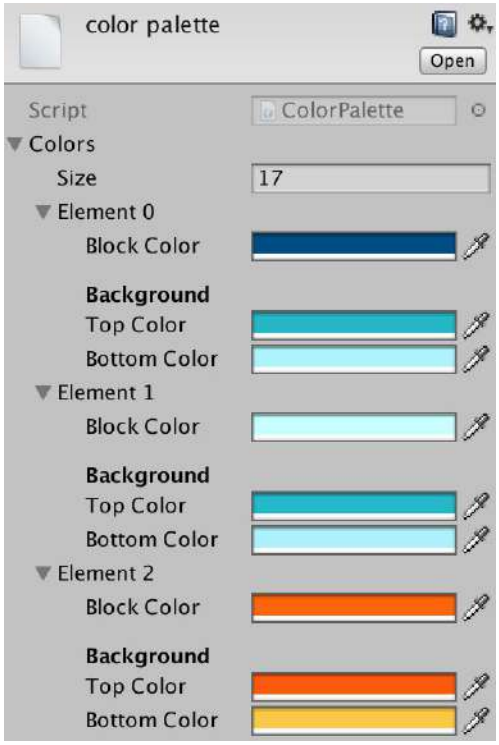


- *Start Base Block:* The block from which the game starts.
- *Perfect Dist:* The threshold for a perfect overlap. Higher the value, easier the perfect overlap.
- *Cheat:* Game plays by itself if turned on.

- *Cheat Randomization:* The playing accuracy when *cheat* is on. Smaller the value, better the accuracy.
- Default Block Speed : Default speed of the block when game starts
- Max Block Speed : The Maximum Block Speed
- Block Speed Delta : Block speed increases by this value
- Block Speed Increase Score : Block speed increases every this score
- Block Speed Reset Score : Block speed resets to default value every this score

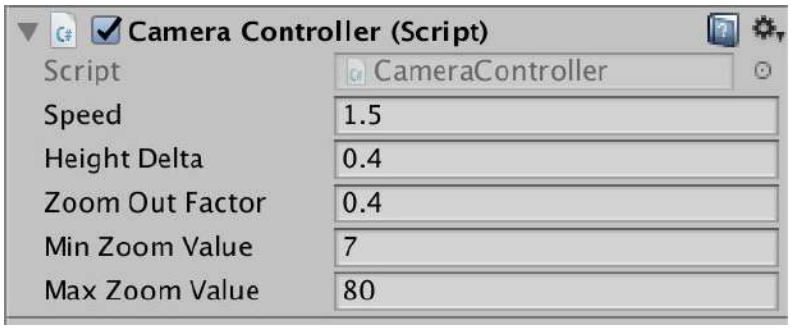
2.1.2 Color Palette

This is a *Scriptable Object* asset containing a list of colors which are applied to the block and background during the gameplay. Each item is a set of block color and the corresponding top and bottom color for the background. You can create a new asset by going to *Assets → Create → Stack the blocks → Color Palette* or you can edit the already created list located at the *SO Assets* folder.



2.1.3 Camera Controller

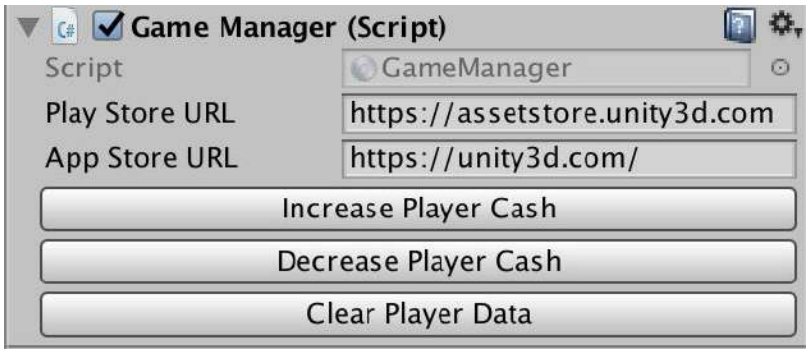
This script is attached on the Camera and controls the movement and zooming out of the camera at gameover. Tweak the values according to your liking.



- *Speed*: Speed at which the camera moves up with every stacked block
- Height Delta: Amount by which the camera moves up
- *Zoom out Factor*: Factor by which the camera zooms out at game over
- *Min Zoom value, Max Zoom value*: Minimum and maximum zoom value to avoid undesired results

2.2 Game Manager

Find the Game Manager at *Main → Game Manager* and set up the respective store links for game rating. You can also use the buttons to increase, decrease or clear player's cash for testing purposes.

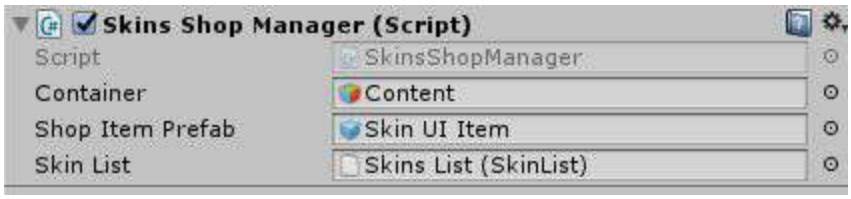


2.3 User Interface

The User Interface is controlled by UIManager on *Main - UI and Shop*. The screens and other UI references are straightforward and pretty easy to set up. The sprites for all the UI are located at *Artwork → UI → Sprites*.

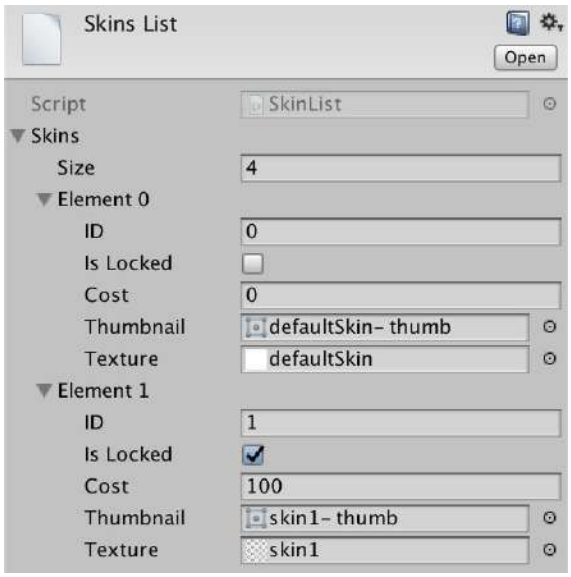
2.4 Shop

The Skin shop is controlled by the *SkinsShopManager* script on the *Main → UI and Shop* GameObject . This takes a list of skins which is again a *Scriptable Object* asset.



2.4.1 Skins List

The list of skins can be created by going to *Assets → Create → Stack the Blocks → Skins List*. The existing list is in *SO Assets* folder.



- *ID:* Unique ID for every skin
- *isLocked:* is the skin locked by default
- *Cost:* Cost of the skin

- *Thumbnail:* The UI thumbnail for this skin
- *Texture:* The skin texture

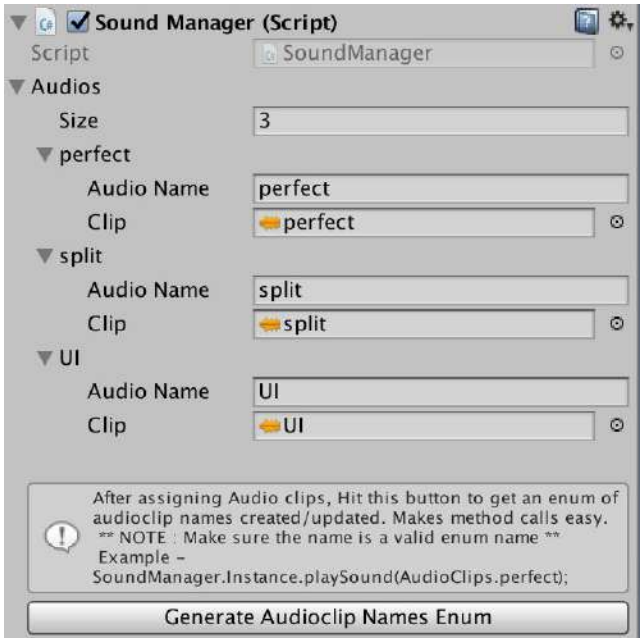
IMPORTANT: Please note that the IDs must be unique. Also, the isLocked setting is the default setting when the game loads for the first time. The lock status of skins is stored at the persistent path and the lock status of a skin in the *Skin list* asset will not change with unlocking it during the game.

2.4.2 Dynamic Grid Fitter

The *Dynamic Grid Fitter* is a component attached to the shop item list container in UI. This component works with the *Grid Layout Group* to calculate the cell size for items at runtime according to the screen resolution and given aspect ratio, number of columns and

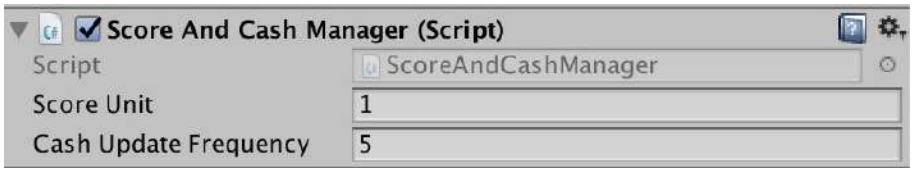
2.5 Sounds

The game sounds are managed by *SoundManager* and can be found on the *Main→Sound Manager* GameObject. The sounds can be assigned in the inspector.



2.6 Score and Cash

The score and Cash/Diamonds are managed by *ScoreAndCashManager* script on *Main → Score and Cash* GameObject.



- *Score Unit:* Score increases by this value.
- *Cash Update Frequency:* Cash/Diamonds update after this many scores.

3. Gameplay Logic in detail

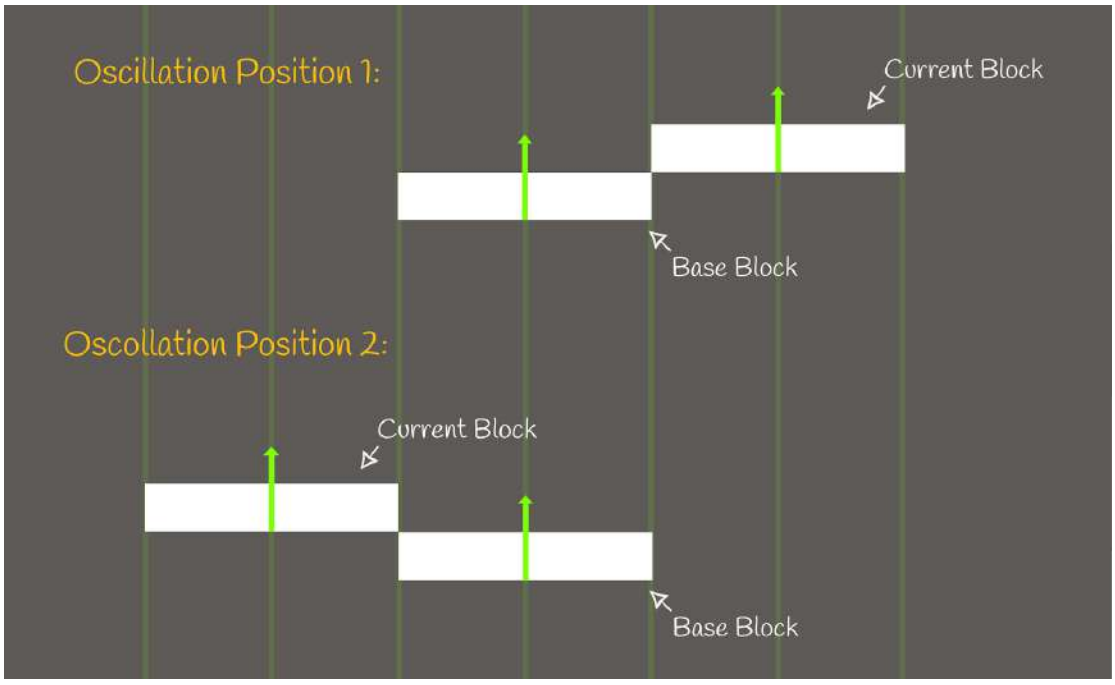
The GameplayController script takes care of the complete gameplay. We will go through the complete gameplay logic in detail. Let's understand a few things first to make the further explanation easier:

- *Start Base block*: The already present block from which game starts.
- The incoming block is of the same scale as the base block.
- The incoming blocks come from x and z direction alternatively. The logic is both for same. Only the vectors change. Blocks incoming from x have the change in x component and those from z have the change in z component. This is where the *xOrZ* boolean, conditionals and if statements are utilized in a lot of places. **Having stated this, the further explanation will show only x block of the code.**
- All the diagrams below will be as 2D side views.

3.1 Incoming blocks and the range of oscillation

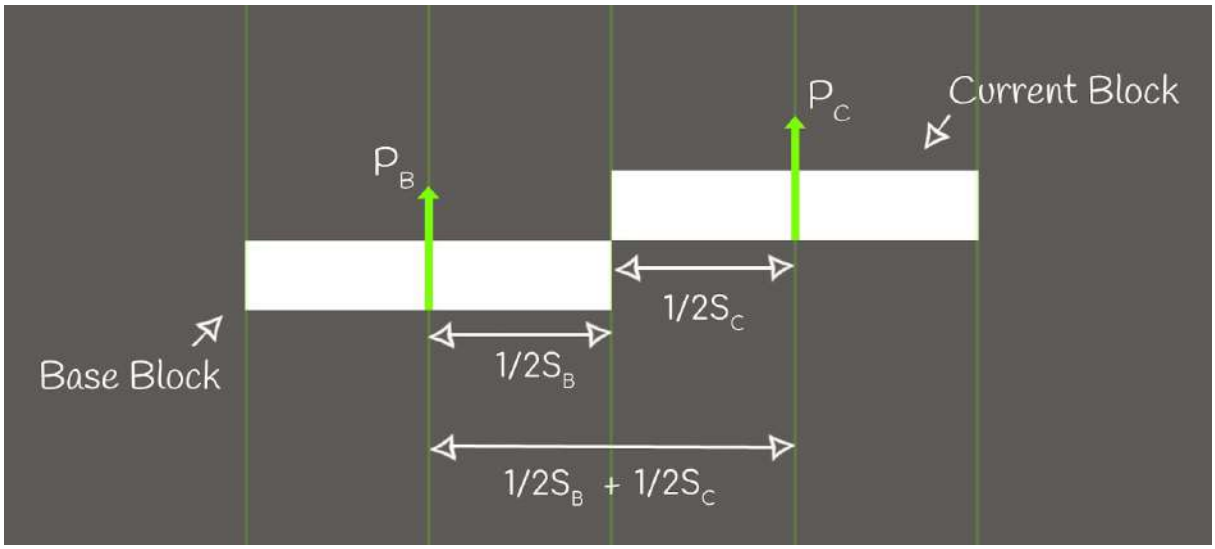
The game starts with the Start base block in the hierarchy. The *BringNewBlock()* method handles instantiation of new block and calculation of the range in which it moves back and forth on top of the underlying block.

The gameplay is in such a way that the oscillation range of all the incoming blocks should be equal to or greater than the bounds of the first Block of the game.



Though the bounds will always be the same for all blocks, the start and end position will never be the same for all the blocks since:

- Pivot is at the centre of each block.
- The size of the blocks change as we progress.
- Blocks come from x and z direction alternatively.



If you look at the image above, you will notice that the start position of the block is at a distance of *half of the base block's scale (½ S_b) + half of the incoming block's scale (½ S_c)*. In other words, this is the minimum distance from the centre of *start base block* for the start and end positions of oscillation. Let's call all it *offset.*

```
float offset = startBaseBlock.transform.lossyScale.x * 0.85f + block.transform.localScale.x
* 0.5f;
```

85 % of the *start base block's* scale is taken just to maintain some distance.
This *offset* is then added and subtracted from the centre i.e the *start base cube's* x position to get the range of oscillation for the incoming cube. Y coordinate will be equal to the *previous cube' height* + *heightoffset* and *z* will be the same as the previous block since this oscillation is along x.

```
float y = baseBlock.transform.position.y + yOffset;
float z = baseBlock.transform.position.z;
block.GetComponent<BlockScript>().setPositions(
                new Vector3(startBaseBlock.transform.position.x + offset,
y, z),
                new Vector3(startBaseBlock.transform.position.x - offset,
y, z));
```

The instantiation position is offset by twice the value so that the block starts from outside the screen

```
block.transform.position = baseBlock.transform.position - offsetVector * 2
+ yoffset;
```

3.2 Block Chopping logic

There are three cases when the player taps :

3.2.1 Game over

It is gameover When user taps and no part of the current block overlaps with the base block. For this, It is required to determine how far the block has to be from the base block so that they do not overlap. It is clear from any of the above images that the game-over distance is *half of base block's scale + half of the current block's scale.*

```
  float gameOverDist = xOrZ ?
currentBlock.transform.lossyScale.x * 0.5f + baseBlock.lossyScale.x * 0.5f
:  currentBlock.transform.lossyScale.z * 0.5f + baseBlock.lossyScale.z *
0.5f;
```

Also, we are interested in the distance between the blocks along one axis or the XZ plane only. For this, some custom methods are implemented which simply return the vector along one axis (For example : (2,9,8) to (2,0,0) along x)

```
Vector3 currentBlockPosVector = xOrZ ?
VectorHelper.GetXVector(currentBlock.transform.position) :
VectorHelper.GetZVector(currentBlock.transform.position);
Vector3 baseBlockPosVector = xOrZ ?
VectorHelper.GetXVector(baseBlock.transform.position) :
VectorHelper.GetZVector(baseBlock.transform.position);
```

Now simply check the distance and if it is greater than the game over distance, it is gameover

```
if (Vector3.Distance(currentBlockPosVector, baseBlockPosVector) >
gameOverDist)
 {
   //It is game over
 }
```

3.2.2 Perfect score

A perfect score is when the block is exactly on top of the base block. This is calculated by simply finding out the distance between the two vector positions in the above section. The smaller the distance the better the overlap. Since it can be extremely difficult to tap when the blocks are perfectly overlapping, it is a good idea to keep a threshold. When tapped if the distance between the blocks falls in this range, it is a perfect score.

```
 if (Vector3.Distance(currentBlockPosVector, baseBlockPosVector) <
perfectDist)
{
//Perfect Score
}
```

3.2.3 Block splitting

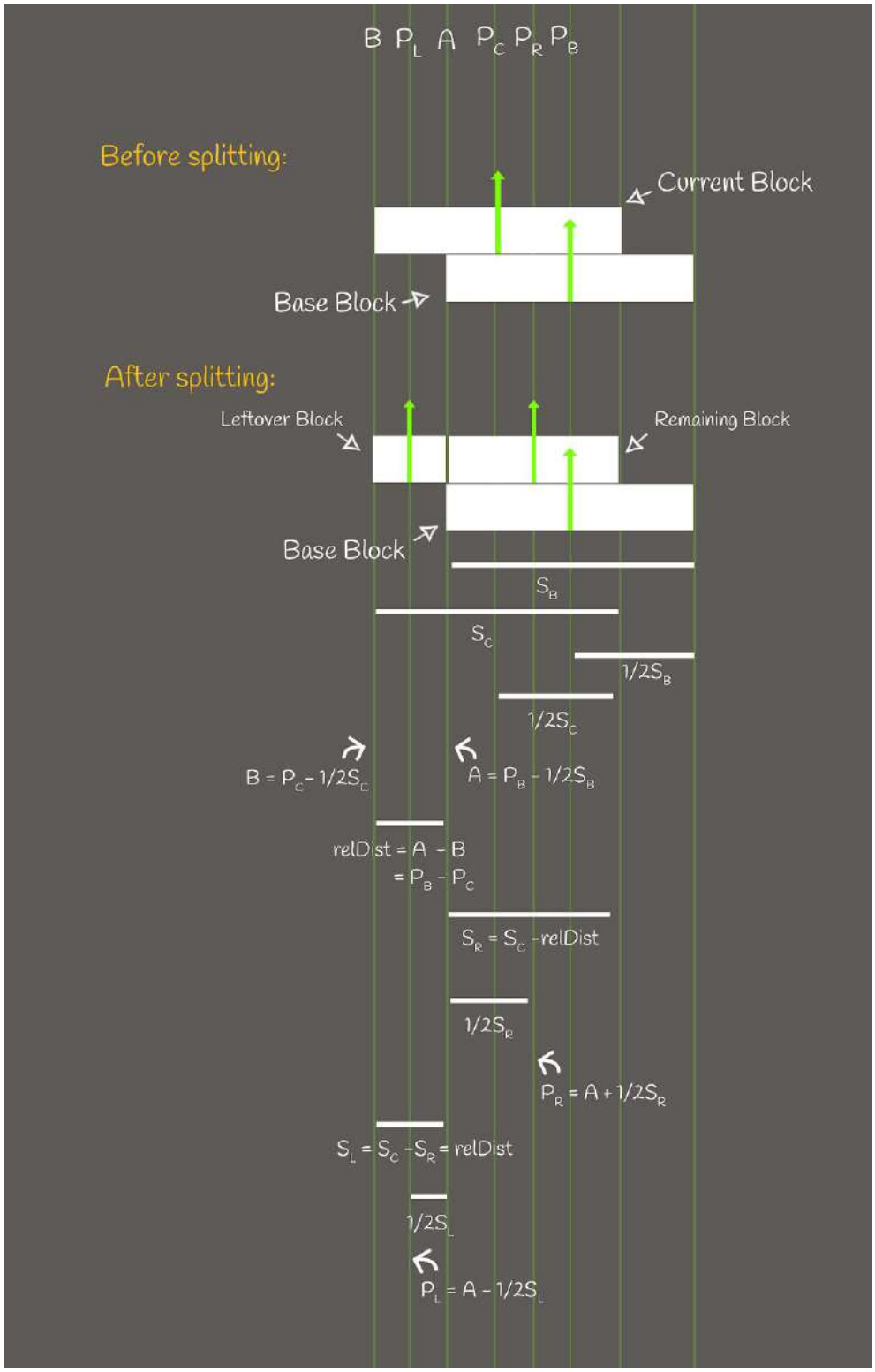
Now here's the main part of the gameplay. When it is not a perfect overlap and the current block has to split into two pieces: One which will be kept and the game continues on top of it and the other one which simply falls down. Let's call the two pieces *remaining* and *leftover.*

Following steps are implemented to achieve the desired result:

1. Calculate the position and scale of the two blocks: remaining and leftover
2. Make the current block the remaining piece by assigning it the position and scale of the remaining piece
3. Instantiate a new block as the leftover piece and assign it the position and scale of the leftover piece
4. Add a Rigidbody to the leftover piece to make it fall

There are two cases when splitting the block. Let's look at these in detail.

Case 1: Current Block is behind the Base block



P_B = Position of Base block

S_B = Base Block's scale

P_C = Position of Current Block

S_C = Current Block's scale

P_R = Position of remaining block after the split

S_R = Scale of remaining block after the split

P_L = Position of the leftover block after the split

S_R = Scale of the leftover block after the split

Points A and B will be

A = P_B - ½*block's scale

B = P_C - ½*block's scale

Let's calculate the distance between point A and point B first and call it *relDist*.

relDist = A - B

1. Now *relDist* becomes:

relDist = P_B - ½*block's scale - (P_C - ½*block's scale)

relDist = P_B - P_C

2. The Remaining block's scale:

S_R = block's scale - relDist

3. The Remaining block's Position:

P_R = A + ½* block's scale

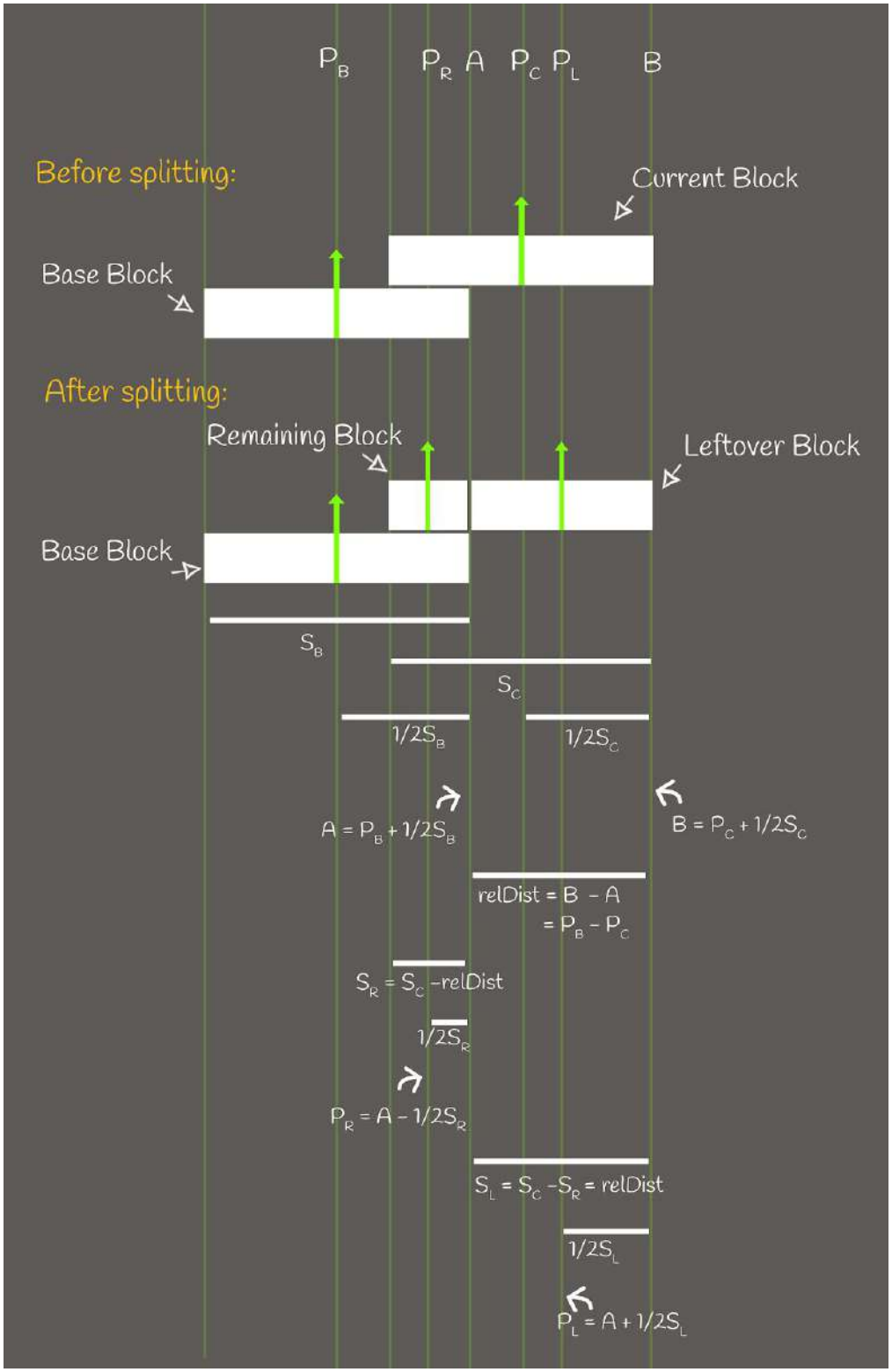
4. Scale for the leftover piece:

S_L = relDist = block's scale - New block's scale

5. Position for the leftover piece:

P_L = A - ½ *discarded block's scale

Case 2: Current Block is ahead of the Base block



P_B = Position of Base block

S_B = Base Block's scale

P_C = Position of Current Block

S_C = Current Block's scale

P_R = Position of remaining block after the split

S_R = Scale of remaining block after the split

P_L = Position of the leftover block after the split

S_R = Scale of the leftover block after the split

Points A and B will be

A = P_B + ½*block's scale

B = P_C + ½*block's scale

Let's calculate the distance between point A and point B first and call it *relDist*.

relDist = B - A

1. Now *relDist* becomes:

relDist = P_B + ½*block's scale - (P_c + ½*block's scale)

relDist = P_B - P_C

2. The Remaining block's scale:

S_R = block's scale - relDist

3. The Remaining block's Position:

P_R = A - ½* block's scale

4. Scale for the leftover piece:

S_L = relDist = block's scale - New block's scale

5. Position for the leftover piece:

P_L = A + ½ *discarded block's scale

If you look at the calculation in both the cases, you will observe that the difference is only in the interchanging of addition and subtraction. This is why an *int* variable *sign* is taken which is either 1 or -1 according to the current case.

```
int sign = currentBlockPos < baseBlockPos ? -1:1;
```

Following is how the above calculation looks like in code. (Line numbers correspond to the calculations above):

1.
```
float relDist = Mathf.Abs(baseBlockPos - currentBlockPos);
```

2.
```
float remainingBlockScale = baseBlockScale - relDist;
```

3.
```
float remainingBlockPos = (baseBlockPos + sign * baseBlockScale * 0.5f) -
sign * remainingBlockScale * 0.5f;
```

4.
```
float leftoverBlockScale = relDist;
```

5.
```
 float leftoverBlockPos = remainingBlockPos + sign * remainingBlockScale *
0.5f + sign * leftoverBlockScale * 0.5f;
```

4. Game Architecture and Modules

The game consists of various modules which communicate with each other directly or through events.
On top, we have the *GameManager* which consists of the game state events and general game functionalities.
All the other modules register to events present in *GameManager* and when events are raised the respective functionalities in these modules execute. For instance, when a game initializes the *GameInitialized* event is raised after which:

- *UIManager* switches to homescreen
- *GameplayController* sets up the blocks
- *ShopManager* fetches the selected skin and assigns to the *GameplayController*
- And so on....

4.1 PnC Casual Game Kit

The ***PnCCasualGamekit*** namespace consists of a set of reusable and independent scripts which provide fundamental functionalities such as UI management, object pooling, data storage, sound management, shop etc. These modules can be taken out of this project and implemented elsewhere without having to worry about removing other script references, though a set of instructions must be followed.

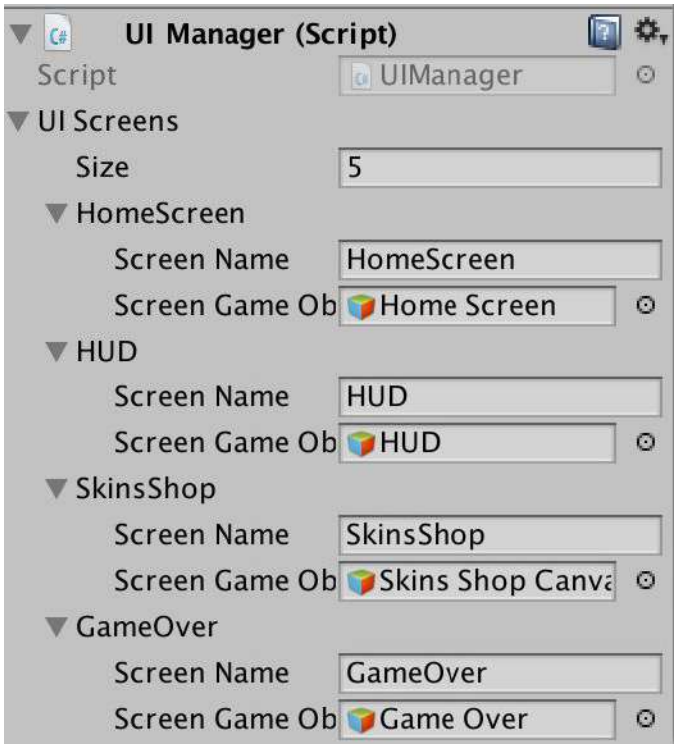
4.2 User Interface

The *PnCUIManager* provides basic functionalities required to manage the game UI:

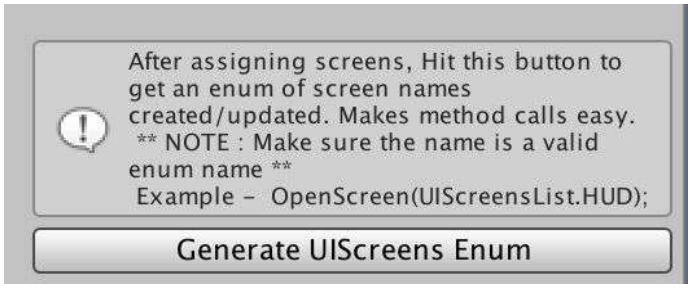
1. Switching between screens
2. Modal windows(pop-ups)

You can use the *PnCUIManager* directly, though in this project it is extended in *UIManager* script so that project-specific methods to maintain reusability.

The list of UI Screens consists of screen names and their respective Gameobjects.



After assigning the screens hit the “Create enum” button in the inspector to generate an enum of the screen names.



The enum file will be generated at the script’s location.

```
public enum UIScreensList
{
    HomeScreen,
    HUD,
    SkinsShop,
    GameOver,
    BuyPopUp,
}
```

This is not mandatory but makes referencing screens easy and readable.

```
UIManager.Instance.OpenScreen(UIScreensList.GameOver);
```

Note: Make sure names are unique and follow enum naming rules. (Editor validations will be added in next update)

4.3 Object Pooling

The *ObjectPooler* script provides fundamental object pooling functionality along with some important methods to manage the pool. This is again a reusable script and is not coupled with any other script in the project.

The items to be pooled should be dropped in the *ItemsToPool* list with Name, Prefab, amount and if it should expand specified. As with *UIManager* here also you can press the create enum button in the inspector to generate an enum of the Item names. This enum makes referencing items easy and readable. For example:

```
ObjectPooler.Instance.GetPooledItem(Itemstopool.block);
```

Note: Make sure names are unique and follow enum naming rules. (Editor validations will be added in next update)

4.4 Sound Management

The *PnCSoundManager* script provides the basic functionality for the playing of sounds and is also one of the usable scripts. It is extended in the *SoundManager* script. It also follows a similar enum creation approach as described for UIManager and ObjectPooler above.

4.5 Data Management

PlayerDataHandler provides a generic solution for storing, updating and retrieving data from persistent storage. To use this class all you have to do is extend this class and declare all the data fields in it. For example:

```
[System.Serializable]
public class PlayerData : PlayerDataHandler<PlayerData> {
    public int score;
    public int highScore;
    public float cash;

    public PlayerData()
    {
        score = 0;
        cash = 10;
        highScore = 0;
    }
}
```

Now you can use the singleton instance to do all the data related operations.

Create Instance: Accessing the Singleton instance for the first time automatically creates the instance and fetches data from the storage if present. If data is not present it creates a new empty instance. It is still recommended to create the instance at game start.

```
PlayerData.Create();
```

Fetch data:

```
Float cash =  PlayerData.Instance.cash;
```

Update Data:

```
 PlayerData.Instance.score += 10;
 PlayerData.Instance.SaveData();
```

Clear data:

```
PlayerData.Clear();
```

4.6 Shop

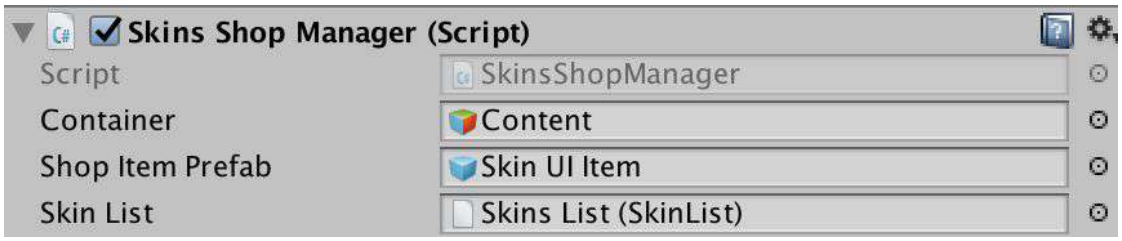
The Shop Module is also a reusable module of the project which provides the following functionalities:

- Populates the UI list of shop items.
- Updates the item's status according to the saved data.
- Handles selection of an item with an appropriate callback.
- Handles successful/failed unlocking of items with appropriate callbacks.

The module works on top of three classes: *ShopManager*, *ShopItemData* and *ShopItem* Implementation of the shop involves the following:

1. Extend ShopItemData to create your Data model class:
   The ShopItemData is the base model data class containing basic data fields such as ID, lock status, cost and thumbnail. Your data class must extend from this class with other project-specific data fields. Refer *SkinItemData* in the project.
2. Extend ShopManager passing the extended ShopItemData as the generic type :
   The ShopManager consists of all the implementations for generating, locking/unlocking of shop items. All you have to do is extend from this class and provide implementations for all the abstract methods. Refer *SkinsShopManager* in the project.
3. Create the Shop item Prefab: The shop item is basically a Prefab with a button and other lock and selection images components. It also has a script component called *ShopItem* which basically relays the item's data to *ShopManager* on selection and toggles the selection and lock view of the item. Refer "Skin UI Item" under the Prefabs folder.

Once done with the above steps, Attach the extended *ShopManager* script on any GameObject in the hierarchy. You will find two variables exposed in the inspector.



- *Container*: UI GameObject which will contain all the shop items. Basically the content child of a scroll view.
- *Shop Item Prefa*b: The list item Prefab. Prefab is already set up in the project. Prefab must have ***ShopItem*** the script attached with all the necessary references.

The Skin list is a Scriptable Object asset for easy management of skin textures. It is project specific field defined in the derived *ShopManger* class. It is already described in the "Customizing the template" section.

Thank you for Purchasing the Asset.

Best of Luck!

For any suggestions, feature requests and support write at:

PolyandCode@gmail.com