Technical Design Document

SCRMG – Juho Pirttioja

2017.05.22

Contents

[Tools 1](#_Toc483225730)

[Timeframe 1](#_Toc483225731)

[Goals 1](#_Toc483225732)

[Script Relations 1](#_Toc483225733)

[Class Descriptions 2](#_Toc483225734)

[Toolbox 2](#_Toc483225735)

[Event Manager 2](#_Toc483225736)

[Subscribing and unsubscribing 2](#_Toc483225737)

[Broadcasting 2](#_Toc483225738)

[Creating new events 3](#_Toc483225739)

[Delegates 3](#_Toc483225740)

[Game Manager 3](#_Toc483225741)

[Initializing a match 4](#_Toc483225742)

[Keeping track of match timer 5](#_Toc483225743)

[Declaring match winner 6](#_Toc483225744)

[Application Manager 6](#_Toc483225745)

[Input Manager 6](#_Toc483225746)

[UI Manager 8](#_Toc483225747)

[Main Menu 8](#_Toc483225748)

[In-game HUD 9](#_Toc483225749)

[Pause Menu 11](#_Toc483225750)

[Game End Menu 12](#_Toc483225751)

[Global Variable Library 12](#_Toc483225752)

[Ship Controller 12](#_Toc483225753)

[Local Player Controller 12](#_Toc483225754)

[AI Player Controller 12](#_Toc483225755)

[Other Controllers 13](#_Toc483225756)

[Scene Infos 13](#_Toc483225757)

[Tag-scripts 13](#_Toc483225758)

[Pausing the game 13](#_Toc483225759)

[Naming conventions 13](#_Toc483225760)

[Postfixes 13](#_Toc483225761)

[Android support 14](#_Toc483225762)

[Other guidelines 14](#_Toc483225763)

[Tags 14](#_Toc483225764)

[Graphical guidelines 14](#_Toc483225765)

[Asset list 14](#_Toc483225766)

[Ship 14](#_Toc483225767)

[Projectile 14](#_Toc483225768)

[Wall 15](#_Toc483225769)

[Floor 15](#_Toc483225770)

[UI 15](#_Toc483225771)

[Version control 15](#_Toc483225772)

[Link to repository 15](#_Toc483225773)

# Tools

* Unity 3d
* Android SDK
* Visual Studio 2015
* Adobe Photoshop
* SourceTree

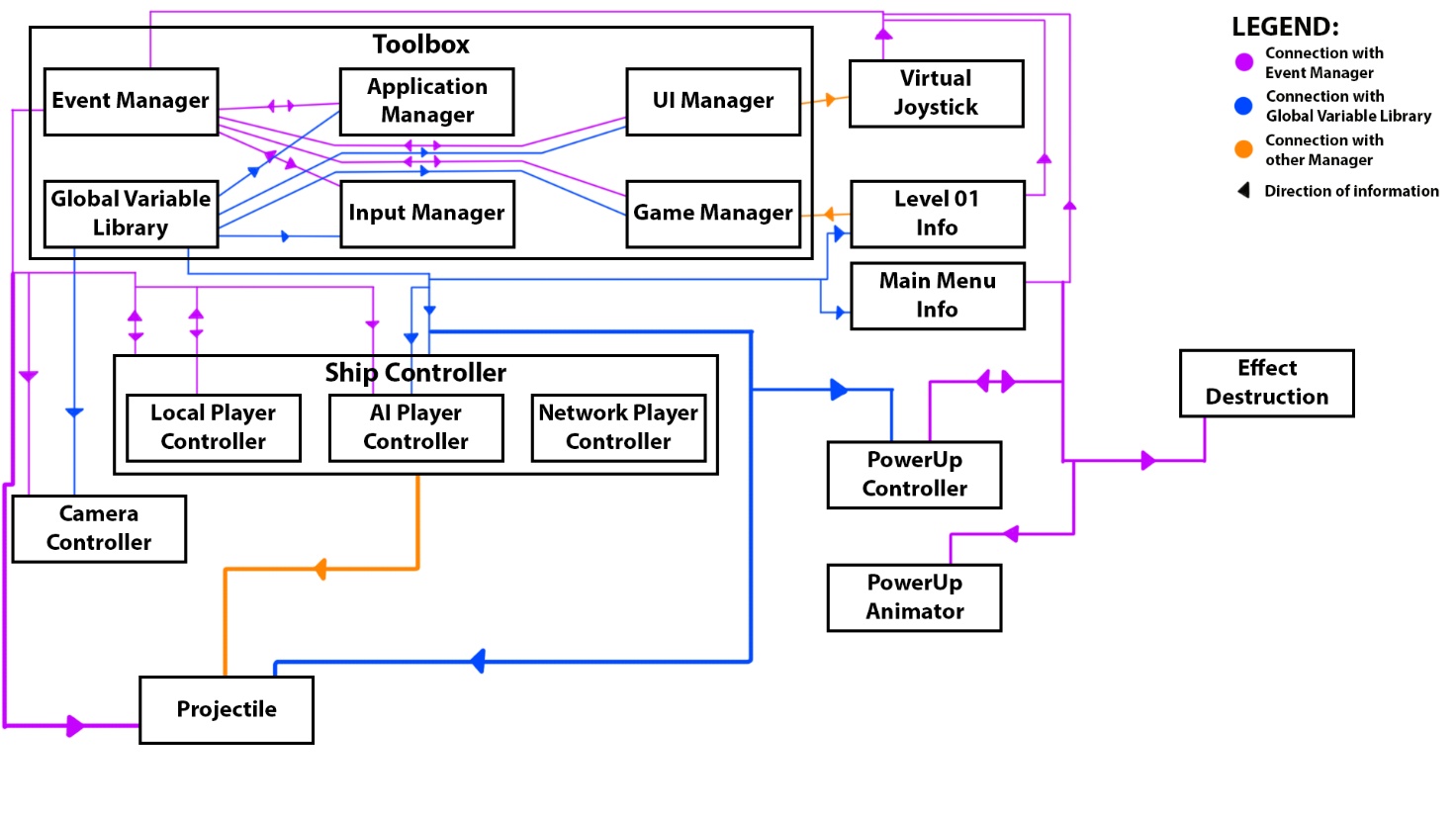
# Timeframe

2017 spring (25. February – 22. May)

# Goals

1. Gameplay (spawning, moving, aiming, shooting, destroying ships)
2. Android support (intuitive controls, optimized performance aka. no noticeable lag)
3. Main Menu (main menu, with settings, play and exit functionality)
4. Pause Menu (can be opened during gameplay, pauses game, enables restarting a match and returning to main menu mid-game)

# Script Relations



# Class Descriptions

## Toolbox

The only the code the Toolbox-script contains is to make it a singleton. Its function is primarily to enable easy finding of the Toolbox gameObject, which contains all the Manager-scripts, and to ensure that only one Toolbox exists at a time.

## Event Manager

Main method of communication between scripts. Reference can be found through toolbox. Is a singleton and one always exists. Works on subscription / broadcast basis. Scripts can subscribe their methods through delegates. When an event is broadcasted, all the methods subscribed to the event are called.

### Subscribing and unsubscribing

Any script can subscribe a method to an event in event manager. When an event is broadcasted, all methods that are subscribed to the event in question are called. Subscribe a method to an event with “+=” modifier and unsubscribe with “-=” modifier. The subscribing method must have identical parameter types as the respective event. The subscribing method should be named identical to the event for clarity’s sake.

**Example 01:** Subscribing to MouseButtonLeftDown-event

*em.OnMouseButtonLeftDown += OnMouseButtonLeftDown;*

Where “*em*” is a reference to the EventManager, the former “*OnMouseButtonDown*” is the name of the event in EventManager, and the latter “*OnMouseButtonDown*” is the name of the method being subscribed.

In the case of unsubscribing the “+” would be changed to “-“.

Subscribing is often done in the OnEnable method, to ensure the script is aware of events from the moment it is enabled. However, this is not mandatory. For example, when of pausing the game LocalPlayerController could unsubscribe from all input events and then re-subscribe to them when the game is unpaused, to prevent player inputs to the ship during game pause.

**NB:** Whenever a script is subscribing to an event, ensure that it is also unsubscribed to the respective event in its OnDisable method. This is done to prevent EventManager from trying to call methods in scripts that no longer exist.

### Broadcasting

Any script can broadcast an event. Broadcasting events is the primary way of communicating between scripts in cases when we have no reference to the script we want to communicate with. To broadcast an event, the script first needs a reference to the EventManager (generally abbreviated as “em”). EventManager can always be found as a component of Toolbox. Once we have the reference, simply call the event’s Broadcast method from the EventManager.

**Example 02:** Broadcasting mouse movement

*em.BroadcastMousePosition(keyboardAndMouseIndex, mousePosition);*

Where “*em*” is a reference to the EventManager, “*BroadcastMousePosition*” is the name of the broadcast method of the desired event, ”*keyboardAndMouseIndex*” is the controllerIndex of the keyboard and mouse pair as integer, and “*mousePosition*” is the position of the cursor on screen as Vector2.

### Creating new events

When creating a new event in EventManager, first declare the event. If necessary, create a new delegate type for the event. Then implement a broadcast method directly below the new event. Remember to add a check to prevent broadcasting in the case of zero subscribers. The events always start with “On” and the broadcast methods with “Broadcast”.

**Example 03:** Movement input

*public event IntVector2Void OnMovementInput;*

*public void BroadcastMovementInput(int controllerIndex, Vector2 movementInputVector)*

*{*

*if (OnMovementInput != null)*

*{*

*OnMovementInput(controllerIndex, movementInputVector);*

*}*

*}*

Where “*OnMovementInput*” is the event, “*BroadcastMovementInput*” is the broadcast-method of the event, and the parameters of the broadcast-method (determined by the delegate type of the event) are the parameters given when calling the subscribed methods.

### Delegates

Delegates are used when implementing new events in EventManager. A delegate is a type representing references to methods with particular parameter list and return type. EventManager delegates are named as follows: “Parameter1Parameter2…ParameterNReturnType”.

**Example 04**: IntIntVoid delegate

*public delegate void IntIntVoid(int integer1, int integer2);*

Where “*IntIntVoid*” is the name of the delegate, constructed from the parameter types.

## Game Manager

Manages game state and all gameplay related functions. Reference can be found through toolbox. Is a singleton and one always exists. Functions by sending and receiving events through EventManager.

When the application is launched, GameManager starts in the Main Menu’s default view. Most of the GameManager’s functions start only once a match is started, while the menus are handled by the UIManager. At the beginning of a match, the GameManager initializes the areana, by clearing any existing entities, placing power-ups, spawning ships and counting down the start timer. During the match, GameManager keeps track of the match timer, how many ships are still alive and once all but one of the ships are destroyed, it declares the winner.

All of the GameManagers functions are called as a result of an event being broadcasted (such as OnGameStart or OnShipDead). All of the actions managed by GameManager are then broadcasted forward (if needed) through the EventManager by event broadcasts.

### Initializing a match

When the “Level01” scene is fully loaded, an event “OnNewSceneLoaded” is broadcasted. This event is received by the GameManager, which then starts initializing a new match.

private void OnNewSceneLoaded(int sceneIndex)

{

if (sceneIndex == sceneIndexMainMenu)

{

}

else if (sceneIndex == sceneIndexLevel01)

{

inGame = true;

DestroyAllObjects();

resetUsedShipColors = true;

resetUsedSpawnPointsList = true;

InitializeGame();

StartMatchStartTimer();

}

}

private void InitializeGame()

{

matchStarted = false;

matchTimer = 0;

em.BroadcastMatchTimerValueChange(matchTimer);

#region Instantiate ships

currentlyAliveShipIndices.Clear();

for (int i = 0; i < numberOfShips; i++)

{

Transform spawnPoint = FindAvailableSpawnPoint();

GameObject newShip = Instantiate(Resources.Load("Ships/Ship", typeof(GameObject)), spawnPoint.position, spawnPoint.rotation) as GameObject;

Core\_ShipController newShipController;

Color newShipColor = FindNewShipColor();

if (currentGameModeIndex == gameModeSingleplayerIndex)

{

#region Singleplayer ship instantiating

if (i == 0)

{

newShipController =

newShip.AddComponent<Core\_LocalPlayerController>();

GameObject newPlayerIndicator =

Instantiate(Resources.Load("Effects/PlayerIndicator",

typeof(GameObject)), newShip.transform.position, Quaternion.identity,

newShip.transform) as GameObject;

ParticleSystem.MainModule pIMain = newPlayerIndicator.GetComponentInChildren<ParticleSystem>().main;

//Set playerIndicator color

pIMain.startColor = new Color(newShipColor.r, newShipColor.g, newShipColor.b, 1);

currentPlayerCamera = Instantiate(Resources.Load("Cameras/PlayerCamera",

typeof(GameObject)), Vector3.zero, Quaternion.identity) as GameObject;

Core\_CameraController currentCameraScript =

currentPlayerCamera.GetComponentInChildren<Core\_CameraController>();

currentCameraScript.SetTarget(newShip.transform);

currentCameraScript.SetMyShipIndex(i + 1);

}

else

{

newShipController =

newShip.AddComponent<Core\_AIPlayerController>();

}

//Set currentGameMode in shipController

newShipController.SetGameMode(currentGameModeIndex);

//Give ship an index and color

newShipController.GiveIndex(i + 1);

newShipController.SetShipColor(newShipColor);

#endregion

}

foreach (GameObject ship in currentlyAliveShips)

{

em.BroadcastShipReference(ship);

}

#endregion

#region Instantiate PowerUps

if (!powerUpsDisabled)

{

int powerUpBaseIndexCounter = 0;

foreach (Transform position in powerUpPositions)

{

powerUpBaseIndexCounter++;

GameObject newPowerUp = Instantiate(Resources.Load("PowerUps/PowerUpPlatform", typeof(GameObject)), position.position,

Quaternion.identity) as GameObject;

currentlyExistingPowerUps.Add(newPowerUp);

newPowerUp.GetComponent<Core\_PowerUpController>().SetPowerUpPlatformIndex(powerUpBaseIndexCounter);

}

}

#endregion

}

### Keeping track of match timer

Match timer is simply updated on every Update-loop and then broadcasted for the UIManager to display

private void Update()

{

if (inGame)

{

#region HUD Timer

if (matchStarted && !isPaused)

{

matchTimer += Time.deltaTime;

em.BroadcastMatchTimerValueChange(matchTimer);

}

#endregion

}

}

### Declaring match winner

The GameManager keeps track of the ships still alive and when only one ship remains, it broadcasts “OnMatchEnded” event, with the last ship’s index as the event parameter.

private void OnShipDead(int shipIndex)

{

if (currentlyAliveShipIndices.Count > 1)

{

currentlyAliveShipIndices.Remove(shipIndex);

if (currentlyAliveShipIndices.Count == 1)

{

em.BroadcastMatchEnded(currentlyAliveShipIndices[0]);

}

}

}

## Application Manager

Manages the scene loading and exiting the application. Reference can be found through toolbox. Is a singleton and one always exists. Functions by sending and receiving events through EventManager.

Functions very simply by receiving event broadcast through the EventManager, then starts loading a new scene and broadcasts “OnLoadingNewScene” with the new scene’s scene index as a parameter.

private void OnRequestSceneSingleLevel01()

{

Debug.Log("SceneManager: Received Level01 load request!");

em.BroadcastNewSceneLoading(sceneIndexLevel01);

//Load scene "Level01" in single mode

SceneManager.LoadScene(sceneIndexLevel01, LoadSceneMode.Single);

}

## Input Manager

Manages the input detection through keyboard / mouse controls when buildPlatform is set to PC. Reference can be found through toolbox. Is a singleton and one always exists.

Key and mouse button presses as well as mouse position are checked on the Update-loop and if a change is detected, it is then broadcasted to the EventManager using the proper event broadcast.

In the Android build, user inputs are handled by the UIManager (is menu interactions) and VirtualJoystickController (is movement, aim or shoot inputs).

void Update ()

{

#region KeyCode inputs

movementInputVector = Vector2.zero;

//right, up == 1, 1; left, down == 0, 0

if (Input.GetKey(KeyCode.W))

{

movementInputVector.y += 1;

}

if (Input.GetKey(KeyCode.A))

{

movementInputVector.x -= 1;

}

if (Input.GetKey(KeyCode.S))

{

movementInputVector.y -= 1;

}

if (Input.GetKey(KeyCode.D))

{

movementInputVector.x += 1;

}

//If movement vector's magnitude is not already 1

if (movementInputVector.x != 0 || movementInputVector.y != 0)

{

movementInputVector.Normalize();

}

if (movementInputVector != Vector2.zero)

{

em.BroadcastMovementInput(keyboardAndMouseIndex, movementInputVector);

movementZeroSent = false;

}

else if (!movementZeroSent)

{

em.BroadcastMovementInput(keyboardAndMouseIndex, movementInputVector);

movementZeroSent = true;

}

#endregion

#region Keys

if (Input.GetKeyDown(KeyCode.Escape))

{

em.BroadcastEscapeButtonDown(keyboardAndMouseIndex);

}

if (Input.GetKeyUp(KeyCode.Escape))

{

em.BroadcastEscapeButtonUp(keyboardAndMouseIndex);

}

#endregion

#endregion

#region Mouse movement

mousePosition = Input.mousePosition;

em.BroadcastMousePosition(keyboardAndMouseIndex, mousePosition);

#endregion

#region Mouse buttons

if (Input.GetMouseButtonDown(0))

{

em.BroadcastMouseButtonLeftDown(keyboardAndMouseIndex);

}

if (Input.GetMouseButtonUp(0))

{

em.BroadcastMouseButtonLeftUp(keyboardAndMouseIndex);

}

if (Input.GetMouseButtonDown(1))

{

em.BroadcastMouseButtonRightDown(keyboardAndMouseIndex);

}

if (Input.GetMouseButtonUp(1))

{

em.BroadcastMouseButtonRightDown(keyboardAndMouseIndex);

}

#endregion

}

## UI Manager

Manages the menu and HUD functionality. Reference can be found through toolbox. Is a singleton and one always exists. Functions by sending and receiving events through EventManager.

### Main Menu

When the application is launched, the UIManager starts in “MainMenuDefault” state. From there the user can access SettingView, GameModeView or exit the application. Parts of the MainMenu (mostly holders) exist constantly as disabled objects on the canvas, while most of them are instantiated as needed (buttons etc.)

UIManager contains methods for opening and closing each possible menu view, which are called when appropriate menu buttons are pressed. Opening any other main menu view, besides the default view, is handled internally in the UIManager, since the manager itself handles both the button interactions and opening / closing the menus. Only events broadcasted by the UIManager when in Main Menu, are GameMode selection, loading new scene (= starting a match), and exiting the application.

private void OpenMainMenuDefaultView()

{

mainMenuPlayButton = Instantiate(Resources.Load("UI/MainMenu/MainMenuButton", typeof(GameObject)), mainMenuCenter.transform) as GameObject;

mainMenuPlayButton.GetComponentInChildren<Text>().text = "PLAY";

mainMenuPlayButton.GetComponent<Button>().onClick.

AddListener(OnMainMenuPlayButtonPressed);

mainMenuExitButton = Instantiate(Resources.Load("UI/MainMenu/MainMenuButton", typeof(GameObject)), mainMenuCenter.transform) as GameObject;

mainMenuExitButton.GetComponentInChildren<Text>().text = "EXIT";

mainMenuExitButton.GetComponent<Button>().onClick.

AddListener(OnMainMenuExitButtonPressed);

mainMenuSettingsButton = Instantiate(Resources.Load("UI/MainMenu/MainMenuSettingsButton", typeof(GameObject)),

mainMenuRightSlotBot) as GameObject;

mainMenuSettingsButton.GetComponent<Button>().onClick.AddListener(OnMainMenuSettingsButtonPressed);

uiState = UIState.MAINMENUDEFAULT;

}

private void CloseMainMenuDefaultView()

{

if (mainMenuPlayButton != null)

{

mainMenuPlayButton.GetComponent<Button>().onClick.RemoveAllListeners();

Destroy(mainMenuPlayButton);

}

if (mainMenuExitButton != null)

{

mainMenuExitButton.GetComponent<Button>().onClick.RemoveAllListeners();

Destroy(mainMenuExitButton);

}

if (mainMenuSettingsButton != null)

{

mainMenuSettingsButton.GetComponent<Button>().onClick.RemoveAllListeners();

Destroy(mainMenuSettingsButton);

}

}

UI button presses are detected by using Unity’s UI event system. When a button is created, a listener (= any method, specific for each button) is added to its “onClick” event. When the event system detects a button press, the listener method is called.

private void OnMainMenuGameModeSinglePlayerButtonPressed()

{

em.BroadcastSetGameMode(gameModeSingleplayerIndex);

OpenLoadingScreen();

em.BroadcastRequestSceneSingleLevel01();

}

### In-game HUD

In-game HUD is opened when the “Level01” scene is loaded. Parts of the HUD exist at all times (holders, loadingscreen), most are loaded when the HUD is opened (buttons, match-timer). On Android platform, the HUD layout is depended on the “InvertedHUD” setting of the settings view.

On PC platform, the HUD contains only the match-timer and the “OffsceenTargetIndicators”. On Android platform the HUD also contains an UI button for opening the pause menu and virtual joysticks for controlling the player ship.

Offscreen Target Indicators indicate the direction and distance of the other ships in the match if the ships are outside of player screen-area. Indicators are instantiated when the player controlled ship receives references to the other ships (broadcasted by GameManager once all ships are instantiated). Indicator location / scale update method is called on the UIManager’s Update-loop.

private void FollowOffscreenTargets()

{

if (offscreenIndicatorTargets.Count > 0)

{

for (int i = 0; i < offscreenIndicatorTargets.Count; i++)

{

if (offscreenIndicatorTargets[i] == null)

{

offscreenIndicatorTargets.RemoveAt(i);

//Destroy obsolete indicators

if (offscreenIndicatorPool.Count >= offscreenIndicatorTargets.Count)

{

if (offscreenIndicatorPool[offscreenIndicatorTargets.Count].

gameObject != null)

{

Destroy(offscreenIndicatorPool[offscreenIndicatorTargets.Count].gameObject);

}

offscreenIndicatorPool.RemoveAt(offscreenIndicatorTargets.Count);

}

i--;

}

else

{

//If not enough indicators, create a new one

if (offscreenIndicatorTargets.Count > offscreenIndicatorPool.Count)

{

GameObject newOffscreenIndicator = Instantiate(Resources.Load("UI/HUD/HUDOffscreenIndicator", typeof(GameObject)), offscreenIndicatorHolder.position, offscreenIndicatorHolder.rotation, offscreenIndicatorHolder) as GameObject;

Transform newOffscreenIndicatorTransform = newOffscreenIndicator.transform;

offscreenIndicatorDefaultPosition = newOffscreenIndicatorTransform.position;

offscreenIndicatorPool.Add(newOffscreenIndicatorTransform);

newOffscreenIndicator.SetActive(false);

}

Transform target = offscreenIndicatorTargets[i];

Transform indicator = offscreenIndicatorPool[i];

Vector3 screenPosition = Camera.main.WorldToViewportPoint(target.position);

if (screenPosition.x >= (-0.08f + offscreenIndicatorSidebuffer) && screenPosition.x <= (1.08f - offscreenIndicatorSidebuffer) &&

screenPosition.y >= -0.08f && screenPosition.y <= 1.08f)

{

//Target is within screenspace

indicator.gameObject.SetActive(false);

indicator.position = offscreenIndicatorDefaultPosition;

}

else

{

//Target is outside of screenspace

if (!indicator.gameObject.activeSelf)

{

indicator.gameObject.SetActive(true);

}

Color indicatorColor = target.GetComponent<Core\_ShipController>().GetShipColor();

indicatorColor.a = indicator.GetChild(0).GetComponent<Image>().color.a;

indicator.GetChild(0).GetComponent<Image>().color = indicatorColor;

screenPosition.x -= 0.5f;

screenPosition.y -= 0.5f;

screenPosition.z = 0;

float angle = Mathf.Atan2(screenPosition.x, screenPosition.y);

screenPosition.x = 0.5f \* Mathf.Sin(angle) + 0.5f;

screenPosition.y = 0.5f \* Mathf.Cos(angle) + 0.5f;

screenPosition.z = Camera.main.nearClipPlane + 0.01f;

indicator.localEulerAngles = new Vector3(0.0f, 0.0f, -angle \* Mathf.Rad2Deg);

screenPosition.x = Mathf.Clamp(screenPosition.x, (0 + offscreenIndicatorSidebuffer), (1 - offscreenIndicatorSidebuffer));

indicator.position = Camera.main.ViewportToScreenPoint(screenPosition);

#region Indicator modifications based on target distance

//Calculating a factor depending on target distance, to modify indicators accordingly

Vector3 screenCenterInWorldSpace = Camera.main.ViewportToWorldPoint(new Vector3(0, 0, 0));

Vector3 cameraPositionOnArena = new Vector3(screenCenterInWorldSpace.x,

screenCenterInWorldSpace.y - 30, screenCenterInWorldSpace.z);

Debug.DrawRay(cameraPositionOnArena, Vector3.up);

float indicatorTargetDistance = Vector3.Distance(cameraPositionOnArena, target.position);

float maxIndicatorTargetDistance = 60;

float minIndicatorTargetDistance = 20;

float minIndicatorDistanceFactorValue = 0.2f;

float maxIndicatorDistanceFactorValue = 1;

indicatorTargetDistance = Mathf.Clamp(indicatorTargetDistance, minIndicatorTargetDistance,

maxIndicatorTargetDistance);

indicatorTargetDistance -= minIndicatorTargetDistance;

float indicatorTargetDistanceFactor = indicatorTargetDistance /

(maxIndicatorTargetDistance - minIndicatorTargetDistance);

indicatorTargetDistanceFactor = Mathf.Clamp((1 - indicatorTargetDistanceFactor),

minIndicatorDistanceFactorValue, maxIndicatorDistanceFactorValue);

//Horizontal size scaling depending on target distance

RectTransform indicatorRectTransform = indicator.GetChild(0).GetComponent<RectTransform>();

indicatorRectTransform.localScale = new Vector3(indicatorTargetDistanceFactor, 1, 1);

#endregion

}

}

}

}

}

### Pause Menu

Pause Menu works somewhat similarly to Main Menu. Pause Menu holder exists at all times, but the buttons are instantiated whenever the menu is opened and destroyed when it’s closed. Pause Menu contains a panel title (“Pause Menu”), and three buttons, “Resume”, “Restart” and “Main Menu”, which either close the menu, restart the match or return the user to the main menu, respectively.

Pause Menu can be opened any time during a match, by pressing the Escape button of the keyboard (if on PC) or the UI button for opening the menu (if on Android). It is automatically opened if the user presses the Return button on Android platform (functions essentially as an Escape button). While the menu is open, the game is paused. The Pause Menu can be closed when open, by pressing the Escape button (if on PC) or by pressing the “Resume” button on the Pause Menu (PC and Android).

### Game End Menu

Game End Menu is essentially the pause menu, with the panel title and button order changed. Game End Menu contains the panel title (reading either “Victory” or “Defeat”) and two buttons: “Restart” and “Main Menu”. When the game menu is opened, the game is paused like when the Pause Menu is open.

## Global Variable Library

Contains all the variables of the game. Reference can be found through toolbox. Is a singleton and one always exists. Functions by sending and receiving events through EventManager.

GlobalVariableLibrary (or GVL) is a library of all the variables of the game gathered in one place for easy access. All scripts that require any preset values for variables, read the desired values from GVL in their Awake-method. The variables in GVL are public and accessible through the Unity’s inspector panel. GVL contains separate categories (classes) for different types of variables, such as “ShipVariables” or “ProjectileVariables”.

Whenever multiple variables would logically fit in their separate category, a new category should be implemented, to keep the GVL in easily readable order.

## Ship Controller

Ship Controller is a base-class of the ships’ controllers. It contains all the variables and methods needed for a single ship to function in a match. Methods for functionalities such as moving, aiming, shooting, taking damage, dying, etc. are included. Specific commands for the ship controller are given by the specified Player Controllers, which inherit the Ship Controller base-class.

### Local Player Controller

Local Player Controller subscribes to the input events of the EventManager, transforms the input events to commands and calls the suitable methods from the Ship Controller.

### AI Player Controller

AI Player Controller decides proper actions based on its functionality and then calls necessary commands from the Ship Controller. Currently the AI is very simple:

* Target the closest ship
* If the target is within the shootingRange, shoot at the target’s direction
* If my distance to the target is longer than the desiredDistanceToTarget, start moving towards the target’s current position
* If any other ship is closer to me than my current target for longer than closestTargetTimerDuration, make that ship my new target
* If I have not changed direction within the last changeDirectionTimerDuration seconds, randomize new direction
* Move straight in the current direction
* If a wall is encountered, calculate reflection, and make that my new direction

## Other Controllers

Other controllers of the game control single entities within the game and are created and used whenever necessary. For example, Camera Controller controls the camera by following the local player’s controlled ship, when Projectile Controller instead handles a single projectile’s movement and collision detection.

## Scene Infos

Scene Infos contains scene specific info, like a list of scene specific spawn-points. Scene Info scripts are also responsible for broadcasting an “OnNewSceneLoaded” event when a new scene is successfully loaded.

## Tag-scripts

Tag scripts’ do not actually contain any code and exist only to function as tags for specific gameObjects in cases where using the Unity’s internal tag-system is not preferable.

# Pausing the game

Pausing the game is handled by implementing a check for the state of a Boolean variable “isPaused” on all methods which move any entity, count down timers, etc. In addition to cheking the boolean state, when the game is paused, Unity’s internal timescale will be set to zero, to ensure the pausing of effects, physics etc.

# Naming conventions

All asset names in the project should use the “CamelCase” practise, mixed with underscores as limiters when necessary. In CamelCase each word (or an abbreviation of a word) starts with a capital letter and there are no spaces or punctuation. Assets should be numbered if multiple assets of the same name would otherwise exist. All numbering should be expect to be able to continue at least till 99 and therefore numbers below 10 should start with a zero (e.i. 01, 02, 03… 09, 10)

For example: A texture for a generic menu button should be named: “MenuButtonBackground01\_tex”, where “tex”-postfix denotes the type of the asset.

## Postfixes

Names of some asset types should end with a type-specific postfix. List of current postfixes are below. If an asset type is missing from the list, new postfix should be created and added to the documentation if deemed necessary (consult your project head in unclear cases).

Scripts None

Shaders None

Prefabs None

Textures tex

Materials mat

Models mod

# Android support

Android support is mostly handled by implementing an integer variable “buildPlatform” where 0 = PC and 1 = Android. All relevant scripts (UIManager!) should check this variable and modify their functionality accordingly. For example, if the buildPlatform is set to PC, UIManager does not instantiate the virtual joysticks or the UI button for opening pause menu in the HUD when a match is started, as it would if the buildPlatform was set to Android.

Other things to consider regarding the Android build are: keeping the game’s graphical style simple and graphical assets’ file size small. Designing and implementing intuitive controls for both PC and Android platforms (VirtualJoysticks vs Keyboard / Mouse).

Building the Android version is handled by Unity’s build tool.

# Other guidelines

## Tags

Unity’s internal tag-system should be used sparsely. While the limit of the tag count is 10 001, bloating the tag-manager with unnecessary and obsolete tags would complicate the tag usage too much. When multiple gameObjects would be using a single tag, or the hierarchy under which the gameObject is placed is unknown, the internal tag-system should be used. When a tag is needed for individual gameObject, or the gameObject(s) exist under a known parent object “tag-scripts” should be used. Tag-scripts do not actually contain any code, and only exist to function as a component which can easily be searched with for example the “GetComponentInChildren” or “FindObjectOfType” methods.

## Graphical guidelines

The graphics of the game should remain quite simple. Ship models can be upgraded with actual models (instead of using Unity’s default models), if deemed necessary and recourses allow. Textures should be kept to a minimum, and when used, use 512x512px as a guideline for max size. Single-colored materials with Unity’s default particle-shaders should be used on most 3d assets.

# Asset list

## Ship

Model (Not necessary, using Unity’s default models)

Textures (Not necessary, using single-colored materials)

Materials

* Colorable parts
* Non-colorable parts

## Projectile

Model (Not necessary, using Unity’s default models)

Textures (Not necessary, using single-colored materials)

Materials

* Colorable parts

## Wall

Model (Not necessary, using Unity’s default models)

Textures (Not necessary, using single-colored materials)

Material

* Colorable parts

## Floor

Model (Not necessary, using Unity’s default models)

Textures (Not necessary, using single-colored materials)

Material

* Grid shader

## UI

Textures

* MainMenuButtonLongBackground
* MainMenuButtonSquareBackground
* MainMenuTitle
* LoadingScreen
* HealthBar (Circular)
* OffscreenTargetIndicator (Triangle)

# Version control

Version control of the project is handled through a GitHub repository. Primary Git client is SourceTree. “Master” branch contains a working PC version, “Android” branch is reserved for a working Android version. No broken features should ever be committed to the repository (e.i. all repository commits should be successfully run and be buildable if needed). Committing WIP features is permitted, as long as they do not break any existing features or interfere with running the application.

## **Link to repository**

<https://github.com/Taikatassu/SCRMG>