Documentation

Peabody
Software Engineering Project

Dominik Radulay

2021/2022

Faculty of Engineering and Environment
Northumbria University
United Kingdom

0.1 Requirement Analysis

As a first stage of the development process it was important to capture requirements to establish a broad and comprehensive understanding of the expected outcome of the whole process is and what it entails. Requirement captures served as the foundation for the whole project, however, as previously mentioned in the Development methodology section, new requirements were captured as part of weekly sprints and code was changed accordingly. Because of the nature of the project being remake of originally game made for PC to Android device some mechanics needed to be change to reflect different user interface, the most obvious of which being controls (touch screen, as opposed to a keyboard or a mouse and keyboard). Large parts of early requirements capture was described already in Terms of Reference document (appendix **??**) in the Gameplay Section. For a better understanding of requirements, they are divided into multiple Subsections described in the following.

Game Mechanics

Peabody is a game where the user is given a figure called, as the name of the game indicates, Peabody. However, as opposed to conventional games, instead of controlling the protagonist/character (in our case the Peabody figure itself), the player controls the environment. Therefore, levels can be best described as frames filled with various blocks like solid blocks/bricks, cracked blocks, diamonds, or empty spaces (all blocks are illustrated in Appendix 0.6) as displayed on the mock-up of a game level in figure 1.

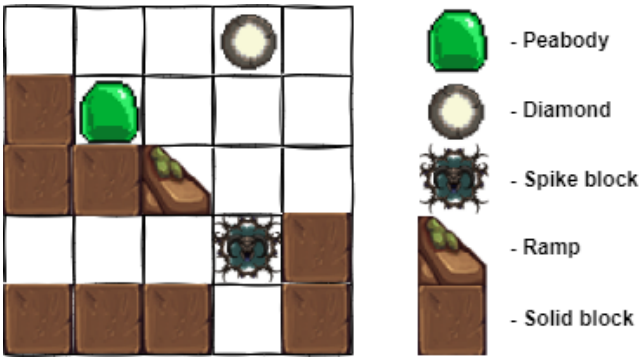

Figure 1: Mock-up of game level

The game is divided into multiple levels, and the main goal in each level is to collect all diamonds by moving the rows using the touch screen. This allows user to either push Peabody to the side or let him fall to the lower rows. When Peabody falls to the lowest row, there is a possibility of falling through the bottom of the screen and appearing at the top (at the same horizontal position) if there is an empty space in the first row. The diagram depicted in figure 3 illustrates how this process looks like.

In this scenario shown in figure 3, the user swipes (towards either direction) a row of bricks/blocks under Peabody, which makes Peabody fall a row lower because the user decides to swiping right when the row is such that there is an opening with no blocks right under Peabody. However, since its the bottom row and the first row has an empty position (an opening with no blocks) in the same column, the fall continues and Peabody appears at the

top of the screen. At this point, because of the existence of a solid block under Peabody, the fall stops and Peabody rests on the second row.

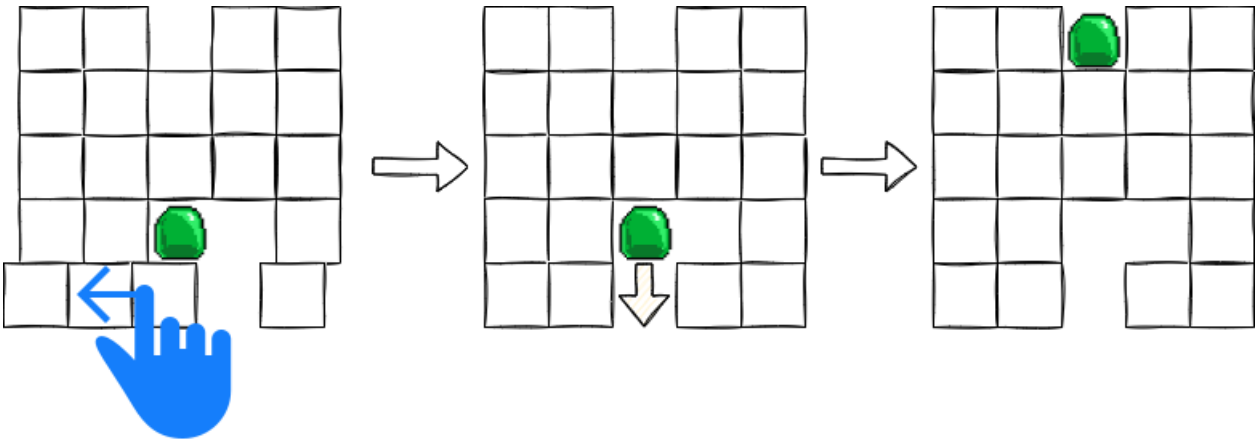

Figure 2: Diagram of falling through the bottom of the screen

The user can also push Peabody through the side of the screen because as any other block, if pushed through the side of the screen, Peabody appears at the vertically opposing side. This process can be described as a two-dimensional rubric cube. For a better understanding, one should look at the diagram in figure 3 as it displays what happens when a row is swiped towards the left side. As we can see, when a blue block reaches a certain point and the swipe continues, the position of the blue block changes from absolute left, to absolute right.

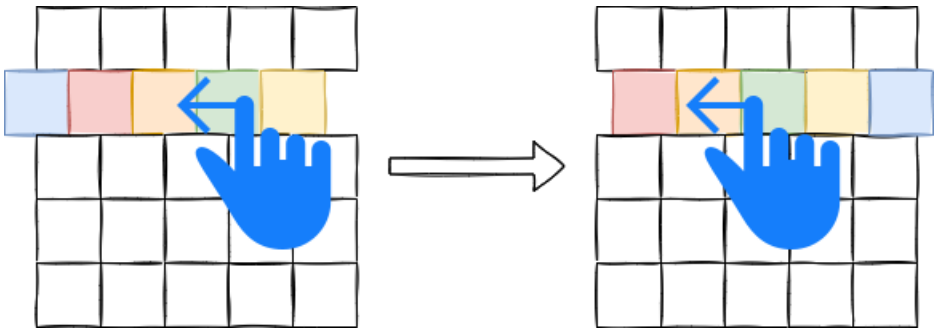

Figure 3: Graphic diagram of swipe process

As mentioned before, the game includes special blocks. The full list of these blocks is given and illustrated in Appendix 0.6. Some blocks have their own unique mechanics. For instance, a Spike block takes life from Peabody (1 life point). The Teleport block changes the position of Peabody from one point to another with a unique animation. Those special blocks and mechanics create a unique experience of a puzzle-solving game.

Technical requirements

As one of the main requirements set up at the beginning of the development process was that the final product should be targeting mobile platforms, more specifically Android platforms for Smartphone devices and Tablets (Android TVs, Smart Watches and other Android

platforms were excluded because their unique characteristics). This brought multiple challenges because of the uniqueness of each android device these days. Devices don't strictly follow a screen ratio of 1:1.86 anymore (Android.com 2022a) and each one has very unique size like Samsung Galaxy fold shown in figure 11. The safe area of the screen isn't equal to the whole screen area anymore. Many smartphone manufacturers have also implemented various processes to lower battery demand like dynamically changing CPU clock rates. The main difference between the "Legacy" version of Peabody and the final product of this development process is the use of a touch screen and that connected functionality where a row follows the finger in real-time during the swipe.

1 Design, Implementation and Testing

Unity, like any other game engine, has its own terminology and way of implementing functions. The entire design and implementation process was altered as a result of these facts.

1.1 Design

In the Unity game engine a game is divided into "Scenes" where each scene represents separate instance of a game, it may for example a game level or main menu. Final version of Peabody consists of 14 of such screens. One for main menu, one for introduction screen after Player starts a new game, and 12 Level screens. Level screens are copies of the first level with only difference being tutorial texts, and level layouts. For In-game menus unity uses a "Panels, for example a panel to display a tutorial information. . The following list gives us description about what is a purpose of each screen and panel. To summarise relation between all menus and scenes implemented in Peabody, we can look on diagram in figure 4.

- Main menu - Let player to Start a new Game or continue where he finished
- About Page - Information about copyrights of used graphics and sounds
- Introduction Screen - After player starts a new game this screen display basic information about the game and how to report a bug in-game
- Game Level - Core scene of the whole game, here player plays the game or access appropriate sub-menus.
- Tutorial - If level includes a tutorial, this panel is displayed at the beginning of the level
- Report Menu - In this menu player can report bug inside the game and his report is sent to Unity Dashboard
- Level Panel - If player dies or successfully finish the level, this panel is displayed to either allow him to continue to next level or tell him that he died and that he needs to restart the level

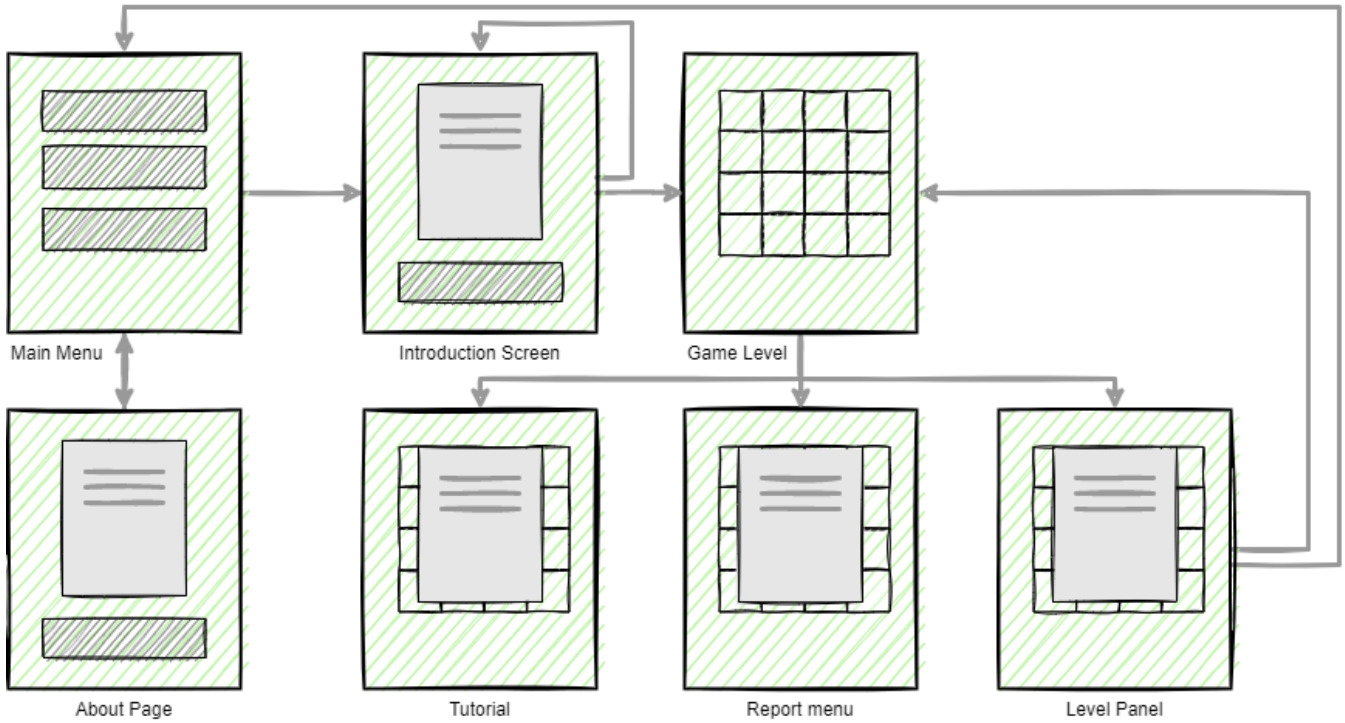

Figure 4: Diagram of scenes and panels in Peabody

When user starts the app, the first screen he sees will be the main menu. From here he can navigate to the ”About page” where he can read Copyright information, or Start a new game. If its not a first time user played this game, the ”Load Game” button is unlocked and transfers the user directly into the highest level archived by the Player. If user decides to start a New Game, he can do by pressing a button ”New Game” which transfers him to the ”Introduction screen” scene. Here he is acquainted with the information about the fact that this game is a University project and what data are collected during the in game report. User also gets information about how to open the ”Report menu” while in the game. After reading all information user can enter the first level by pressing ”Play” button.

When user enter the first level the tutorial panel shows up informing user how to play the game, after pressing ”Continue” button user can start solving the Level. User see a game field which was sized based on HCI research described in previous chapter. During the gameplay user can call ”Report menu” by pressing a button in the top right corner of the screen. This menu allow the user to fill out a form and send the report to the cloud dashboard. If user die or finish the level, the ”Level Panel” shows up and offer to either continue to next level or tell the User that he died and that he needs to restart the level. Every other level follow the same principles.

Main difference between Final version of Peabody and the Swipe prototype is less amount of Panels and difference in reporting of the testing results. As we can see on the Diagram in figure 5 users don’t interact with tutorial or level panels. All in formations were told in the introduction screen where user at the same time agreed with consent form. After finishing each level user would report their in formations with ”Send email” screen where appropriate button triggered native email client and pre-filled with consent form, metric information and

specific blank spaces to be filled out by the user.

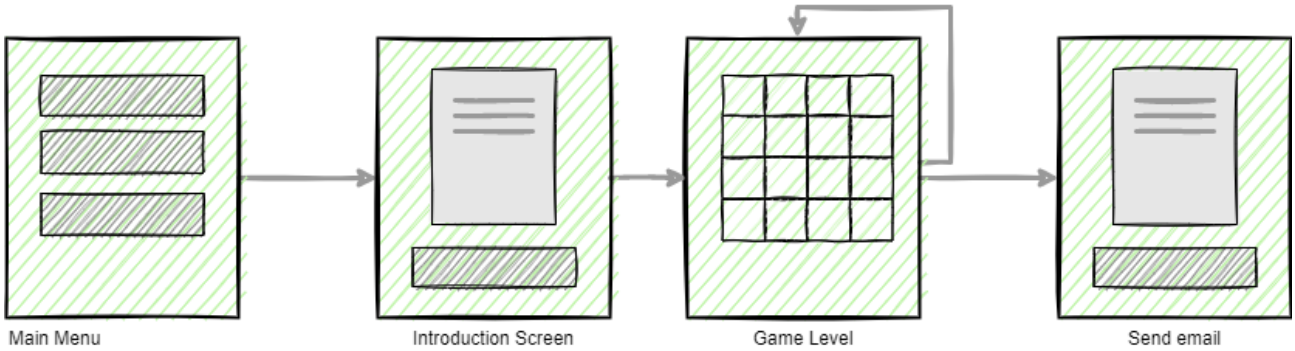

Figure 5: Diagram of scenes and panels in Swipe prototype

1.2 Implementation

Unity engine allows background programming in C# source code files (.cs) called "Scripts" which are later attached to game objects. In the moment when script file is attached to any game object inside the Scene, it allows us to practically manipulate with every aspect of active scene or even close the scene and open another one.

Peabody have in total 8 of such scripts with each one managing specified part of the game. In figure 6 we can see what scripts are attached to Main menu and each Game Level. Apart from Game Levels and Main Menu we have also the "Introduction screen" which includes only "Scene Control" script with purpose of managing change of scenes.

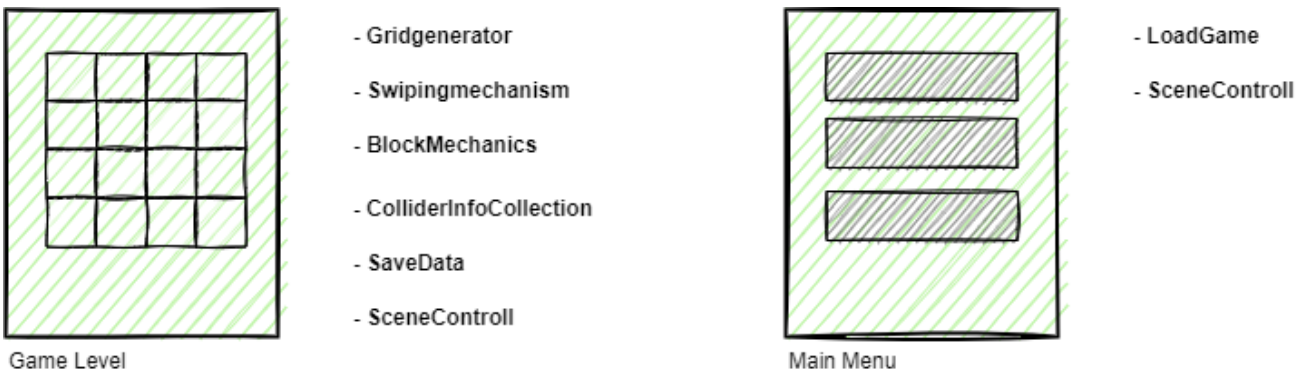

Figure 6: List of scripts included in each scene

Figure 7 shows a UML diagram with a list of variables and methods for each of the primary classes.

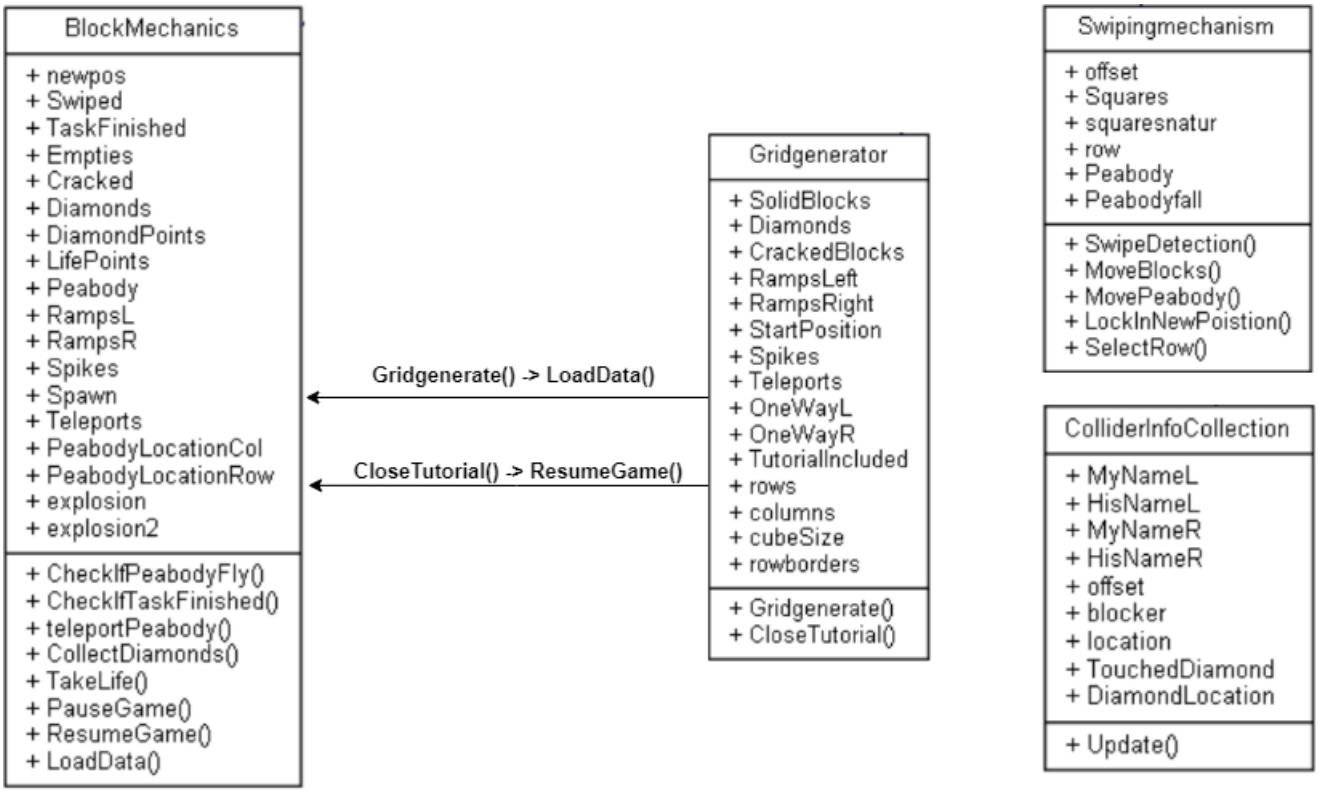

Figure 7: UML Diagram

Grid Generator

Mechanics implemented via ”gridgenerator” script can be divided into two categories. The first category can be called a ”Visual optimisation”. Visual optimisation is a result of deep research and this part of the script ensures that when Game field is generated, all cubes will be sized accordingly based on the size of the safe are of the screen, centred and device will be informed to increase frame rate.

Safe area is an rectangle area in the screen, that does not overlap with display cutouts. A display cutout is an area on some devices that extends into the display surface. It allows for an edge-to-edge experience while providing space for important sensors on the front of the device. (Android.com 2021) In figure 8 we can see device that have a camera inside the display area. The Safe area is marked with the blue colour while the whole screen is both red and blue areas combined. Because of a vast variety of Android devices with each one having cutouts on the display in different area, it is wiser to detect safe area on each device independently and don’t insert important UI elements into the cutout area. In fact, by Android.com (2021) touch sensitivity may be even lower in the cutout area and it is in the best practice to not let the cutout area obscure any important text, controls, or other information.

Another visual optimisation in the ”gridgenerator” script is setting out frame rate of the device. Traditionally, most devices have supported only a single display refresh rate, typically 60Hz, but this has been changing. Many devices now support additional refresh rates such as 90Hz or 120Hz. (Technologies n.d.) Some devices variably change frame rate to increase battery life when higher frame rate isn’t requested, for example while web browsing. A study

Figure 8: Safe area visualised with blue colour

by Goel (2018) researching how battery saving settings affect user experience during the web browsing suggests that low frame rate caused by battery saving settings can degrade the user experience because under low FPS, the page becomes unresponsive to user interactions.

During our development we experienced similar problems before frame rate settings was implemented. Update method used in the Unity development is called every frame, which resulted in very unresponsive gameplay on devices with low frame rate settings. A line ”Application.targetFrameRate = 300” request from the device to set out maximal possible frame rate, which results in the best user experience on each device.

”Gridgenerator” is the initial script that dynamically generates the entire grid of blocks at the start of each level. The script obtains a ”Square” prefab with supplied parameters and changes the size of it dependent on the screen size and grid size. With the engine library command ”Screen.safeArea.width” or ”Screen.safeArea.height,” you may get the screen size, or more accurately the size of the ”Safe area,” which is the region of the screen without cutouts. The size of the grid is determined by how the user configures the number of columns in the editor used during the HCI research phase. As Algorithm 1 explains, the number of rows is determined using the absolute value of ”Height * 1.85”. Size of the block is then defined by algorithm which calculates what number is lower: 85% of screen width divided by number of columns or 85% of the screen height divided by number of rows. This calculation is necessarily because not every device have rectangle shape (for example tablets). With all of the data acquired, the script can start producing the entire grid one block at a time, from top left to bottom right.

Algorithm 1: Algorithm that sets out size of the grid and size of each cube.

Input: HSA - Height of the Safe Area of the Screen
WSA - Width of the Safe Area of the Screen
$Cols$ - Number of Columns defined in Level Editor
$Prefab$ - Predefined Game object that serves as model for Reference block

Output: $Rows$ - Number of Rows
$BlockSize$ - Size of each side of the block
$RefBlock$ - Tile that serves as model block in algorithm 2

```
rows ← |cols × 1.85| ;                    // Number of Rows is calculated
/* Size of the block is defined based on which one of these numbers is lower  */
if ( (HSA × 0.84)/Rows ) < ( (WSA × 0.84)/Cols ) then
    BlockSize = (HSA × 0.84)/Rows
else
    BlockSize = (WSA × 0.84)/Cols
end if
RefBlock ← Instantiate(Prefab) ;   // Reference Block is created with values setted up in Prefab
RefBlock.Size ← Vector2(BlockSize, BlockSize); ;   // Height and Width of Reference Block is set to be equal to BlockSize
```

What type of block will be created at specific position inside of the grid is defined via own created interface in the Unity Engine. This interface is its kind of Level Designer and apart from Tutorials and Level Panel texts its the only difference between the levels. Position is defined by String in following format: row, "&" symbol, and column. In figure 9 we can see how does this Level editor look like. Based on what type of the block is requested at certain position, Game objects with appropriate parameters like Tag, Collision boxes or Texture are created.

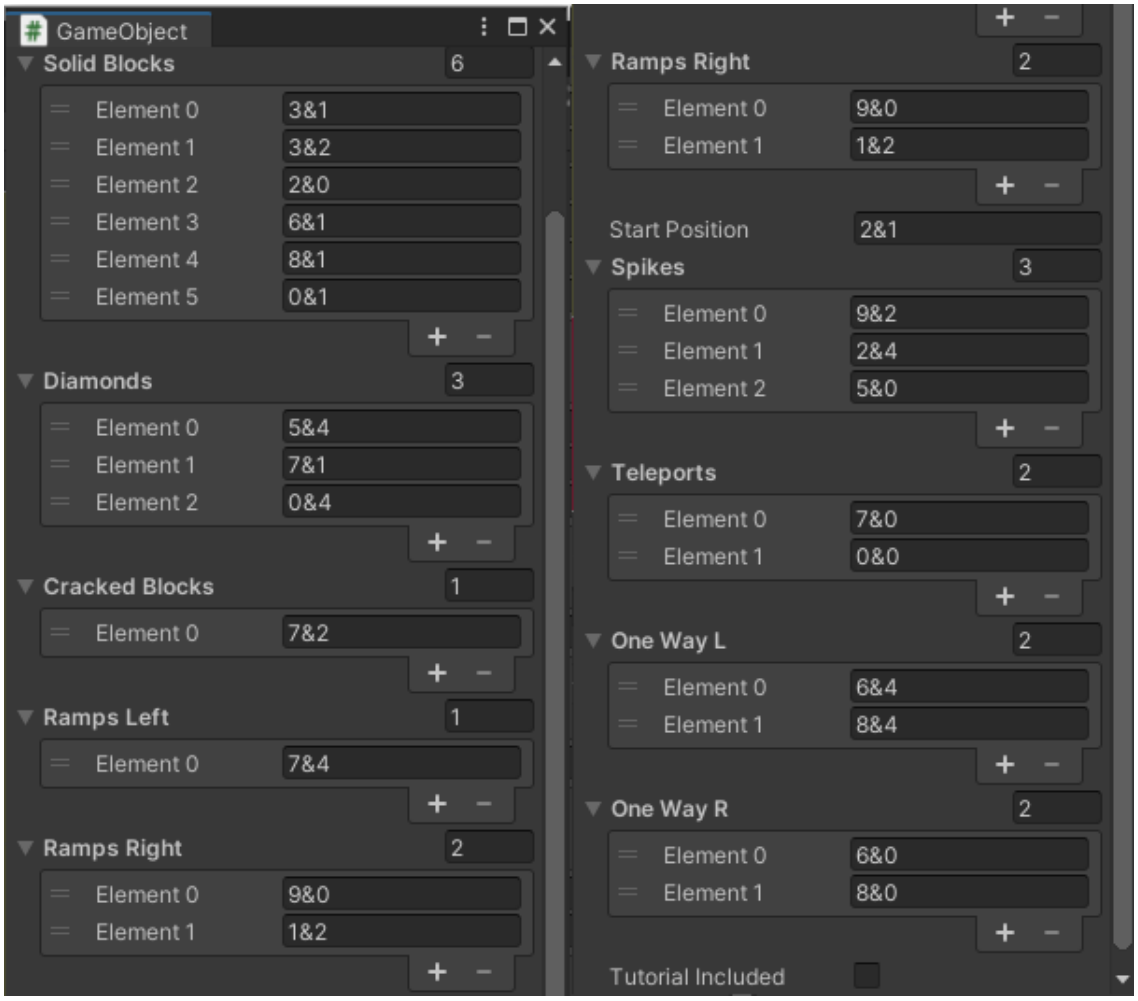

Figure 9: Level Designer interface in the Unity Engine

When the script reaches a specified location in the grid, an instance of the reference block is created and transferred to the correct location, as determined by the Algorithm 2. The block is then given a name depending on its position in the grid, and the script begins to match the block's name to lists created in the Level editor. Algorithm 2 shows how the script would set the required parameters like tag and construct solid block if the same string was included in the list of "Solid Blocks". If the block's name is not found in the list of solid blocks, the script moves on to the next list until it find a match. The script will generate an empty space if the block name is not present in any of the lists. The lists are arranged in order of how frequently each block appears throughout the levels.

Algorithm 2: Algorithm that generates every game block and builds the grid

Input: $Rows$ - Number of Rows
HSA - Height of the Safe Area of the Screen
WSA - Width of the Safe Area of the Screen
PSA - A Vector of X (Horizontal) and Y (Vertical) coordinates of bottom left corner of the screen safe area
$Cols$ - Number of Columns defined in Level Editor
$RefBlock$ - Tile that serves as model block
$BlockSize$ - Size of each side of the block
$SolidBlocks$ - List of strings defining where should be Solid Blocks
$CrackedBlocks$ - List of strings defining where should be Cracked Blocks

Output: $Rows$ - Number of Rows
X - Horizontal Position of the block
Y - Vertical Position of the block
$Block$ - Game Block

```
for i ← 0 to Rows − 1 do
    for y ← 0 to Columns − 1 do
        block ← instantiate(RefBlock) ;          // Create an instance of Reference Block
        X ← (WSA/2 + (PSA.x − Cols × BlockSize)/2 + (y × BlockSize) + BlockSize/2)
        Y ← (HSA/2 + (PSA.y − Rows × BlockSize)/2 − (i × BlockSize) − BlockSize/2)
        block.position ← Vector2(X, Y) ;          // Block is moved to correct location
        block.name ← i + y ;   // Block is named in following format: "Row" & "Column"
        /* If string equal to name of the block is in the list of Solid Blocks    */
        if ( SolidBlocks.contains(block.name)) then
            block.tag ← "Solid" ;                 // Set tag of to block to "Solid"
            block.addComponents(2DCollider) ;                    // Add 2D Collider
            block.Texture ← (SolidBlockTexture) ;                    // Set Texture
        /* If string equal to name of the block is in the list of Cracked Blocks  */
        else if ( CrackedBlocks.contains(block.name)) then
            ... ;   // Same processes as with solid block but different textures etc...
    end for
end for
RefBlock.destroy ;          // Destroy Reference Block since its no longer needed
/* If Tutorial is ticked as "Included"                                          */
if Tutorial == Included then
    TutorialPanel.Set(active) ;                   // Set Tutorial panel as "active"
end if
```

Following the generation of each row, the script saves the Y (Height) coordinates of the bottom corner of each row into an array of floats, which is later used in the Swiping mechanism

script. When the script has finished producing the entire grid, the reference tile is destroyed, and the script checks to see if the Tutorial panel should appear; if it does, the Tutorial Panel is set to ”Active.” Figure 10 shows a Unity engine view of how a new items appeared in the list of Game Objects after the Level started and script generated Game Objects.

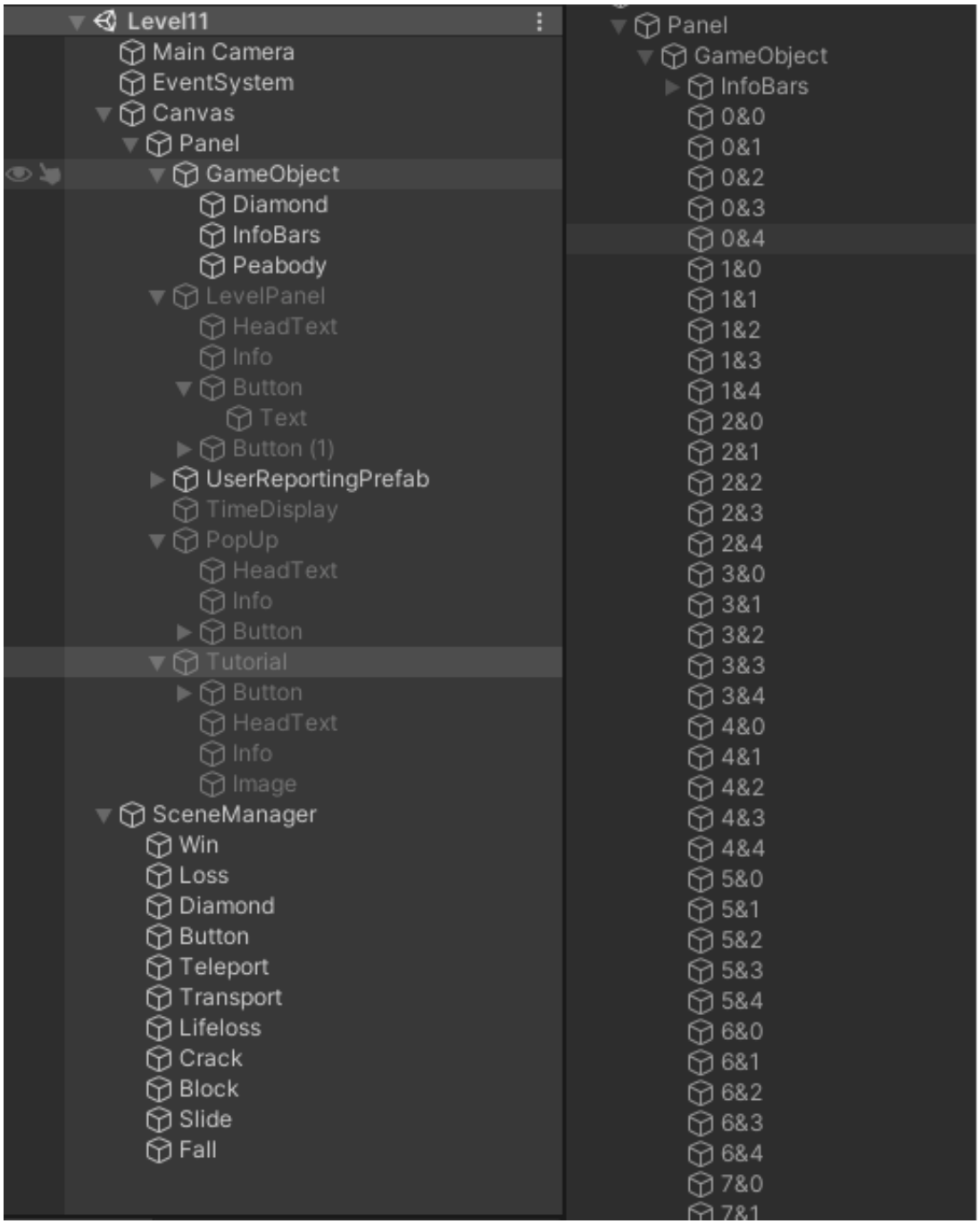

Figure 10: Change in the list of game objects after start of the Level

Block Mechanics

Block Mechanics is a script that takes care about unique mechanics of each block in relation to the Peabody. This script also checks if all diamonds were collected and in such case shows up the Level Panel to inform the player that Level was finished, or in case when Peabody collides with the spike block, this script triggers the Level Panel, displaying that Peabody died and Player needs to try this Level again.

After player finish swiping ”Swiping Mechanism” script informs ”Block Mechanics” that all blocks were moved to appropriate place and saves information about location of Peabody into appropriate variables. This process can be sen in Algorithm 3. If there is no occurring fall at this point, script starts checking what block is under Peabody.

In Algorithm 4 we can see how this process goes. At the first point script loads special game blocks in the level and their position. Then Script checks if there isn’t a teleport at the same position as Peabody. If yes Peabody may be teleported, if no, script proceeds to check what block is under the Peabody. If block under Peabody is for example a cracked block or empty tile, Peabody move onto that position. Empty tiles are coded as a blocks without the texture and colliders, and when Peabody lands on cracked block, this script just erase the collider, make the texture transparent, and change tag of the block. If Peabody’s position is on the lowest row, this script checks what block is in the same column in the highest row. When Peabody’s position change to the next row, script check again what block is under the Peabody, this allows Peabody to fall through multiple rows. If ramp appears under the Peabody, this script also needs to check what block is on the side of the ramp, for example if its a left sided ramp, script check what block is on the left side from the ramp. If there is for example solid block, it means that there is no space where Peabody can slide, and ramp therefore behaves same as a solid block. Special cases that needed to be handled were when Ramp aims to the side of the screen, therefore when Peabody slides through the screen side onto the other side of the row. In that case Script checks if there is an empty space on the opposite side of the screen, therefore on the position where will Peabody land. When Script decides position onto which should Peabody fall, Algorithm 5 start changing position of Peabody until Peabody gets there frame by frame.

Algorithm 3: General layout of Block Mechanics script with algorithm for checking if all diamonds were collected

Input: $Swiped$ - Boolean value of "Swiping Mechanism" reporting tat swipe is finished
PLR, PLC - Row, and Column in which is Peabody located
$PPos$ - A Vector of X (Horizontal) and Y (Vertical) coordinates of current position of Peabody
$PTarget$ - A Vector of X (Horizontal) and Y (Vertical) coordinates where should Peabody Fall
$Completed$ - Boolean value that is set as true when there are no more diamonds to collect
$Diamonds, Cracked, Teleports...$ - Lists of Game objects that have game tag with the same name

Output: $NextRow$ - Row under Peabody

```
   /* If swiped was reported as completed                                    */
1  if ( Swiped ← true) then
2      Diamonds ← GameObjectsWithTag("Diamond") ;   // Reload the list of Diamonds
       /* If the length of the Diamond list is 0 - contain no items          */
3      if ( Diamonds.length ← 0) then
4          Completed ← true ;                          // Sets Completed to true
5      end if
       /* Check if Peabody is not in middle of the fall                      */
6      if ( PPos ← PTarget) then
7          RELOAD_LISTS() ; // Cracked and Teleports and other lists of Game Objects
            are refreshed in a same way as Diamonds on line 2
8          NextRow ← CHECK_NEXT_ROW(PLR) ; // Checking of which row is under
            the Peabody, if Peabody is at the last row, NextRow ← 0

           /* ******************************************************************* */
           /* For this part that manage special blocks please see Algorithm 4    */
           /* ******************************************************************* */
9          PTarget ← (Output of Algorithm 4)
10     else
           /* ******************************************************************* */
           /* For this part that manage fall of Peabody please see Algorithm 5   */
           /* ******************************************************************* */
11     end if
       /* If Alogrithm reported that all diamonds were collected                */
12 else if (Completed ← true) then
13     BlocksGrid.Destroy ;  // Destroy the grid of game blocks at the end of the level
14     LevelPanel.Set(active) ;    // Display panel that states that level was completed
15     BREAK ;                                  // Stops update method for this script
16 end if
```

Teleport is a very unique block in a way of when its triggered. It can be swiped onto Peabody or Peabody can fall to its location. From a code standpoint, when Peabody is right above the Teleport, it behaves in a same way as empty space, therefore Peabody fall to its location. When the location of Peabody and Teleport is same, script checks which teleport is behind the Peabody. If the Teleport behind the Peabody is Blue, (Which is checked by its number in the array of teleports.) Peabody is teleported onto location of the Green Teleport. If Green teleport is behind the Peabody, nothing happens. This behaviour was defined because otherwise Peabody ended in endless loop of teleportations between the teleports. When Teleportation happens, script triggers an animation on location of both teleports to help player focus onto right area.

From perspective of ”Block Mechanics” script, Diamonds behave as empty spaces, and their behaviour is coded in the ”Collider Info Collection” script.

When spike-block appear under the Peabody, the Game over screen is triggered and Player needs to restart the level. In previous versions this script also managed Life Points system where if player touched the spike, one life point was lost, player had 3 life points after which game over screen showed up. This ended up creating problems where if player re-spawned at the start position, rows and columns were already moved into position where Peabody just fallen onto spikes again and ended up loosing all three lives in a row. It was therefore decided to let player restart the whole level right after the first death

In early stages of the development an ”if statement” was used for purpose of checking what block is under the Peabody, but during the optimisation process, the whole code was rewritten and uses now a ”switch statement”. Based on GeeksforGeeks (2017) a Switch statement tends to have better performance when there are more than 5 cases, which correlates with the development process where blocks were added over the time.

Algorithm 4: Algorithm that manage interaction between the Peabody and special blocks

Input: PLR, PLC - Row, and Column in which is Peabody located
$PPos$ - A Vector of X (Horizontal) and Y (Vertical) coordinates of current position of Peabody
$PTarget$ - A Vector of X (Horizontal) and Y (Vertical) coordinates where should Peabody Fall
$Teleports$ - List containing game objects with tag "Teleport"
$Spike, Cracked, RampLeft, EmptyTile, Teleport...$ - Tags of Game objects defining behaviour

Output: $Return$ - A Vector of X (Horizontal) and Y (Vertical) coordinates that is returned from this method
$UnderPeabody$ - Game Object that is located under Peabody

```
/* Check if there is an Teleport in the Level                                  */
if Teleports.length > 1 then
    /* Check if location of the Blue teleport is same as location of Peabody   */
    if Teleports[1].name == (PLR + "&" + PLC) then
        RUN_ANIMAION ;     // Script will call Appropriete Unity Library an run
         animation on position of both teleports
        PPos ← Teleports[1].position ; // Peabody's position is changed to position of
         the Green teleport
    end if
end if
UnderPeabody = find(GameObject.name((PLR + 1) + "&" + PLC)); ; // find block that
 is under Peabody's locaton
/* Based on tag of the block, follow specific case                             */
switch UnderPeabody.tag do
end switch
case EmptyTile do
    Play_Audio(fall) ;                          // Play audio of Peabody falling
    PLR ← PLR + 1 ;                             // Set variable PLR to next row
    return(UnderPeabody.position) ;  // return X.Y position where should Peabody fall
end case
case Cracked do
    Play_Audio(crack) ;                     // Play audio of block being destroyed
    UnderPeabody.tag ← "EmptyTile" ;                    // Set tag to "EmptyTile"
    UnderPeabody.texture ← "Transparent" ;              // Make block transparent
    PLR ← PLR + 1 ;                             // Set variable PLR to next row
    Destroy() ;                                 // Set variable PLR to next row
    return(UnderPeabody.position) ;  // return X.Y position where should Peabody fall
end case
```

```
case RampToLeft do
    /* Standard case with block space on the left side of the ramp                    */
    if PLC > 0 then
        /* If there is an empty space, let Peabody slide                               */
        if find(GameObject.name((PLR + 1) + "&" + PLC − 1)).tag ← "EmptyTile" then
            Play_Audio(slide);                          // Play audio of Peabody sliding
            PLR ← PLR + 1;                              // Set variable PLR to next row
            PLR ← PLC − 1;                          // Set variable PLC to one row left
            return(find(GameObject.name((PLR + 1) + "&" + PLC − 1)).position);
             // return X.Y position where should Peabody slide
        /* Otherwise do nothing                                                        */
        else
            return(newVector2(0, 0));                           // return an empty vector
    /* Case when ramp is on the left side of the screen and Peabody will slide
       through the side of the screen to the right side                                */
    else
        /* If there is an empty space, let Peabody slide                               */
        if find(GameObject.name((PLR + 1) + "&" + 4)).tag ← "EmptyTile" then
            Play_Audio(slide);                          // Play audio of Peabody sliding
            PLR ← PLR + 1;                              // Set variable PLR to next row
            PLR ← PLC − 1;                          // Set variable PLC to one row left
            return(find(GameObject.name((PLR + 1) + "&" + PLC − 1)).position);
             // return X.Y position where should Peabody slide
        /* Otherwise do nothing                                                        */
        else
            return(newVector2(0, 0));                           // return an empty vector
case RampToRight do
    /* ******************************************************************             */
    /* Same logic as with RampToLeft but with modified values                          */
    /* ******************************************************************             */
case Spike do
    Play_Audio(GameOver);                                   // Play Game Over Audio
    GameOverPanel.Set(active);                          // Display Game Over Panel
    BlocksGrid.Set(disabled);                           // Hide the Grid of Blocks
    return(newVector2(0, 0));                            // return an empty vector
case Teleport do
    Play_Audio(fall);                                   // Play audio of Peabody falling
    PLR ← PLR + 1;                                      // Set variable PLR to next row
    return(UnderPeabody.position);  // return X.Y position where should Peabody fall
/* If there is no special block or an empty space under the Peabody                    */
case default do
    return(newVector2(0, 0));                           // return an empty vector
```

Algorithm 5: Algorithm that manage fall of the Peabody onto next row.

Input: $DeltaTime$ - Time between each frame
$PPos$ - A Vector of X (Horizontal) and Y (Vertical) coordinates of current position of Peabody
$PTarget$ - A Vector of X (Horizontal) and Y (Vertical) coordinates where should Peabody Fall

```
/* If vertical position that is set up as target position of Peabody is lower
   thanthe current Peabody's position                                        */
if ( PPos.y > PTarget.y) then
    PPos.y ← (PPos.y − (1500 × DeltaTime)) ;    // Move Peabody's position lower by
     (1500 × DeltaTime)
    /* If position of Peabody is now Lower than the target position          */
    if PPos.y < PTarget.y then
        PPos.y ← PTarget.y ;    // Set Peabody's position Y to be same as the Target
         Position Y
    end if
/* If horizontal position that is set up as target position of Peabody is on the
   right from the current Peabody's position                                 */
else if PPos.x < PTarget.x then
    PPos.x ← (PPos.x + (1500 × DeltaTime)) ; // Move Peabody's position to right by
     (1500 × DeltaTime)
    /* If position of Peabody is now Lower than the target position          */
    if PPos.x > PTarget.x then
        PPos.x ← PTarget.x ;    // Set Peabody's position X to be same as the Target
         Position X
    end if
/* If horizontal position that is set up as target position of Peabody is on the
   left fromthe current Peabody's position                                   */
else if PPos.x > PTarget.x then
    PPos.x ← (PPos.x − (1500 × DeltaTime)) ;  // Move Peabody's position to left by
     (1500 × DeltaTime)
    /* If position of Peabody is now Lower than the target position          */
    if PPos.x < PTarget.x then
        PPos.x ← PTarget.x ;    // Set Peabody's position X to be same as the Target
         Position X
    end if
```

Swiping Mechanism

Swiping Mechanism handles interaction between the user and game via touch screen. More specifically, when user swipes in the area of the block grid. As Algorithm 6 suggeests, the whole process of swipe starts when User touch the screen in the area with blocks. Algorithm 8 tells us that based on the vertical position Y on the screen, script checks which row user wants to swipe. If Y position of touch is higher than the bottom line of a row, script decides that user wants to move with this row and save position of each block. If Y position is higher than bottom line of currently checked row, scripts check another row. Because rows are checked from bottom to the top, there is no need to specify ”if touch is lower than” since script already checked bottom line of lower rows.

Algorithm 6: Main body of the Algorithm managing swipe function

Input: $Input$ - Class from Unity Library that manage touch input
$ColliderInfoCollection.offset$ - Variable in "Collider Info Collection" script that save value of swipe offset at moment when Diamond detects collision
$ColliderInfoCollection.blocker$ - Variable in "Collider Info Collection" script that is set true when Swipe cause collision with the Diamond
$NPCB$ - horizontal position of colliding (with Diamond) block at beginning of the swipe
DP - Horizontal position of the Diamond that reported collision
$BlockSize$ - Size of the block (variable is reported by Gird Generator script)

```
/* If there is at least one finger touching the screen and Peabody is not in
   middle of the fall                                                        */
if ( Input.touchCount > 0 & Peabodyfall ← false) then
    /* ******************************************************************    */
    /* SwipeDetection(offset) - described in Algorithm 8                    */
    /* ******************************************************************    */
else if (Offset! = 0) then
    /* If "Blocker" variable in the "Collider Info Collection" script is true    */
    if (ColliderInfoCollection.blocker ← true) then
        /* If Diamond was touched during swipe to the left                   */
        if ( ColliderInfoCollection.offset > 0) then
            offset ← NPCB − DP − BlockSize
            /* ******************************************************************  */
            /* MoveBlocks(offset) - described in Algorithm 7                    */
            /* ******************************************************************  */
        /* If Diamond was touched during swipe to the right                  */
        else
            offset ← NPCB − DP + BlockSize
            /* ******************************************************************  */
            /* MoveBlocks(offset) - described in Algorithm 7                    */
            /* ******************************************************************  */
        end if
    end if
    /* ******************************************************************    */
    /* LockInNewPoistion() - described in Algorithm 9                       */
    /* ******************************************************************    */
/* If there is no active touch input, reset value of Peabodyfall             */
else if (Input.touchCount ← 0) then
    Peabodyfall ← false
end if
```

When start position of swipe is known and script decided what row is supposed to be moved, the script start tracking current position of the finger. The distance between current position and starting position of the finger on horizontal line X defines an offset by which the

row moves in the real time. In algorithm 7 we can see how are the blocks moved. When offset is higher than half size of block, it means that block on the side of the row is already too far from the grid and "Swiping Mechanism" script move him on the opposite side of the row. For better understanding look back at Figure 3 in Section 5.3 under Game Mechanics. When user release the finger from the screen, script needs to check where to move blocks so they are precisely aligned with the rest of the grid. This process is defined in Algorithm 9 If we would take an example where user swipes to the left, at the moment when user release the finger, script checks which block is currently nearest to the original position of the block on the absolute right and then calculate by how many blocks the whole row moved. After that the blocks are moved onto new positions and renamed by their new location (for example if row is moved by two blocks, block with name 23 become 25).

Algorithm 7: $MoveBlocks(offset)$ - An algortihm for moving the blocks based on position of the finger

Input: $Offset$ - Distance between current and starting position of the finger
$Squares$ - Array containing GameObjects in the row
NPB - Array with horizontal position of blocks at beginning of the swipe
$BlockSize$ - Length of the side of each block

```
/* Move each block in the row according to position of the finger                    */
for i ← 0 to Squares.Length do
    Squares[i].position.x ← (NPB[i].x − offset) ;    // Adjust the vertical position by
     the offset
    /* When the cube on the far left moves out of view, it appears on the far
       right.                                                                          */
    if ( Squares[i].position.x < (NPB[i].x − BlockSize/4)) then
        /* Change the block's position by multiplying the block's width by the
           number of columns.                                                          */
        Squares[i].position.x ← Squares[i].position.x + (BlockSize × Squares.Length)
    /* If the cube on the far right moves out of view, it appears on the far left.
       */
    else if ( Squares[i].position.x > (NPB[Squares.Length − 1].x + BlockSize/4)) then
        /* Deduct the width of the block multiplied by the number of columns to
           change the block's position.                                                */
        Squares[i].position.x ← Squares[i].position.x − (BlockSize × Squares.Length)
    end if
    /* ******************************************************************************  */
    /* MovePeabody() - described in Algorithm 10                                       */
    /* ******************************************************************************  */
end for
```

Algorithm 8: $SwipeDetection(offset)$ - Algorithm that initialise swipe when finger touch the screen

Input: $Input$ - Class from Unity Library that manage touch input
Output: NPB - Array with horizontal position of blocks at beginning of the swipe
NPP - position of Peabody at beginning of the swipe
$SwipeStart$ - position of finger at beginning of the swipe
$Squares$ - Array wit Blocks in the swiped row

```
if ( Input.Position is inside the grid area) then
    /* If this is the first frame when touch was detected                    */
    if ( Input.Phase ← Began) then
        SwipeStart ← Input.position NPP ← Peabody.position
        i ← (LowestRow)
        while Input.position.y > bottom of the row "i" do
            i ← (i − 1)
        end while
        Squares ← GameObjects at row "i + 1" NPB ← Squares.position
    /* If touch continues                                                    */
    else if Input.Phase ← Moved or Input.Phase ← Stationary then
        offset ← SwipeStart.x − Input.position.x
        if ( If no Collision with Diamond or Peabody dont levitate) then
            if ( Squares doesnotcontains GameObjects with tag(OneWayR) and
              tag(OneWayL)) then
                /* MoveBlocks(offset) - described in Algorithm 7             */
            else if ( Squares contains GameObject with tag(OneWayR) which is not trying
              to get through the Left side of the screen) then
                /* MoveBlocks(offset) - described in Algorithm 7             */
            else if ( Squares contains GameObject with tag(OneWayL) which is not trying
              to get through the Right side of the screen) then
                /* MoveBlocks(offset) - described in Algorithm 7             */
            else if ( Squares contains both GameObject with tag(OneWayR) and
              GameObject with tag(OneWayL) niether of which is trying to get through the
              opposite side of the screen) then
                /* MoveBlocks(offset) - described in Algorithm 7             */
            else
                if Didnt reported blocked swipe yet then
                    PLAY_AUDIO(Block) ;              // Play audio of blocked swipe
                end if
            end if
        else if collision but retracting swipe then
            /* MoveBlocks(offset) - described in Algorithm 7                 */
        else if Peabody Levitate then
            /* LockInNewPoistion() - described in Algorithm 9                */
        end if
    end if
end if
```

Algorithm 9: $LockInNewPoistion()$ -Algorithm that saves position of Blocks when swipe is finished

Input: $Squares$ - Array wit Blocks in the swiped row
NPB - Array with position of blocks at beginning of the swipe

```
/* If player moved a row left                                          */
if ( Offset > 0) then
    for i ← 0 to Squares.Length − 1; i ← i + 1 do
        integer move ← i − (Squares.Length − 1)
        for n ← Squares.Length − 1 to n > −1; n ← n − 1 do
            /* move the first square from absolute left to absolute right  */
            if n + move > −1 then
                Squares[n].position ← NPB[n + move]
                Squares[n].name ← (row + "&" + (n + move))
            /* Move square "i" to the postion of square one block left     */
            else
                Squares[n].position ← NPB[Squares.Length + move]
                Squares[n].name ← (row + "&" + (n + move))
            end if
        end for
    end for
    offset ← 0
    Peabody.position.x ← NewPositionPeabody.x
    PeabodyStartOffset = 0
    NewPositionPeabody ← null
    BlockMechanics.Swiped = true ;    // Send info to Block Mechanics that swipe was
     finished
/* If player moved a row right                                         */
else if ( Offset < 0) then
    /* ******************************************************************  */
    /* Same logic as with Offset > 0 but with modified values              */
    /* ******************************************************************  */
```

Algorithm 10: $MovePeabody()$ - Algorithm that manage movement of Peabody during the swipe

Input: $Squares$ - Array wit Blocks in the swiped row
CBL, CBR - Block Coliding with Peabody from the Left side (CBL), and from the Right side (CBR)

```
if (ColliderInfoCollection.CBL != null) and (offset < 0) then
    if ColliderInfoCollection.CBL.name contains Active Row then
        PeabodyOffset ← offset − PeabodyStartOffset ;    // Peabody Start offset is recorded in Collider Info Collection in the moment of collision. Its offset of swipe in the moment of collision
        if NewPositionPeabody != null then
            if PeabodyOffset > 1 then
                Peabody.position = Position of Peabody Before Collision ; // Is recorded in Collider Info Collection in the moment of collision
                NewPositionPeabody ← null PeabodyStartOffset ← 0
            else
                Peabody.position ← NewPositionPeabody
            end if
        else if PeabodyOffset < 0 then
            NewPositionPeabody ← CBL.name + 1 column
        end if
    end if
else if (ColliderInfoCollection.CBR != null) and (offset > 0) then
    /* ******************************************************************    */
    /* Same logic as with Collision from the left but with modified values    */
    /* ******************************************************************    */
end if
```

If Peabody is in the row being swiped, a separate process is triggered in cooperation with "Collider info Collection" script. As we can see in Algorithm 10, "Collider info Collection" checks if Peabody collides with a block by side that is in the opposite to the direction of swipe. For example If we swipe to the right, this script checks if any block touches the Peabody from the left, therefore if Peabody is being pushed. If not Position of the Peabody isn't affected by swiping. This function allows user for example to swipe teleports onto Peabody. In same way script checks if there is an Diamond in the row that is swiped and if there is an block touching the Diamond. If yes, the swiping is blocked since diamonds cant be moved. Blocking is also implemented if One-way block get to to the side of the screen and try to get through while his arrows point to the opposite direction. For example if One-way block with arrows to the left try to pass through the right side of the screen.

Another process that is triggered if Peabody in the row that is being swiped is a process checking if there isn't an empty space (or other block where Peabody can fall) under Peabody while Peabody moves. This process prevents user from pushing Peabody over spaces where would Peabody usually fall, and therefore preventing users form cheating the game with flying.

Collider Info Collection

Collider Info Collection is a script highly cooperating with other scripts. Its an implementation of 2D physics engine and collects information from colliders. This script is the only script that appears multiple times in the Game Level because it needs to be attached to every diamond and to Peabody, to collect information about the collisions. In general collision reporting is divided into 4 methods where Collision information are either saved or removed from variables. ”OnTriggerEnter2D” is supposed to report information about collision in the first moment of collision. However Unity Engine tends to not report start of Collision and therefore OnTriggerStay2D method was implemented to check ongoing collisions every frame. ”OnTriggerExit2D” is a method that reports that collision ended and erases data about colliding game objects. The last method that works with functions from Unity 2D Physics engine is a custom written use of ”Physics2D.OverlapCollider” this checks once per frame what objects colliding with the main game object (the object to which is the script assigned) even though it should have been already reported by other methods, this method tends to report new information and helped to solve many problems.

Save Data

Script for ”Save Data” was more developed for Swipe prototype where time of how long it took player to finish each level was recorded and at the end contained method for sending an pre-defined email via standardised application API (Application.OpenURL(”mailto:...). In Final version of Peabody the role of ”Save Data” script is to Load next level, and save information about what was the last accessed (Unfinished) Level into registry called ”PlayerPrefs” for later access from ”Load Game” script. On Android, PlayerPrefs are stored in ”/data/data/pkg-name/shared_prefs/pkg-name.v2.playerprefs.xml”. Unity stores PlayerPrefs data on the device, in SharedPreferences. (Unity Technologies 2022)

Load Game

When user opens the Main menu, the ”Load Game” script is triggered. If there is an information in registry about last played level, the ”Load Button” becomes intractable and let user Load last accessed (Unfinished) Level.

Scene Control

Scene control is used in Main menu and Form screen as a script with a functions that allows user to change screen via buttons. That means when user press the ”New Game” button, the Form is opened via this script. From there user use same script for accessing the first level via ”Start” button.

Internal Testing and Deployment

As mentioned in the Development methodology chapter 5.1 the whole development was conducted in weekly springs of which was result a prototype with new functions. To be able to

test the new version properly it was important to compile whole game into files that can be deployed on android device. At early stages game was compiled into .apk files and directly installed on the Android device. To compile files for android devices a Android SDK NDK Tools are needed. These tools are offered to be installed during the unity engine installation. At the point when prototype is ready to be deployed, developer needs to set up parameters of the deployment. For .apk file that is going to be deployed directly on Android device, a debug mode is sufficient. Debug mode however require from the owner of the device to enable installation of .apk files from unknown sources.

Because we wanted our game to be tested by people who may not trust .apk file from unknown source for example out of fear that it may contain a virus, we needed to deploy our game to the Google Play market which is a trusted source of Android apps. which requires long approval process. For that reason we needed to set out our deployment for Google Play store in an early stage.

Publishing a game to Google Play store required various changes. For example new apps and app updates must target Android 11 (API level 30) or higher because every new Android version introduces changes that bring security and performance improvements and enhance the Android user experience. Some of these changes only apply to apps that explicitly declare support through their "targetSdkVersion" manifest attribute (also known as the target API level). This doesn't mean that our game will be able to be played only on devices with newest Android version. Configuring the app to target a recent API level ensures that users can benefit from these improvements, while the app can still run on older Android versions. (Android.com 2022b)

Another requirement is that the game is signed with a private key that is managed by Google Play. By Google (2022b) Android apps are signed with a private key to ensure that app updates are trustworthy, every private key has an associated public certificate that devices and services use to verify that the app update is from the same source. Devices only accept updates when their signature matches the installed app's signature. By letting Google manage your app signing key, it makes this process more secure.

After each update Google Play checks that uploaded game is signed with proper key and is automatically tested as part of the pre-launch report. After the upload of the game to the Google Play, it's installed on a set of Android devices in their test lab. After that its automatically launched and through automatised scripts crawled for several minutes. The crawler performs basic actions such as typing, tapping and swiping. This testing is to find issues with Stability, Android compatibility, Performance, Accessibility, Security vulnerabilities and Privacy (Google 2022a).

When pre-launch report is finished, the option to publish the update is unlocked and update can be deployed to Android devices. As part of the testing process game was deployed in open-testing mode. In this mode no personal information are required from the user, however game is accessible on the Google Play store only via URL link. Updates were published weekly to allow testing of new functionalities, fast spotting of bugs and checking with Professor Martyn Amos that game is following the general idea.

1.3 Testing

There are several techniques used for the game testing, where each technique focus on a different part of the game, like functionality or Compatibility. For example, Starloop Studios (2020) sets 5 different testing techniques. Ad-Hoc Testing, Functionality Testing, Compatibility Testing, Progression Testing, and Regression Testing. For testing Peabody we created a testing strategy with all techniques being implemented in the testing process

Approach

Ad-hoc testing can be described as a way of testing where the tester casually plays through the whole game, but with a focus on finding problems with the game (bugs). This technique was the core of our testing strategy, each player played through the game and then reported his experience in the provided google form. Functionality testing was covered as part of Ad-Hoc testing because in part of the game will be explained what is the expected behaviour of each box as a sort of tutorial. Therefore it's expected that users will also report any behaviour non-described in those "tutorials". Compatibility testing was for a large part conducted as part of the development process, as described in previous sections. However, if users experienced any incompatibility problems, there was a chance to report those problems inside the google forms document. Unity cloud reporting was also implemented in the game, where if the game crashed, all information was automatically and anonymously reported into the Unity Dashboard.

Compatibility

In terms of optimal compatibility, it was expected that if device owned by the user was declared as supported one (for example Android TVs and Google watches were excluded), user was able to download the game from the Google Play store and run it without crash of the game. It was also expected that all interfaces and game areas will be always inside of the screen safe area, even in cases of unique screen sizes like Samsung fold. In figure 11 we can see the Samsung galaxy fold with a safe area marked by yellow colour.

Performance

In the performance part of testing, users were asked various questions to rate the performance. For example, how would they rate speed of loading of game levels, or how would they rate responsiveness while swiping.

Testing of performance is the most complicated one because users can report the game as slow even though the game works as intended. However this is part of the user experience and if users feel like the game should be more swift and for example, Peabody should fall way faster, its and important input from the user side, and if the majority of the users have the same feeling, the game should be optimised accordingly.

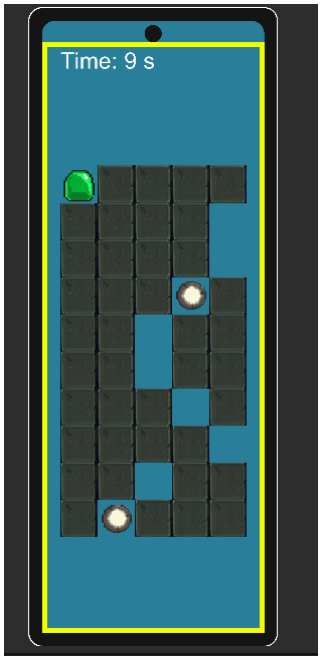

Figure 11: Samsung Galaxy Fold with its unique display proportions

Functionality

Game included tutorials explaining the game mechanics and expected behaviour of each block and as part of functionality testing, users were encouraged to report in-game behaviour that was not specified in those tutorials. For example tutorials specified that Diamond cant be moved, if there would be case where user managed to move the diamond, it would be an unexpected behaviour and user should have report it. Users were also encouraged to report GUI problems, like problems in-game button. In-game reporting functionality was implemented mainly for this area of testing because it allowed developer to see what was happening at the exact moment, however users could reported these problems also in the google form.

User reporting

Users were provided with a google form document which will included a consent form. Users were asked various questions covering previous mentioned areas. If user found a problem with the game, he had an option to report this problem directly in the game, or later in the google form doc. Users were encouraged to report problems already in the game. This reporting was possible thanks to the Unity cloud reporting tool where users had the option to send a report of found problems by pressing a report button which triggered the form for reporting of found bugs. All necessarily metrics and screenshots were included in this report while keeping users anonymous, this allowed in-game reporting also for people who did not wish to participate in the research but wanted to enjoy Peabody just as any casual game and contribute to its development.

1.4 Testing results

Public testing was conducted from 1st of April until the 8th of April. During this period 8 people decided to fill out the participation sheet while data from Google play console indicate that 12 users downloaded the Game. Participants were using their own android devices which helped with variety of devices being used and after finished game reported their answers in provided google form. Game was possible to download from google play store via link that was on the front page of the google form and users were able to watch game trailer that was on the first page of online google form and decide if they want to participate. Having more participants in this testing process would be more insightful and in multiple cases it would be beneficial to collect contact information from the users to be able to reflex on their feedback.

Compatibility

First category of question was focused on Compatibility. When we look at summarised data from the forms, in figure 12 we can see that most common Android system used by participants was Android 11 while the lowest version of Android used by participants was Android 10. Lowest supported Android version of Peabody was Android 8 which none of the Participants reported to use. Models of the Phones reported by the users varied but in general most common brand was Samsung with second one being Xiaomi.

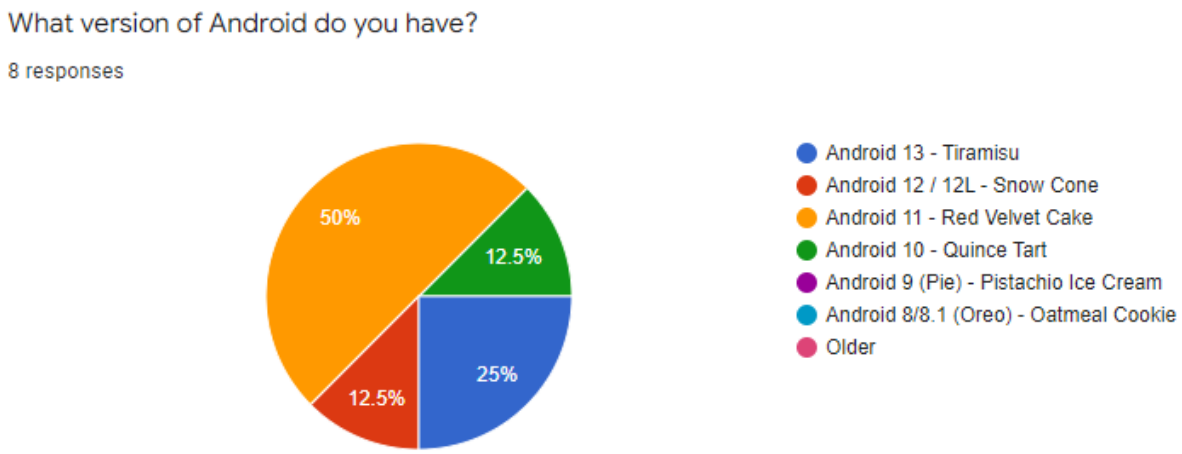

Figure 12: Version of Android system reported by participants

One of the compatibility-related questions was regarding downloading and installing the game via the Google Play store. The Google Play store handles the entire process of downloading and installing the game, so anyone with a compatible version of Android should be able to acquire Peabody from there. If there would be reports of people with appropriate Android versions who are unable to download the game, it would mean that there are issues with the configuration of Google Play store listing.

There were no issues with compatibility reported by any of the participants. This includes game crashes, game elements appearing outside of the screen, or installation issues. Partic-

ipants were given fields to fill out with further information in case of answering yes to any of the questions.

Performance

Participants were asked to score the game's performance on a scale of 1 to 5 in the second section of the questionnaire. The first question asked participants to rate how long it took for the game to load the levels 7 out of 8 users rated loading time as "Fast" with a score of five, whereas one user rated the loading time with score four.

Question asking participants to judge the game's responsiveness to swipe gestures, elicited a wide range of responses. 4 out of 8 participants gave a score of 5 (Fast) for swiping response, 3 participants provided a score of 4, and one participant gave a score of 3 out of 5. Swiping has not been noted as being "very slow" by any user.

We modified the scale for the question where participants were asked to rate Peabody's falling speed from 5 to 1, with 5 being "Too Fast" and 1 being "Too Slow." As a result, the best speed should be ranked at 3. Because falling animation is based on frame rate, the frame rate of the user device had the greatest visual impact on this phase of the game, which was discovered during internal testing throughout the development process. The speed of falling was assessed as a 3 by half of the subjects. Two participants awarded the speed a score of 4, and two others said it was too fast. This grade can be based on a specific device or, as described in the Performance section of the Testing chapter, it might be based on individual expectations of how fast would they want Peabody to fall.

As shown in figure 13, the majority of the participants gave the performance a 5 out of 5 rating, with only 1 participant giving a 4 rating, which is considered good. This suggests that even those who didn't give the game the highest grade for swiping response or falling speed believed that the game performed well in general. In response to a non-mandatory question, one participant wrote, "Very satisfied performance wise."

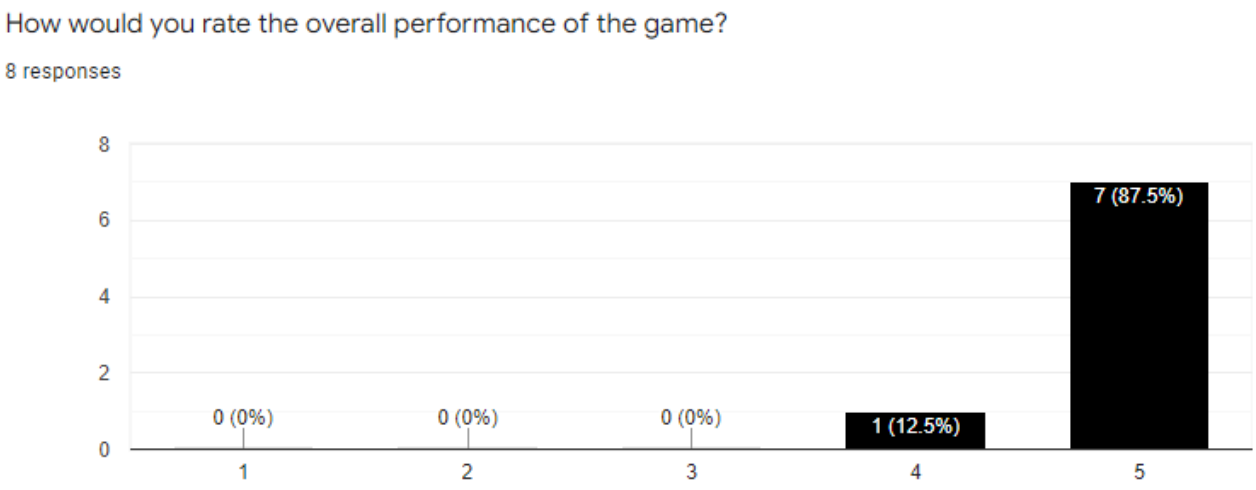

Figure 13: Participants rating of overall performance

Functionality

Questions in Functionality category were focused on reporting if user experienced any behaviour considered by him as non standard. In the first question we asked participants about problems with UI elements, for example that some buttons were not working properly, and no participant experienced such problems.

When we asked participants if they experienced any improper behaviour of game blocks, one participant reported that he experienced a problems but didn't reported them inside the game. In voluntarily description window he wrote "Block appeared within another block after I moved it". Because report wasn't send inside the game, its hard to find out what exactly happened, which was the main reasons why users were encouraged to do in-game reports at the start of the game. The problem experienced by the participant may be the problem described in the development evaluation speaking about problems with physics engine.

In last question from the performance section we asked participants to report if they experienced any problems with game mechanics, for example if they were in situation where swiping doesn't work. One participant reported that he experienced an improper behaviour and filled in the answer that he reported this problem with in-game reporting system. However after checking the Unity Dashboard we've found no report matching with this problem. In description participant wrote us the following: "Sometimes Peabody does unexpected abrupt movements". This participant may miss-clicked between positive question that states that he reported this problem instead of the positive answer that states that he didn't reported this problem via unity in-game reporting. Without possibility to contact the participant it was impossible to recreate this situation and therefore do repairs.

Gameplay

Final chapter of the questionnaire was about participants rating of the game experience. Which was in overall very positive. All participants rated their game experience with highest score.

Graph in figure 14 shows rating of participants in how much they find game mechanics interesting. This can tell us how much were the game mechanics innovative and different from other games on the market. Seven out of eight participants rated mechanics as "perfectly fitting the game".

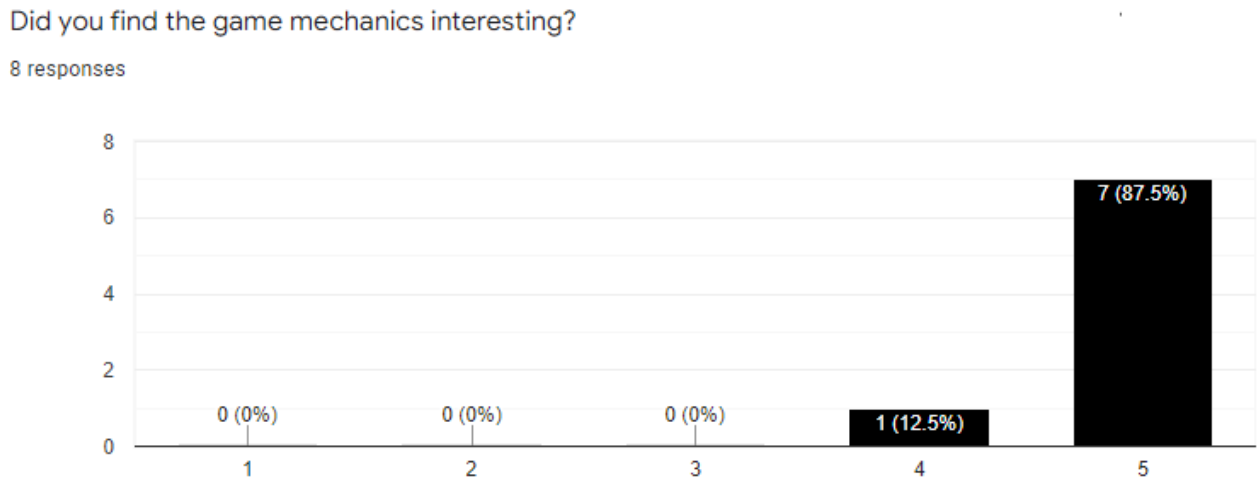

Figure 14: Answers to question about the game experience

When we asked participants which level was their favourite, seven out of eight said the last one, which they regarded as "tricky but engaging." One user said his favourite level was Level 6, where spikes were introduced to the game, bringing much-needed challenge to the puzzles in his perspective.

We received three types of responses when we asked if participants would modify anything in the game. Some participants claimed they wouldn't change anything in the game, while others said they'd like to see more levels, and one participant wrote in to say he'd like to see problem patches.

We asked participants if they would suggest the game to their friends in the last question of the survey. This question was answered affirmatively by all participants. This has a strong correlation with user ratings of overall game experience and is very certainly related.

1.5 Product evaluation

This section evaluates the final product and how it met given requirements.

Game Mechanics

From standpoint of Game Mechanics, all requested mechanics were implemented with proper behaviour apart from reported bugs in the testing process which because of stated reasons was not possible to replicate and therefore fix.

All types of blocks based on given specifications were implemented and all participants were able to play through the whole game and successfully finish it.

Compatibility

Based on user reports from testing process no user experienced compatibility problems and whole game was very well optimised to run across all android phones with various graphics

APIs like Vulkan or OpenGL. There was no report of users experiencing problems with screen size or game crashes.

References

Android.com (2021), 'Support display cutouts | android developers'. Available at: https://developer.android.com/guide/topics/display-cutout (Accessed: 24 April 2022).

Android.com (2022a), 'Declare restricted screen support'. Available at: https://developer.android.com/guide/practices/screens-distribution (Accessed: 24 April 2022).

Android.com (2022b), 'Meet google play's target api level requirement'. Available at: https://developer.android.com/google/play/requirements/target-sdk#:~:text=New%20apps%20and%20app%20updates (Accessed: 24 April 2022).

GeeksforGeeks (2017), 'switch vs if else - geeksforgeeks'. Available at: https://www.geeksforgeeks.org/switch-vs-else/ (Accessed: 24 April 2022).

Goel, U. (2018), 'Does android's battery-saver degrade the mobile web experience?'. Available at: https://developer.akamai.com/blog/2018/05/29/does-androids-battery-saver-degrade-mobile-web-experience (Accessed: 24 April 2022).

Google (2022a), 'Use a pre-launch report to identify issues - play console help'. Available at: https://support.google.com/googleplay/android-developer/answer/9842757?hl=en-GB (Accessed: 24 April 2022).

Google (2022b), 'Use play app signing - play console help'. Available at: https://support.google.com/googleplay/android-developer/answer/9842756?hl=en-GB (Accessed: 24 April 2022).

Hyptosis (2022), 'Tile art'. Available at: https://hyptosis.newgrounds.com/news/post/793230 (Accessed: 24 April 2022).

Starloop Studios (2020), 'Game testing 101: Basic tips and strategies | starloop studios'. Available at: https://starloopstudios.com/game-testing-101-tips-and-strategies/ (Accessed: 24 April 2022).

Technologies, U. (n.d.), 'Android game development | unity'. Available at: https://unity.com/solutions/mobile/android-game-development (Accessed: 24 April 2022).

Unity Technologies (2022), 'Unity - scripting api: Playerprefs'. Available at: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (Accessed: 24 April 2022).

0.6 Block specifications

Table 1: Solid block

Peabody Landed on	standard behaviour
Pushed from side	standard behaviour
Interaction with other block	standard behaviour
Pushed from the screen	standard behaviour
Special function	standard behaviour

Table 2: Cracked block

Peabody Landed on	cracks and disappears
Pushed from side	standard behaviour
Interaction with other block	standard behaviour
Pushed from the screen	standard behaviour
Special function	standard behaviour

Table 3: Spike block

Peabody Landed on	takes life
Pushed from side	standard behaviour
Interaction with other block	standard behaviour
Pushed from the screen	standard behaviour
Special function	standard behaviour

Table 4: Ramp block

Peabody Landed on	cracks and disappears
Pushed from side	standard behaviour
Interaction with other block	standard behaviour
Pushed from the screen	standard behaviour
Special function	standard behaviour

Table 5: One-way block

Peabody Landed on	standard behaviour
Pushed from side	standard behaviour
Interaction with other block	standard behaviour
Pushed from the screen	Can be pushed only through one side
Special function	standard behaviour

Table 6: Teleport block

Peabody Landed on	Peabody teleport elsewhere
Pushed from side	standard behaviour
Interaction with other block	standard behaviour
Pushed from the screen	standard behaviour
Special function	Peabody teleport to location of second teleport

Table 7: Diamond block

Peabody Landed on	Diamond disappear
Pushed from side	standard behaviour
Interaction with other block	cant be moved
Pushed from the screen	standard behaviour
Special function	Is collected by Peabody

Source of images: (Hyptosis 2022)

0.7 Version History

Release history

Release	Version code	Released
30 (1.3.0)	30	1 Apr 2022 17:25
29 (1.2.9)	29	1 Apr 2022 13:25
28 (1.2.8)	28	31 Mar 2022 16:53
27 (1.2.7)	27	31 Mar 2022 15:58
26 (1.2.6)	26	30 Mar 2022 20:22
25 (1.2.5)	25	29 Mar 2022 17:10
24 (1.2.4)	24	22 Mar 2022 17:38
23 (1.2.3)	23	15 Mar 2022 21:01
22 (1.2.2)	22	8 Mar 2022 17:18
21 (1.2.1)	21	21 Feb 2022 16:41

Release history

Release	Version code	Released
20 (1.2.0)	20	12 Feb 2022 12:41
19 (1.1.9)	19	8 Feb 2022 21:09
18 (1.1.8)	18	7 Feb 2022 13:07
17 (1.1.7)	17	5 Feb 2022 23:03
16 (1.1.6)	16	5 Feb 2022 21:34
15 (1.1.5)	15	5 Feb 2022 20:49
14 (1.1.4)	14	5 Feb 2022 19:49
13 (1.1.3)	13	1 Feb 2022 20:05
12 (1.1.2)	12	31 Jan 2022 06:51
11 (1.1.1)	11	26 Jan 2022 12:10

Release history

Release	Version code	Released
10 (1.1.0)	10	26 Jan 2022 11:52
9 (1.0.9)	9	25 Jan 2022 21:46
8 (1.0.8)	8	19 Jan 2022 11:22
7 (1.0.7)	7	18 Jan 2022 14:36
6 (1.0.6)	6	10 Jan 2022 19:23
5 (1.0.5)	5	10 Jan 2022 16:30
4 (1.0.4)	4	10 Jan 2022 12:40
3 (1.0.3)	3	18 Dec 2021 14:55
1 (1.0.2)	1	13 Dec 2021 16:59

0.8 Source code - Gridgenerator

C:\Users\Dominik\Peabody\Assets\Scripts\Gridgenerator.cs 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;



/* Gridgenerator.cs
 * This file contain methods for generating of whole gamefield,
 * ensuring that its scaled and centered in middle of the screen according to used device
 * Frame rate is also set out in this script
 *
*/

public class Gridgenerator : MonoBehaviour
{




    //public float ColumnsEnter;
    //public int LevelNumberEnter;

    // Following variables are for purpose of editing in the Engine GUI
    [Tooltip("Postition of solid blocks in format Row&Column")]
    public List<string> SolidBlocks;
    [Tooltip("Postition of diamonds in format Row&Column")]
    public List<string> Diamonds;
    [Tooltip("Postition of cracked blocks in format Row&Column")]
    public List<string> CrackedBlocks;
    [Tooltip("Postition of ramps with slide to the left side in format Row&Column")]
    public List<string> RampsLeft;
    [Tooltip("Postition of ramps with slide to the right side in format Row&Column")]
    public List<string> RampsRight;
    [Tooltip("Spawn position of Peabody in format Row&Column")]
    public string StartPosition;
    [Tooltip("Postition of spike blocks in format Row&Column")]
    public List<string> Spikes;
    [Tooltip("Postition of teleports in format Row&Column")]
    public List<string> Teleports;
    [Tooltip("Postition of blocks that allows to swipe only to the left in format Row&Column")]
    public List<string> OneWayL;
    [Tooltip("Postition of blocks that allows to swipe only to the right in format Row&Column")]
    public List<string> OneWayR;
```

```
    [Tooltip("Show tutorial panel at the start of the level")]
    public bool TutorialIncluded;

    //variables used for storing information about game field  (must be static)
    public static float rows;
    public static float columns;
    public static float cubeSize;
    public static float[] rowborders;

    private void Update()
    {
        if (Input.GetKey(KeyCode.Escape))
        {
            UserReportingScript other = (UserReportingScript) GameObject.Find("/Canvas/Panel/UserReportingPrefab/UserReporting").GetComponent(typeof(UserReportingScript));
            other.CreateUserReport();
        }
    }


    // Start is called before the first frame update
    void Start()
    {
        Application.targetFrameRate = 300;
        Gridgenerate();

    }

    public void Gridgenerate()
    {


        // prevent wrong values
        if (columns < 5)
        {
            columns = 5;
        }


        //calculate number of rows depending on the number of columns
        rows = columns * 1.85f;

        //set cube size so it fits screen either by width or height (smaller number of those two)
        cubeSize = Mathf.Min((Screen.safeArea.height * 0.84f) / rows, (Screen.safeArea.width * 0.84f) / columns);

```

```
        //set size of array containing position of bottom line of each row
        rowborders = new float[(int)rows + 1];

        //Load reference block from the prefab and generate it inside of the Level
        GameObject referenceTile = (GameObject)Instantiate(Resources.Load("Square"));

        //set size of reference tile based on CubeSize calculated in previous rows
        referenceTile.transform.localScale = new Vector2(cubeSize, cubeSize);

        // position bug report button
        PositionReportButton();

        // create Diamonds bar
        GenerateDiamondPoints(referenceTile);

        //Nested loop for generating of grid filled with blocks
        for (int row = 0; row < rows; row++)
        {
            for (int col = 0; col < columns; col++)
            {
                //Instantiate reference block
                GameObject tile = (GameObject)Instantiate(referenceTile, transform);

                //Calculate position for the block so the final grid is exactly in middle of the screen
                float posX = ((Screen.safeArea.width / 2) + Screen.safeArea.position.x - (columns * cubeSize) / 2 + (col * cubeSize) + (cubeSize / 2));
                float posY = ((Screen.safeArea.height / 2) + Screen.safeArea.position.y + (rows * cubeSize) / 2 - (row * cubeSize) - (cubeSize / 2));

                //Set position of the block
                tile.transform.position = new Vector2(posX, posY);

                //Set name of instantiated object to be equal to its location in the grid
                tile.name = (row + "&" + col);

                //Generate bock with predefined properties based on Level editor settings
                if (SolidBlocks.Contains(tile.name))
                {
                    tile.tag = "Solid";
```

```
                    tile.AddComponent<BoxCollider2D>();
                }
                else if (CrackedBlocks.Contains(tile.name))
                {
                    GenerateBlock(tile,"Cracked", "Squares/cracked", true);
                }
                else if (RampsLeft.Contains(tile.name))
                {
                    GenerateBlock(tile, "RampL", "Squares/rampL", true);
                }
                else if (RampsRight.Contains(tile.name))
                {
                    GenerateBlock(tile, "RampR", "Squares/rampR", true);
                }
                else if (Spikes.Contains(tile.name))
                {
                    GenerateBlock(tile, "Spike", "Squares/spike", true);
                }
                else if (OneWayL.Contains(tile.name))
                {
                    GenerateBlock(tile, "OneWayL", "Squares/onewayL", true);
                }
                else if (OneWayR.Contains(tile.name))
                {
                    GenerateBlock(tile, "OneWayR", "Squares/onewayR", true);
                }
                else if (Diamonds.Contains(tile.name))
                {

                    GenerateBlock(tile, "EmptyTile", "Squares/empty", false);
                    GenerateSpecialTiles(referenceTile, posX, posY, "Diamond");
                }
                else if (Teleports.Contains(tile.name))
                {
                    GenerateBlock(tile, "Teleport", "Squares/Teleport", true);
                    if (GameObject.FindGameObjectsWithTag("Teleport").Length > 1)
                    {
                        //Blue Teleport
                        tile.GetComponent<Image>().color = new Color32(0, 200, 255, 255);
                    }
                    else
```

```
                    {
                        //Green Teleport
                        tile.GetComponent<Image>().color = new Color32(0, 255, 50, 255);

                    }

                }
                else if (StartPosition.Contains(tile.name))
                {
                    //Generate empty space
                    GenerateBlock(tile, "EmptyTile", "Squares/empty", false);
                    //tile.GetComponent<Image>().color = new Color32(255, 255, 255, 0);
                    //Write location of peabody
                    BlockMechanics.PeabodyLocationCol = col;
                    BlockMechanics.PeabodyLocationRow = row;
                    //Generate Peabody
                    GenerateSpecialTiles(referenceTile, posX, posY, "Peabody");
                }
                else
                {
                    GenerateBlock(tile, "EmptyTile", "Squares/empty", false);
                }



            }

            //map coordinates of each row
            string Tilename = row + "&0";
            rowborders[row] = (GameObject.Find(Tilename).transform.position.y) - (float)(cubeSize * 0.5);
            //Debug.Log("bottom line of row " + row + " is:" + rowborders[row]);

        }
        Destroy(referenceTile);
        BlockMechanics.LoadData();
        GameObject.Find("/Canvas/Panel/GameObject/Diamond").transform.SetAsLastSibling();
        GameObject.Find("/Canvas/Panel/GameObject/Peabody").transform.SetAsLastSibling();
        Swipingmechanism.Peabody = GameObject.FindGameObjectsWithTag("Peabody");

```

```
        //If Level contains tutorial panel, show it
        if (TutorialIncluded)
        {
            Tutorial();
        }
        //Load Peabody variable in Block Mechanics Class
        BlockMechanics.Peabody = GameObject.FindGameObjectsWithTag ("Peabody");


    }

    // ensure that report button is always in visible area in upper center of the screen
    void PositionReportButton()
    {
        GameObject.Find("/Canvas/Panel/UserReportingPrefab/UserReportButton").transform.position = new Vector2 (Screen.safeArea.width + Screen.safeArea.position.x - cubeSize/2, Screen.safeArea.height + Screen.safeArea.position.y - cubeSize / 4);

    }

    //Generate GUI that shows how many diamonds are left to collect
    void GenerateDiamondPoints(GameObject referenece)
    {
        for (int i = 2; i < Diamonds.Count + 2; i++)
        {
            GameObject DiamondPoint = (GameObject)Instantiate(referenece, transform);
            DiamondPoint.transform.localScale = new Vector2(cubeSize / 2, cubeSize / 2);
            Sprite Texture = Resources.Load<Sprite>("Squares/diamond");
            DiamondPoint.GetComponent<Image>().sprite = Texture;
            DiamondPoint.tag = "DiamondPoint";
            DiamondPoint.name = ("DiamondPoint");
            //DiamondPoint.transform.SetParent(GameObject.Find("/Canvas/Panel/GameObject/InfoBars").transform);
            DiamondPoint.GetComponent<Image>().color = new Color32(25, 25, 25, 255);
            DiamondPoint.transform.SetParent(GameObject.Find("/Canvas/Panel/GameObject/InfoBars").transform);
            DiamondPoint.transform.position = new Vector2 (Screen.safeArea.position.x + i * (cubeSize / 2), Screen.safeArea.height + Screen.safeArea.position.y - cubeSize / 2);

        }
```

```
    }

    //Generate Diamonds, Peabody
    private void GenerateSpecialTiles(GameObject Reftile, float X, float Y, string name)
    {
        GameObject tile = (GameObject)Instantiate(Reftile, transform);
        tile.transform.position = new Vector2(X, Y);

        //Diamonds need to be in set place in GameObjects hierarchy to ensure that they are always in background
        tile.transform.SetParent(GameObject.Find("/Canvas/Panel/GameObject/" + name).transform);
        tile.name = (name);

        //If Peabody is the special tile
        if (name == "Peabody")
        {

            GenerateCollisionBoxes("Right",tile,0.8f, 0.5f);
            GenerateCollisionBoxes("Left", tile, -0.8f, 0.5f);


        }
        //If Diamond is the special tile
        else if (name == "Diamond")
        {
            GenerateCollisionBoxes("Collider", tile, 0, 0.5f);
        }


        Sprite Texture = Resources.Load<Sprite>("Squares/" + name);
        tile.GetComponent<Image>().sprite = Texture;
        tile.tag = name;
    }

    //Method for creating collision boxes
    void GenerateCollisionBoxes(string name, GameObject Praenttile, float offset, float scale)
    {
        GameObject Collider;
        Collider = new GameObject();
        Collider.transform.position = Praenttile.transform.position;
        Collider.name = name;
        Collider.transform.SetParent(Praenttile.transform);
        Collider.transform.localScale = new Vector2(scale, 0.3f);


        Collider.AddComponent<BoxCollider2D>();
```

```
        Collider.GetComponent<Collider2D>().isTrigger = true;
        Collider.GetComponent<Collider2D>().offset = new Vector2(offset, 0);
        Collider.AddComponent<Rigidbody2D>();
        Collider.GetComponent<Rigidbody2D>().gravityScale = 0.0f;
        Collider.AddComponent<ColliderInfoCollection>();

        //If created collision box is for Peabody
        if (name == "Right" ||  name == "Left")
        {

            Collider.GetComponent<Rigidbody2D>().interpolation = RigidbodyInterpolation2D.Interpolate;
        }

    }

    //Method for creating of usual blocks - basically all blocks except Peabody and Diamonds
    void GenerateBlock(GameObject tile, string tag, string texture, bool Colider)
    {
        Sprite Texture = Resources.Load<Sprite>(texture);
        tile.GetComponent<Image>().sprite = Texture;
        tile.tag = tag;

        //some boxes require colider
        if (Colider)
        {

            // add colider box
            tile.AddComponent<BoxCollider2D>();

            // spikes have smaller colider boxes
            if (tag == "Spike" || (tag == "Teleport"))
            {
                tile.GetComponent<BoxCollider2D>().size = new Vector3(0.5f, 0.5f, 1);
            }
        }
    }

    //If tutorial is ticked as "included" in the Level editor, show the Tutorial Panel at the start of the game
    static void Tutorial()
    {
        GameObject.Find("/Canvas/Panel/Tutorial").SetActive(true);
        BlockMechanics.PauseGame();
    }
```

```
    //Method for closing Tutorial panel when button is pressed
    public static void CloseTutorial()
    {
        GameObject.Find("/Canvas/Panel/Tutorial").SetActive(false);
        BlockMechanics.ResumeGame();
    }
}

```

0.9 Source code - Gridgenerator

C:\Users\Dominik\Peabody\Assets\Scripts\BlockMechanics.cs 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System;

/* BlockMechanics.cs
 * This script sets out mechanics of each game block
 * check if all diamonds were collected and display appropriate menu
*/


public class BlockMechanics : MonoBehaviour
{
    public static bool Swiped = false;

    public static bool TaskFinished = false;
    public Vector3 newpos;


    private static int counter = 0;


    static GameObject PopUpPanel = null;
    static GameObject HeadText = null;
    static GameObject LowerText = null;

    public static GameObject[] Empties = null;
    public static GameObject[] Cracked = null;
    public static GameObject[] Diamonds = null;
    public static GameObject[] DiamondPoints = null;
    public static GameObject[] LifePoints = null;
    public static GameObject[] Peabody = null;
    public static GameObject[] RampsL = null;
    public static GameObject[] RampsR = null;
    public static GameObject[] Spikes = null;
    public static GameObject[] Spawn = null;
    public static GameObject[] Teleports = null;
    public static int PeabodyLocationCol;
    public static int PeabodyLocationRow;
    private static GameObject UnderPeabody;
    public static GameObject explosion;
    public static GameObject explosion2;



    // Start is called before the first frame update
    void Start()
    {
```

```
        PopUpPanel = GameObject.Find("/Canvas/Panel/PopUp");
        HeadText = GameObject.Find("/Canvas/Panel/LevelPanel/HeadText");
        LowerText = GameObject.Find("/Canvas/Panel/LevelPanel/Info");

    }

    // Update is called once per frame
    void FixedUpdate()
    {

        //run check ony after each swipe to lower computing demands
        if (Swiped == true && (Input.touchCount == 0 ||
          Swipingmechanism.row != PeabodyLocationRow ||
          Swipingmechanism.Peabodyfall))
        {


            //Check if there are still diamonds to collect
            CheckIfTaskFinished();

            //If value that usually contains new position for Peabody isnt same as current peabody location or isnt equal zero, proceed
            if ((newpos != Peabody[0].transform.position) && ((newpos.y != 0) || (newpos.x != 0)))
            {
                //If new position is lower than position of Peabody
                if (newpos.y != Peabody[0].transform.position.y)
                {
                    Peabody[0].transform.position = new Vector2(Peabody[0].transform.position.x, Peabody[0].transform.position.y - 1500 * Time.deltaTime);
                    if (newpos.y > Peabody[0].transform.position.y)
                    {
                        Peabody[0].transform.position = new Vector2(Peabody[0].transform.position.x, newpos.y);
                    }

                }
                //If new position is on the right from Peabody current location
                if (newpos.x > Peabody[0].transform.position.x)
                {
                    Peabody[0].transform.position = new Vector2(Peabody[0].transform.position.x + 1500 * Time.deltaTime, Peabody[0].transform.position.y);
                    if (newpos.x < Peabody[0].transform.position.x)
                    {
```

```
                        Peabody[0].transform.position = new Vector2(newpos.x, Peabody[0].transform.position.y);
                    }
                }
                //If new position is on the left from Peabody current location
                else if (newpos.x < Peabody[0].transform.position.x)
                {
                    Peabody[0].transform.position = new Vector2(Peabody[0].transform.position.x - 1500 * Time.deltaTime, Peabody[0].transform.position.y);
                    if (newpos.x > Peabody[0].transform.position.x)
                    {
                        Peabody[0].transform.position = new Vector2(newpos.x, Peabody[0].transform.position.y);
                    }
                }
            }
            //Else if position was changed during current swipe, finish this method and wait for new swipe
            else if (counter > 2)
            {

                counter = 0;
                Swiped = false;


            }
            //Else check if there isnt some interactive block under Peabody
            else
            {
                newpos = CheckIfPeabodyFly();

            }
        }
        //If all diamonds were collected, show end game panel
        if (TaskFinished == true)
        {
            GameObject.Find("/SceneManager/Win").GetComponent<AudioSource>().Play(0);
            GameObject.Find("/Canvas/Panel/").GetComponent<Image>().color = new Color32(0, 150, 0, 255);
            TaskFinished = false;
            GameObject.Find("/Canvas/Panel/LevelPanel").SetActive(true);
            GameObject.Find("/Canvas/Panel/UserReportingPrefab/UserReportButton").SetActive(true);
            Destroy(GameObject.Find("/Canvas/Panel/GameObject"));
            Swipingmechanism.offset = 0;
```

```
            enabled = false;

        }
    }

    //if there are no diamonds to collect, finish the level
    public static void CheckIfTaskFinished ()
    {

        Diamonds = GameObject.FindGameObjectsWithTag("Diamond");

        //If there are no diamonds left, declare that level is finished
        if (Diamonds.Length == 0)
        {
            TaskFinished = true;
        }
        //Else keep declaring level as not finished
        else
        {
            TaskFinished = false;

        }


    }



   // Check if there is an empty block under peabody - this method works only when player isnt swiping
    public Vector3 CheckIfPeabodyFly()
    {
        LoadData();
        //If peabody is on the last row, load block in the row 0
        if (PeabodyLocationRow == 9)
        {
            PeabodyLocationRow = -1;
            UnderPeabody = GameObject.Find("/Canvas/Panel/GameObject/" + 0 + "&" + PeabodyLocationCol);
        }
        //else load row that is right under peabody
        else
        {
            UnderPeabody = GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow + 1) + "&" + PeabodyLocationCol);

        }
        //If there are teleports in the level
        if (Teleports.Length > 1)
```

```
        {//if second teleport is at the same location as Peabody, teleport Peabody
            if (GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow) + "&" + (PeabodyLocationCol)) == Teleports[1])
            {
                teleportPeabody();
                PeabodyLocationRow = Int32.Parse(Teleports[0].name.Split('&')[0]);
                PeabodyLocationCol = Int32.Parse(Teleports[0].name.Split('&')[1]);
                return new Vector3(0, 0, 0);
            }
        }
        //Switch choosing action based on what interactive block is under Peabody
            switch (UnderPeabody.tag)
        {
            //If there is an empty block, move Peabody on its position
            case "EmptyTile":
                GameObject.Find("/SceneManager/Fall").GetComponent<AudioSource>().Play(0);
                PeabodyLocationRow = PeabodyLocationRow + 1;
                return UnderPeabody.transform.position;

                //If there is an Cracked block, destroy the block and move peabody on its position
            case "Cracked":
                //Find cracked block and transfer it to Empty block
                GameObject.Find("/SceneManager/Crack").GetComponent<AudioSource>().Play(0);
                UnderPeabody.GetComponent<Image>().color = new Color32(0, 0, 0, 0);
                UnderPeabody.tag = "EmptyTile";
                //Destroy collider that belonged to the block - empty spaces have no colliders
                Destroy(UnderPeabody.GetComponent<BoxCollider2D>());
                //Let peabody fall into new created empty space
                PeabodyLocationRow = PeabodyLocationRow + 1;
                return UnderPeabody.transform.position;

                //If there is an ramp with slide to the left side, check if there is an empty space on the left from the ramp, if yes, let peabody slide
            case "RampL":

                //Standart case with block space on the left side of the ramp
                if (PeabodyLocationCol>0)
```

```
                {
                    //If there is an empty space, let peabody slide
                    if (GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow + 1) + "&" + (PeabodyLocationCol - 1)).tag == "EmptyTile")
                    {
                        GameObject.Find("/SceneManager/Slide").GetComponent<AudioSource>().Play(0);
                        PeabodyLocationCol = PeabodyLocationCol - 1;
                        PeabodyLocationRow = PeabodyLocationRow + 1;
                        return GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow) + "&" + (PeabodyLocationCol)).transform.position;
                    }
                    //If there isnt, do nothing
                    else
                    {
                        return new Vector3(0, 0, 0);
                    }

                }
                //Case when ramp is on the left side of the screen and peabody will slide through the side of the screen to the right side
                else
                {
                    //If there is an empty space, let peabody slide
                    if (GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow + 1) + "&" + (4)).tag == "EmptyTile")
                    {
                        GameObject.Find("/SceneManager/Slide").GetComponent<AudioSource>().Play(0);

                        PeabodyLocationCol = 4;
                        PeabodyLocationRow = PeabodyLocationRow + 1;
                        Peabody[0].transform.position = GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow) + "&" + (PeabodyLocationCol)).transform.position;
                        return new Vector3(0, 0, 0);
                    }
                    //If there isnt, do nothing
                    else
                    {
                        return new Vector3(0, 0, 0);
                    }
                }


```

```
            //If there is an ramp with slide to the right side, check if there is an empty space on the right from the ramp, if yes, let peabody slide
            case "RampR":
                //Standart case with block space on the right side of the slide
                if (PeabodyLocationCol + 1 < 5)
                {
                    //If there is an empty space, let peabody slide
                    if (GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow + 1) + "&" + (PeabodyLocationCol + 1)).tag == "EmptyTile")
                    {
                        GameObject.Find("/SceneManager/Slide").GetComponent<AudioSource>().Play(0);
                        PeabodyLocationRow = PeabodyLocationRow + 1;
                        PeabodyLocationCol = PeabodyLocationCol + 1;
                        return GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow) + "&" + (PeabodyLocationCol)).transform.position;
                    }
                    //If there isnt, do nothing
                    else
                    {
                        return new Vector3(0, 0, 0);
                    }
                }
                //Case when ramp is on the right side of the screen and peabody will slide through the side of the screen to the left side
                else
                {
                    //If there is an empty space, let peabody slide
                    if (GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow + 1) + "&" + (0)).tag == "EmptyTile")
                    {
                        GameObject.Find("/SceneManager/Slide").GetComponent<AudioSource>().Play(0);
                        PeabodyLocationRow = PeabodyLocationRow + 1;
                        PeabodyLocationCol = 0;
                        Peabody[0].transform.position = GameObject.Find("/Canvas/Panel/GameObject/" + (PeabodyLocationRow) + "&" + (PeabodyLocationCol)).transform.position;
                        return new Vector3(0, 0, 0);
                    }
                    //If there isnt, do nothing
                    else
```

```
                    {
                        return new Vector3(0, 0, 0);
                    }
                }

                //If there is an spike block under peabody, take life from peabody
            case "Spike":

                TakeLife();
                return new Vector3(0, 0, 0);

                //If there is an teleport under peabody, let peabody fall - teleportation is handled when Peabody is on the same position as teleport
            case "Teleport":

                GameObject.Find("/SceneManager/Fall").GetComponent<AudioSource>().Play(0);
                PeabodyLocationRow = PeabodyLocationRow + 1;
                return UnderPeabody.transform.position;

                //if there isnt interactive block under peabody, do nothing
            default:

                    counter++;
                    return new Vector3(0, 0, 0);





        }

    }

    //Teleport method
    public static void teleportPeabody()
    {
        //Make sound of teleportation
        GameObject.Find("/SceneManager/Teleport").GetComponent<AudioSource>().Play(0);
        //Teleport peabody to new location
        Peabody[0].transform.position = Teleports[0].transform.position;

        //create animation on the location of the first teleport
        explosion = (GameObject)Instantiate(Resources.Load("Teleport"));
```

```
        explosion.transform.localPosition = new Vector2(Teleports[0].transform.position.x, Teleports[0].transform.position.y+50);
        explosion.transform.SetParent(GameObject.Find("/Canvas/Panel/GameObject/Peabody").transform);

        //create animation on the location of the second teleport
        explosion2 = (GameObject)Instantiate(Resources.Load("Teleport"));
        explosion2.transform.localPosition = new Vector2(Teleports[1].transform.position.x, Teleports[1].transform.position.y + 50);
        explosion2.transform.SetParent(GameObject.Find("/Canvas/Panel/GameObject/Peabody").transform);

    }

    //Show in GUI how many diamonds were collected
    public static void CollectDiamonds()
    {
        // proceed with method only if level wasnt finished
        if (Diamonds.Length < DiamondPoints.Length)
        {
            GameObject.Find("/SceneManager/Diamond").GetComponent<AudioSource>().Play(0);
            DiamondPoints[Diamonds.Length].GetComponent<Image>().color = new Color32(255, 255, 255, 255);

        }

    }

    //When peabody touch the spike, 1 life is lost
    public static void TakeLife()
    {
            GameObject.Find("/SceneManager/Loss").GetComponent<AudioSource>().Play(0);
            HeadText.GetComponent<Text>().text = "Game Over!";
            LowerText.GetComponent<Text>().text = "Sadly peabody died, try again!";
            GameObject.Find("/Canvas/Panel/LevelPanel").SetActive(true);
            GameObject.Find("/Canvas/Panel/").GetComponent<Image>().color = new Color32(255, 0, 0, 255);
            GameObject.Find("/Canvas/Panel/LevelPanel/Button (1)").SetActive(false);
            GameObject.Find("/Canvas/Panel/GameObject").SetActive(false);

    }

    // Pause game
    public static void PauseGame()
```

```
    {
        Time.timeScale = 0;
    }



    // Resume game
    public static void ResumeGame()
    {
        PopUpPanel.SetActive(false);
        Time.timeScale = 1;
    }


    //find objects with game tags for further functions
    public static void LoadData()
    {
        Empties = GameObject.FindGameObjectsWithTag("EmptyTile");
        Cracked = GameObject.FindGameObjectsWithTag("Cracked");
        DiamondPoints = GameObject.FindGameObjectsWithTag("DiamondPoint");
        RampsL = GameObject.FindGameObjectsWithTag("RampL");
        RampsR = GameObject.FindGameObjectsWithTag("RampR");
        Spikes = GameObject.FindGameObjectsWithTag("Spike");
        Spawn = GameObject.FindGameObjectsWithTag("Spawn");
        Teleports = GameObject.FindGameObjectsWithTag("Teleport");
        Diamonds = GameObject.FindGameObjectsWithTag("Diamond");

    }

}
```

0.10 Source code - SwipingMechanism

C:\Users\Dominik\Peabody\Assets\Scripts\Swipingmechanism.cs 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System;

/* Swipingmechanism.cs
 * This file contain methods for touch and swiping of the rows
 *
 *
 *
*/


public class Swipingmechanism : MonoBehaviour
{
    private Vector2 startSwipePosition;
    private Vector2 currentFingerPosition;

    public static float offset;
    public static GameObject[] Squares;
    public static Vector2[] squaresnatur;
    public static int row;
    public static GameObject[] Peabody;
    private static GameObject NewPositionPeabody;
    private float PeabodyOffset;
    private float PeabodyStartOffset;
    private Vector3 startPeabodyPosition;
    private bool ContainsOneWayL = false;
    private GameObject OneWayL;
    private bool ContainsOneWayR = false;
    private GameObject OneWayR;
    private bool reportedblock = false;
    public static bool Peabodyfall = false;




    // Update is called once per frame
    void Update()
    {
        //If finger doesnt touch a screen method does not run
        if (Input.touchCount > 0 && !Peabodyfall)
        {
            SwipeDetection();
        }
        else if (offset != 0)
        {
```

```
            //If ColliderInfoCollection reported that Diamond is touching something and swipe should be blocked
            if (ColliderInfoCollection.blocker)
            {
                char position = ColliderInfoCollection.TouchedDiamond[ColliderInfoCollection.TouchedDiamond.Length - 1];
                //logic: '9' = ASCII 57, therefore 57 - 48 = 9
                int c = position - 48;
                // Diamond was touched during swipe to the left
                if (ColliderInfoCollection.offset > 0)
                {
                    offset = squaresnatur[c].x - ColliderInfoCollection.DiamondLocation- Gridgenerator.cubeSize;
                    MoveBlocks(offset);
                }
                // Diamond was touched during swipe to the right
                else
                {
                    offset = squaresnatur[c].x - ColliderInfoCollection.DiamondLocation + Gridgenerator.cubeSize;
                    MoveBlocks(offset);
                }
            }
            LockInNewPoistion();
        }
        //If there is no active touch input, reset value of "Peabodyfall"
        else if (Input.touchCount == 0)
        {
            Peabodyfall = false;
        }
    }


    public void SwipeDetection()
    {
        if ((Input.GetTouch(0).position.y > Gridgenerator.rowborders[Gridgenerator.rowborders.Length - 1])&&(Input.GetTouch(0).position.y <(Gridgenerator.rowborders[0]+ Gridgenerator.cubeSize)))
        {
            if (Input.GetTouch(0).phase == TouchPhase.Began)
            {
                startSwipePosition = Input.GetTouch(0).position;
                startPeabodyPosition = Peabody[0].transform.position;
                Peabodyfall = false;
                ContainsOneWayL = false;
                ContainsOneWayR = false;
```

```
                // Detect which row player swipe on
                SelectRow(startSwipePosition.y);


                //Save Original position of squares
                squaresnatur = new Vector2[Squares.Length];
                for (int i = 0; i < Squares.Length; i++)
                {
                    squaresnatur[i] = Squares[i].transform.position;
                }


            }
            else if (Input.GetTouch(0).phase == TouchPhase.Moved || Input.GetTouch(0).phase == TouchPhase.Stationary)
            {
                offset = startSwipePosition.x - Input.GetTouch(0).position.x;
                // Blocker is activate for example when row touche the diamond while swipe
                if (!ColliderInfoCollection.blocker && !blocker())
                {

                    //calculate offset to know distance which finger traveled
                    if (!ContainsOneWayL && !ContainsOneWayR)
                    {
                        MoveBlocks(offset);
                    }
                    //if one way Left cube get out of the screen on the rght side, its blocked
                    else if (ContainsOneWayL && (!(OneWayL.transform.position.x > squaresnatur[Squares.Length - 1].x) || (offset > 0)) && !ContainsOneWayR)
                    {
                        reportedblock = false;
                        MoveBlocks(offset);
                    }
                    //if one way Right cube get out of the screen on the left side, its blocked
                    else if (ContainsOneWayR && (!(OneWayR.transform.position.x < squaresnatur[0].x) || (offset < 0)) && !ContainsOneWayL)
                    {
                        reportedblock = false;
                        MoveBlocks(offset);
                    }
```

```
                    //if row contain both left and right one way cubes, the swipe is blocked if one of them get out of the screen in opposite way than their allowed direction
                    else if ((ContainsOneWayL && (OneWayL.transform.position.x <= squaresnatur[Squares.Length - 1].x)) && (ContainsOneWayR && (OneWayR.transform.position.x >= squaresnatur[0].x)))
                    {
                        MoveBlocks(offset);
                    }
                    else
                    {
                        if (!reportedblock)
                        {
                            GameObject.Find("/SceneManager/Block").GetComponent<AudioSource>().Play(0);
                            reportedblock = true;
                        }

                    }
                }
                else if (blocker())
                {
                    GameObject.Find("/SceneManager/Block").GetComponent<AudioSource>().Play(0);
                    LockInNewPoistion();
                    Peabodyfall = true;
                }
                else if (offset > 0 && offset < ColliderInfoCollection.offset)
                {
                    MoveBlocks(offset);
                }
                else if (offset < 0 && offset > ColliderInfoCollection.offset)
                {
                    MoveBlocks(offset);
                }
            }
        }
    }



    public void MoveBlocks(float offset)
    {
        // move blocks acordingly to position of the finger
        for (int i = 0; i < Squares.Length; i++)
```

```
        {
            //add offset into position
            Squares[i].transform.position = new Vector2(squaresnatur[i].x - offset, squaresnatur[i].y);



            //if cube on the left get out of the screen, it appears on the right
            if (Squares[i].transform.position.x < squaresnatur[0].x - Gridgenerator.cubeSize / 4)
            {
                Squares[i].transform.position += new Vector3 (Gridgenerator.cubeSize * Squares.Length, 0);

            }

            //if cube on the right get out of the screen, it appears on the left
            else if (Squares[i].transform.position.x > squaresnatur [Squares.Length - 1].x + Gridgenerator.cubeSize / 4)
            {
                Squares[i].transform.position -= new Vector3 (Gridgenerator.cubeSize * Squares.Length, 0);
            }

            //if cube touch peabody, peabody move
            MovePeabody();
        }

    }

    public void MovePeabody()
    {

        // When Cube touched peabody from the left  side
        if ( (ColliderInfoCollection.HisNameL != null) && (offset < 0))
        {
            if ((Int32.Parse(ColliderInfoCollection.HisNameL.Split('&') [0]) == row))
            {
                PeabodyOffset = offset - PeabodyStartOffset;
                if (NewPositionPeabody != null)
                {

                    //If player retracted his swipe above level of original position of peabody, peabody wouldnt move anymore
                    if (PeabodyOffset > 1)
```

```
                    {

                        Peabody[0].transform.position = startPeabodyPosition;
                        NewPositionPeabody = null;
                        PeabodyStartOffset = 0;

                    }
                    else
                    {
                        Peabody[0].transform.position = new Vector2(NewPositionPeabody.transform.position.x, Peabody[0].transform.position.y);
                    }


                }
                else if (PeabodyOffset < 0)
                {
                        NewPositionPeabody = PeabodyMagnet(1, ColliderInfoCollection.HisNameL);
                }
            }
        }
        // When Cube touched peabody from the right side
        else if ( (ColliderInfoCollection.HisNameR != null) && (offset > 0))
        {
            if (Int32.Parse(ColliderInfoCollection.HisNameR.Split('&')[0]) == row)
            {
                PeabodyOffset = offset - PeabodyStartOffset;

                if (NewPositionPeabody != null)
                {
                    //If player retracted his swipe above level of original position of peabody, peabody wouldnt move anymore
                    if (PeabodyOffset < 0)
                    {
                        Peabody[0].transform.position = startPeabodyPosition;
                        NewPositionPeabody = null;
                        PeabodyStartOffset = 0;

                    }
                    else
                    {
                        Peabody[0].transform.position = new Vector2
```

```
                        (NewPositionPeabody.transform.position.x, Peabody [0].transform.position.y);
                         }


                    }
                    else if (PeabodyOffset > 0)
                    {

                        NewPositionPeabody = PeabodyMagnet(-1, ColliderInfoCollection.HisNameR);
                    }
                }

            }


        }



        public void LockInNewPoistion()
        {
            //If player moved a row left
            if (offset > 0)
            {
                for (int i = 0; i < Squares.Length; i++)
                {
                    //Check what original block position is nearest to the rightest block in swpied row
                    if (((Squares[Squares.Length - 1].transform.position.x) > (squaresnatur[i].x - Gridgenerator.cubeSize / 2) && (Squares[Squares.Length - 1].transform.position.x) < (squaresnatur[i].x + Gridgenerator.cubeSize / 2)) || ((Squares[Squares.Length - 1].transform.position.x) < squaresnatur[i].x && (i==0)))
                    {
                        //save into variable by how many blocks was the row moved
                         int move = i-(Squares.Length - 1);

                         for (int n = Squares.Length - 1; n > -1; n--)
                         {
                             if (n+move >-1)
                             {
                                 //move the first square from absolute left to absolute right
                                 Squares[n].transform.position = squaresnatur[n
```

```
                        + move];
                                Squares[n].name = (row + "&" + (n + move));

                                //Copy coordinates if block is on same location as peabody
                                if (Squares[n] == NewPositionPeabody)
                                {
                                    BlockMechanics.PeabodyLocationCol = n + move;
                                    BlockMechanics.PeabodyLocationRow = row;
                                }
                            }
                            else
                            {
                                //Move square "i" to the postion of square one block left
                                Squares[n].transform.position = squaresnatur[Squares.Length + (n+move)];
                                Squares[n].name = (row + "&" + (Squares.Length + (n + move)));

                                //Copy coordinates if block is on same location as peabody
                                if (Squares[n] == NewPositionPeabody)
                                {
                                    BlockMechanics.PeabodyLocationCol = Squares.Length + (n + move);
                                    BlockMechanics.PeabodyLocationRow = row;

                                }
                            }


                        }
                        offset = 0;
                        if (NewPositionPeabody != null)
                        {
                        Peabody[0].transform.position = new Vector2(NewPositionPeabody.transform.position.x, Peabody[0].transform.position.y);
                        PeabodyStartOffset = 0;
                        NewPositionPeabody = null;
                        }
                        //send info to SwpiePrototype Class that swipe was finished
                        BlockMechanics.Swiped = true;
                    }
                }
            }
```

```
        //If player moved a row right
        else if (offset < 0)
        {
            for (int i = Squares.Length - 1; i > -1; i--)
            {
                //Check what original block position is nearest to the leftest block in swpied row
                if (((Squares[0].transform.position.x) > (squaresnatur[i].x - Gridgenerator.cubeSize / 2) && (Squares[0].transform.position.x) < (squaresnatur[i].x + Gridgenerator.cubeSize / 2))|| (Squares[0].transform.position.x > (squaresnatur[i].x) && i==(Squares.Length - 1)))
                {
                    //save into variable by how many blocks was the row moved
                    int move = i;

                    for (int n = 0; n < Squares.Length; n++)
                    {
                        if (n + move < Squares.Length)
                        {
                            //move the first square from absolute left to absolute right
                            Squares[n].transform.position = squaresnatur[n + move];
                            Squares[n].name = (row + "&" + (n + move));

                            //Copy coordinates if block is on same location as peabody
                            if (Squares[n] == NewPositionPeabody)
                            {
                                BlockMechanics.PeabodyLocationCol = n + move;
                                BlockMechanics.PeabodyLocationRow = row;
                            }
                        }
                        else
                        {
                            //Move square "i" to the postion of square one block left
                            Squares[n].transform.position = squaresnatur[(n + move)-Squares.Length];
                            Squares[n].name = (row + "&" + ((n + move)-Squares.Length));

                            //Copy coordinates if block is on same location as peabody
                            if (Squares[n] == NewPositionPeabody)
```

```
                                {
                                    BlockMechanics.PeabodyLocationCol = (n + move) - Squares.Length;
                                    BlockMechanics.PeabodyLocationRow = row;

                                }
                            }
                        }
                        offset = 0;
                        if (NewPositionPeabody != null)
                        {
                            Peabody[0].transform.position = new Vector2(NewPositionPeabody.transform.position.x, Peabody[0].transform.position.y);
                            NewPositionPeabody = null;
                            PeabodyStartOffset = 0;
                        }
                        //send info to SwpiePrototype Class that swipe was finished
                        BlockMechanics.Swiped = true;
                    }
                }
            }
        }


        //Select right row depending on if touch was conducted above bottom corner of block on specific row
        public void SelectRow(float swipelevel)
        {
            int i = (int)Gridgenerator.rows;


            while (swipelevel > Gridgenerator.rowborders[i])
            {
                i--;

            }
            i++;
                //Load variable with specific row
                Squares = new GameObject[(int)Gridgenerator.columns];
                row = i;
                bool found = false;
                for (int y = 0; y < Gridgenerator.columns; y++)
                {
                    Squares[y] = GameObject.Find("/Canvas/Panel/GameObject/" + i + "&" + y);

                    if (!found)
```

```
                {
                    if (Squares[y].CompareTag("OneWayR"))
                    {
                        if (ContainsOneWayL == true)
                        {
                            ContainsOneWayL = true;
                            ContainsOneWayR = true;
                            OneWayR = Squares[y];
                            found = true;
                        }
                        else
                        {
                            ContainsOneWayL = false;
                            ContainsOneWayR = true;
                            OneWayR = Squares[y];
                            //found = true;
                        }
                    }
                    else if (Squares[y].CompareTag("OneWayL"))
                    {
                        if (ContainsOneWayR == true)
                        {
                            ContainsOneWayL = true;
                            ContainsOneWayR = true;
                            OneWayL = Squares[y];
                            found = true;
                        }
                        else
                        {
                            ContainsOneWayL = true;
                            ContainsOneWayR = false;
                            OneWayL = Squares[y];
                            //found = true;
                        }
                    }
                }
            }

    }

    // Find an empty space to which should peabody attach in case when block is already pushing Peabody
    private GameObject PeabodyMagnet (int side, string Hisname)
    {
        PeabodyStartOffset = offset;
        string MagnetTo;
        int newpos = int.Parse(Hisname.Substring(Hisname.Length - 1)) + side;

```

```
        if (newpos > Squares.Length-1)
        {
            newpos = 0;

        }
        else if (newpos < 0)
        {
            newpos = Squares.Length-1;
        }

        MagnetTo = Hisname.Remove(Hisname.Length - 1, 1) + newpos;

        Peabody[0].transform.position = new Vector2(GameObject.Find(MagnetTo).transform.position.x, Peabody[0].transform.position.y);
        return GameObject.Find(MagnetTo);

    }

    // detect if there is an empty space under peabody while swiping
    private bool pitdetector (bool right, int rower)
    {

        //if user swiped to the right
        if (right)
        {
            for (int i = 0; i < 5;i++)
            {
                //find gameobject that is under Peabody
                GameObject gameobject = GameObject.Find("/Canvas/Panel/GameObject/" + (row + rower) + "&" + i);

                //If an empty place or cracked block etc... that was previously on the left from Peabody move right into position that he is now under peabody, make peabody fall
                if (gameobject.CompareTag("EmptyTile") || gameobject.CompareTag("Cracked") || gameobject.CompareTag("RampR") || gameobject.CompareTag("RampL") || gameobject.CompareTag("Teleport"))
                {
                    //In case when peabody is swiped through the right side of the screen and empty space is on the left side of the row
                    if (Peabody[0].transform.position.x > gameobject.transform.position.x && squaresnatur[Squares.Length - 1].x < startPeabodyPosition.x-PeabodyOffset-Gridgenerator.cubeSize/2 && rower ==1)
                    {
                        return true;
```

```
                    }
                    //In case when row under peabody is swiped to the right
                    if (Peabody[0].transform.position.x > gameobject.transform.position.x && squaresnatur[i].x + Gridgenerator.cubeSize * (5) - offset < Peabody[0].transform.position.x && rower == 0)
                    {
                        return true;
                    }
                    //In case when peabody is swiped to the left and empty block was already on the right from peabody
                    else if (Peabody[0].transform.position.x > gameobject.transform.position.x && i > BlockMechanics.PeabodyLocationCol)
                    {
                        return true;
                    }



                }

            }
            return false;
        }
        //if user swiped to the left
        else
        {
            for (int i = 4; i > -1; i--)
            {
                //find gameobject that is under Peabody
                GameObject gameobject = GameObject.Find("/Canvas/Panel/GameObject/" + (row + rower) + "&" + i);
                //If an empty place or cracked block etc... that was previously on the right from Peabody move left into position that he is now under peabody, make peabody fall
                if (gameobject.CompareTag("EmptyTile") || gameobject.CompareTag("Cracked") || gameobject.CompareTag("RampR") || gameobject.CompareTag("RampL"))
                {
                    //In case when peabody is swiped through the left side of the screen and empty space is on the right side of the row
                    if (Peabody[0].transform.position.x < gameobject.transform.position.x && squaresnatur[0].x - Gridgenerator.cubeSize > startPeabodyPosition.x -
```

```
                        PeabodyOffset / 2 && rower ==1)
                        {
                            return true;
                        }
                        //In case when row under peabody is swiped to the left
                        if (Peabody[0].transform.position.x <
                        gameobject.transform.position.x && squaresnatur[i].x -
                        Gridgenerator.cubeSize * (5) - offset > Peabody
                        [0].transform.position.x && rower == 0)
                        {
                            return true;
                        }
                        //In case when peabody is swiped to the left and empty
                        block was already on the left from peabody
                        else if (Peabody[0].transform.position.x <
                        gameobject.transform.position.x && i <
                        BlockMechanics.PeabodyLocationCol)
                        {
                            return true;
                        }
                    }

                }
                return false;
            }
        }

        private bool blocker ()
        {
            if (offset< 0)
            {
                if (row == BlockMechanics.PeabodyLocationRow)
                {
                    return pitdetector(true, 1);
                }
                else if (row == BlockMechanics.PeabodyLocationRow + 1)
                {
                    return pitdetector(false, 0);
                }
                else
                {
                    return false;
                }

            }
            else if (offset > 0)
            {

                if (row == BlockMechanics.PeabodyLocationRow)
```

```
            {
                return pitdetector(false, 1);
            }
            else if (row == BlockMechanics.PeabodyLocationRow + 1)
            {
                return pitdetector(true, 0);
            }
            else
            {
                return false;
            }
        }
        else
        {
            return false;
        }
    }
}
```

0.11 Source code - ColliderInfoCollection

```
...inik\Peabody\Assets\Scripts\ColliderInfoCollection.cs                    1
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/* ColliderInfoCollection.cs
 * This file contain pre defined methods "OnTriggerEnter2D",
   "OnTriggerStay2D" and "OnTriggerExit2D" triggered by collision boxes on
    Peabody and Diamonds
 * Data about collisions are saved depending on specified criterias and
   then used by other classes
 *
 *
 *
*/


public class ColliderInfoCollection : MonoBehaviour
{
    public static string MyNameL = null;
    public static string HisNameL = null;
    public static string MyNameR = null;
    public static string HisNameR = null;
    public static float offset = 0f;
    public static bool blocker = false;
    public static float location = 0f;
    public static string TouchedDiamond = null;
    public static float DiamondLocation = 0f;




    public void Update()
    {
        List<Collider2D> results = new List<Collider2D>();
        if (Physics2D.OverlapCollider(gameObject.GetComponent<Collider2D>
          (), new ContactFilter2D().NoFilter(), results) > 0)
        {
            PeabodyColision(results[0], gameObject);
        }
        else if (Physics2D.OverlapCollider
          (gameObject.GetComponent<Collider2D>(), new ContactFilter2D
          ().NoFilter(), results) == 0)
        {
            if (Input.touchCount > 0)
            {
                // if swipe to the left continues and Peabody appear on
                  right on the screen, he will keep the information about
                  block that was pushing him, if not, erase the
                  information
```

```
                if (gameObject.name == "Right" && Swipingmechanism.offset < 0)
                {
                    PeabodyColisionDelete(gameObject);
                }
                // if swipe to the right continues and Peabody appear on left on the screen, he will keep the information about block that was pushing him, if not, erase the information
                else if (gameObject.name == "Left" && Swipingmechanism.offset > 0)
                {
                    PeabodyColisionDelete(gameObject);
                }
            }
            // erase informations about touching objects if finger doesnt touch the screen
            else
            {
            PeabodyColisionDelete(gameObject);

            }



        }
    }

    private void OnTriggerStay2D(Collider2D other)
    {
        PeabodyColision( other, gameObject);

        // If Diamond was touched by peabody, diamond is collected
        if ((gameObject.name == "Collider" && offset != 0) && ((other.gameObject.name == "Right") || (other.gameObject.name == "Left")))
        {
            DiamondCollection();

        }


    }
    private void OnTriggerEnter2D(Collider2D other)
    {
        PeabodyColision(other, gameObject);


        //Situations for Diamond Colliders
```

```
        if (gameObject.name == "Collider")
        {
            // if the other object is Peabody, diamond is collected
            if ((other.gameObject.name == "Right") || (other.gameObject.name == "Left"))
            {
                    DiamondCollection();
            }
            // Otherwise, block swiping
            else
            {
                GameObject.Find("/SceneManager/Block").GetComponent<AudioSource>().Play(0);
                offset = Swipingmechanism.offset;
                blocker = true;
                location = Input.GetTouch(0).position.x;
                TouchedDiamond = other.gameObject.name;
                DiamondLocation = gameObject.transform.position.x;



            }

        }



    }

    void OnTriggerExit2D(Collider2D other)
    {

        //When there is no object touching the Diamond anymore, swiping is allowed again
        if (gameObject.name == "Collider")
        {
            blocker = false;
        }

    }

    void DiamondCollection()
    {

        //Collect diamond
        transform.parent.gameObject.SetActive(false);
        //Check how many diamonds are still in the game via LoadData method
```

```
        BlockMechanics.LoadData();
        //Show new informations in the GUI
        BlockMechanics.CollectDiamonds();
        // If this was a last diamond, finish the game
        BlockMechanics.CheckIfTaskFinished();

    }

    void PeabodyColisionDelete( GameObject gameObject)
    {
        //Erase informations about block that touched Peabody from the right side but now left
        if ((gameObject.name == "Right"))
        {
            MyNameR = null;
            HisNameR = null;

        }
        //Erase informations about block that touched Peabody from the left side but now left
        if ((gameObject.name == "Left"))
        {
            MyNameL = null;
            HisNameL = null;

        }

    }


    void PeabodyColision(Collider2D other, GameObject gameObject)
    {

        if (gameObject.name == "Right" && other.gameObject.name != "Collider" && other.gameObject.tag != "Teleport")
        {
            //if Peabody touch spike because of player swipe, he loose a life
            if (other.gameObject.tag == "Spike")
            {
                BlockMechanics.TakeLife();
            }
            // otherwise, save informations about boxes that touched peabody
            else
            {
                MyNameR = gameObject.name;
                HisNameR = other.gameObject.name;

```

```
            }
        }
        if (gameObject.name == "Left" && other.gameObject.name != "Collider" && other.gameObject.tag != "Teleport")
        {
            //if Peabody touch spike because of player swipe, he loose a life
            if (other.gameObject.tag == "Spike")
            {
                BlockMechanics.TakeLife();
            }
            // otherwise, save informations about boxes that touched peabody
            else
            {
                MyNameL = gameObject.name;
                HisNameL = other.gameObject.name;
            }
        }
    }
}
```

0.12 Source code - SaveData

C:\Users\Dominik\Peabody\Assets\Scripts\SaveData.cs 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class SaveData : MonoBehaviour
{


    public void SaveExperience(int NextLevel)
    {

        PlayerPrefs.SetInt("LastLevel", NextLevel);

        SceneManager.LoadScene(NextLevel);
    }
}

```

0.13 Source code - LoadGame

C:\Users\Dominik\Peabody\Assets\Scripts\LoadGame.cs 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LoadGame : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

        // if player played the game before, he can continue the game and continue button is unlocked
        if (PlayerPrefs.GetInt("LastLevel") > 0)
        {
            gameObject.GetComponent<Button>().interactable = true;


        }
        else {

        gameObject.GetComponent<Button>().interactable = false;

        }

    }

    // Update is called once per frame


    public static void Load()
    {
        SceneManager.LoadScene(PlayerPrefs.GetInt("LastLevel"));
    }
}

```

0.14 Source code - SceneControll

C:\Users\Dominik\Peabody\Assets\Scripts\SceneControll.cs 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

/* SceneControll.cs
 * This file contain basic methods for buttons to change scenes
*/


public class SceneControll: MonoBehaviour
{
 public void OpenForm()
    {
        SceneManager.LoadScene(13);


    }
 public void OpenGridgenerator()
    {
        SceneManager.LoadScene(1);


    }

    public void openEmailSendScreen()
    {
        SceneManager.LoadScene(8);


    }



    private void Start()
    {
        Application.targetFrameRate = 300;
    }


}

```