Technical Development Document

Meta Spark AR - Gamified use of face tracking

[**Project Overview 1**](#_rvsz6g6ynsfp)

[Game title 1](#_50w4s46kwa4v)

[Team 2](#_ulh3oppmbp9f)

[Environment 2](#_cwzcdt5c1dzg)

[Alt. 2](#_oqcqsbceqpte)

[**Project Details 2**](#_526innzjkwc)

[Introduction 2](#_v5bl22fofnfm)

[Purpose 3](#_w4o3jbgu61gg)

[**Technical Architecture 3**](#_66fyf6jxbxlc)

[Flowchart 3](#_1bmvnbxtcqvz)

[State Management 3](#_bo6r97l7ggqc)

[Scripts & Patches 3](#_dfn9b098mhag)

[Additional Notes 11](#_1gtei4kdo5jb)

[**Development Details 11**](#_sh6dune9ddsd)

[Current Tasks 11](#_8esl8fh6fwh9)

[Bugs & Issues 11](#_wqa61fso0dx8)

[**Test Plan 12**](#_icibbt57xdo0)

[Test Strategy 13](#_b41k2opvokvo)

[Test Environment 13](#_it22p8ecujqi)

[Test Cases 13](#_u5ag059omrlk)

[**Asset Management 14**](#_b122i8t2afbt)

[Art Assets 14](#_s0dt82bic267)

[**Appendices 14**](#_rbdgmpaj7kv7)

# Project Overview

## Game title

SingaBall

## Team

Supervisor: Thomas Koh

Developer: Winston Chiu

## Environment

Meta Spark Studio -> Singleplayer Testing with Meta Spark Mobile Player

Visual Studio Code -> JavaScript & Reactive Code

Plugins: <https://spark.meta.com/learn/scripting/vs-code-extension>

Github: ARISENTU/meta-spark-Singaball

Note: Learn about reactive programming in order to continue with the project

<https://spark.meta.com/learn/scripting/reactive-programming>

<https://spark.meta.com/learn/patch-editor/bridging>

Note the project export file types used: <https://spark.meta.com/learn/articles/fundamentals/project-file-formats>

### Alt.

Meta sparks studio & meta spark ar is a software developed by meta as part of their social media engagement software suite. This software was made to create facebook, instagram & messenger compatible filters, en masse. This was then published for free for the general public to access & create their own filters, either personal, profit driven or engagement. Due to the need for ease of access & fast learning, it has 2 main methods of development, script based for developers & patch editor for visual scripting. They are also interchangeable & linked together via its accompanying plugin. It is for this reason that games & gamified filters can be created, & by many people nonetheless. With such attention & interest, Meta has released official [documentation & guides](https://spark.meta.com/learn/). These effects are compatible with a large audience & can be published to the meta servers under SkyLight. This means that publishing & deployment is easy & accessible by nearly everyone.

# Project Details

## Introduction

SingaBall:

Health based game using head tracking to control a glove on screen. Using the glove, hit a ball at the bricks on the screen. The brick targetted is not in the player’s control. By breaking down all the bricks, the next level appears. After a few levels have been completed, the boss shows up and fires projectiles back to you. Interaction with the game relies on face tracking, specifically the nose, and the angle of the head.

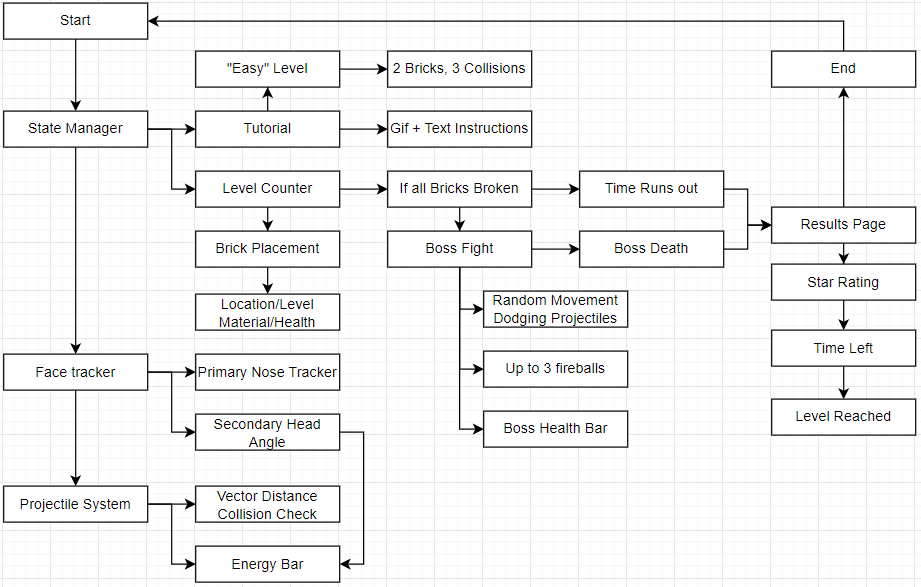
A game similar to brick breaker and table tennis, with a paddle attached to the nose.

## Purpose

The project is a gamified filter with face tracking capabilities. The target audience is for all ages, with focus on the elderly interacting with the youth in video calls. This means that a software compatible with & easily published within a video call service is preferable.

# Technical Architecture

## Flowchart



## State Management

Effect start

State 0: Instruction/Tutorial, 5s, tap to continue

State 1: Gameplay, 120s

State 2: Results, 5s, tap to continue

Effect ends

## Scripts & Patches

GameManager.Js

| **import Time from 'Time' import Patches from 'Patches' import Reactive from 'Reactive' import Diagnostics from 'Diagnostics'  const Scene = require('Scene');  const targetFrames = 20; const timeInterval = 1 / targetFrames;  let currState = 0; *//compare internal script state to the patch state, if different* let s\_GameState = Reactive.val(0);  async function getGameState() {  s\_GameState = (Patches.outputs.getScalarOrFallback('setGameState'));  *//if currstate not gamestate then send a pulse to the patch*  if (s\_GameState.pinLastValue() != currState) {  currState = s\_GameState.pinLastValue();  await Patches.inputs.setPulse('p\_ResetTime', Reactive.once()); *// Send a pulse to indicate the state change*  Diagnostics.log(`Game State: ${currState}`);  }  *//output the state regardless*  await Patches.inputs.setScalar('s\_GameState', s\_GameState.pinLastValue()); }  async function convertTimeToStar() {  *//using hard coded thresholds, return a rating to light up the stars*  let totalTime = (Patches.outputs.getScalarOrFallback("getTime"));  let rating = 0;   *//0 stars*  if (totalTime.pinLastValue() <= 0) { rating = 0; }  *//1 stars*  else if (totalTime.pinLastValue() <= 20) { rating = 1; }  *//2 stars*  else if (totalTime.pinLastValue() <= 45) { rating = 2; }  *//3 stars*  else { rating = 3; }   Patches.inputs.setScalar('StarRating', rating) }  async function GetGameTime() {  let totalTime = (Patches.outputs.getScalarOrFallback("getTime"));  let minutes = Math.floor(totalTime.pinLastValue() / 60);  let seconds = Math.floor(totalTime.pinLastValue() % 60);  *// Format minutes and seconds into MM:SS*  let formattedTime = `${String(minutes).padStart(2, '0')}:${String(seconds).padStart(2, '0')}`;  Patches.inputs.setString('GameTime', formattedTime); }  async function IsEnerygyFull() {  let EnergyFull = (Patches.outputs.getBooleanOrFallback('FullEnergy'));  await Patches.inputs.setBoolean('StopChargingEnergy', EnergyFull.pinLastValue()); }  (async function () { *// Enables async/await in JS [part 1]*  function CaughtBall() {  Patches.inputs.setPulse('EnergyUsed', Reactive.once());  }  let PlayerCatchBall = (Patches.outputs.getPulseOrFallback('PlayerCatchBall'));  let PlayerCatchFireball0 = (Patches.outputs.getPulseOrFallback('PlayerCatchFireball0'));  let PlayerCatchFireball1 = (Patches.outputs.getPulseOrFallback('PlayerCatchFireball1'));  let PlayerCatchFireball2 = (Patches.outputs.getPulseOrFallback('PlayerCatchFireball2'));  PlayerCatchBall.subscribe(CaughtBall);  PlayerCatchFireball0.subscribe(CaughtBall);  PlayerCatchFireball1.subscribe(CaughtBall);  PlayerCatchFireball2.subscribe(CaughtBall);   async function Update() {  getGameState();  IsEnerygyFull();   s\_GameState = (Patches.outputs.getScalarOrFallback('setGameState'));  if (s\_GameState.pinLastValue() == 1) {  GetGameTime(); *//update result's ingame time text*  convertTimeToStar(); *//dont change the star rating during results*  }  }   const intervalTimer = Time.setInterval(Update, timeInterval \* 1000); })();** |
| --- |

getGameState is a function that updates the internal game state value, if the value is not the same, then the script knows that it has been changed, or updated. As such, it sends out a pulse to reset the timer. This means that each state has a consistent max timer(other than an external condition that changes the state before the timer runs out), ensuring loopable state changes.

convertTimetoStar is a function that returns a star rating based on the time taken, which is converted from total time to a hybrid minute:seconds timer. The star rating affects the final result page stars, however they are hardcoded and have no visible formula for further scaling. The seconds left are in part 0, 20, 45. Between 1 and 19, 1 star. Between 20 and 44, 2 stars. 45 seconds and above, 3 stars, and a “Perfect” sticker.

GetGameTime is a function that converts time in seconds, i.e. total time, into minute:seconds format, similar to a digital clock.

IsEnergyFull are simple getter/setter functions meant to convert time and energy values in game, to usable forms by engaging the script to update global values. This circumvents looped code structures and thus recursive calls.

Lastly, the update function. A large subscribe chain has been implemented to watch the value of collisions, then trigger a generic collision response, such as CaughtBall, which deducts energy spent. Then, it leads to a generic per-frame updater for getting time, checking if energy is full and triggering stop.

BallBrickManager.js

| **import** Time **from** 'Time' **import** Patches **from** 'Patches' **import** Reactive **from** 'Reactive' **import** Diagnostics **from** 'Diagnostics'  **const** Scene = require('Scene');  **const** targetFrames = 20; **const** timeInterval = 1 / targetFrames;  *//ball* **async** **function** **checkcatchball**() {  **let** playercatch = (Patches.outputs.getBooleanOrFallback('Player\_catch'));  **if** (playercatch.pinLastValue() == true) {  **let** BallProg = (Patches.outputs.getScalarOrFallback('BallProg'));  **if** (BallProg.pinLastValue() == 1) {  **await** Patches.inputs.setPulse('PlayBallAnimation', Reactive.once());  **await** Patches.inputs.setPulse('ReverseBallDirection', Reactive.once());  }  } }  **async** **function** **RebounceWall**() {  **let** resetBallProgress = (Patches.outputs.getBooleanOrFallback('ResetBallProgress'));  **if** (resetBallProgress.pinLastValue() == true) {  **await** Patches.inputs.setPulse('PlayBallAnimation', Reactive.once());  **let** BallProg = (Patches.outputs.getScalarOrFallback('BallProg'));  **if** (BallProg.pinLastValue() == 0) {  **await** Patches.inputs.setPulse('UnReverse', Reactive.once());  }  } }  *//brick* **async** **function** **GeneratePosition**() {  **let** GeneratePosition = (Patches.outputs.getBooleanOrFallback('BGenerateRandomPos'))  **if** (GeneratePosition.pinLastValue() == true) {  **await** Patches.inputs.setPulse('PF\_GeneratePosition', Reactive.once());  } }  **async** **function** **updateBrickTarget**() {  **let** isIncrementing = (Patches.outputs.getBooleanOrFallback('BIncrementBrickTarget'));  **if** (isIncrementing.pinLastValue() == true) {  **await** Patches.inputs.setPulse('PF\_IncrementBrickTarget', Reactive.once());  }   **let** InactiveBrick = (Patches.outputs.getBooleanOrFallback('BInactiveBrick'));  **if** (InactiveBrick.pinLastValue() == true) {  **await** Patches.inputs.setPulse('PF\_IncrementBrickTarget', Reactive.once());  } }  **async** **function** **UpdateBrickLives**() {  **let** BrickLives = (Patches.outputs.getScalarOrFallback('N\_BrickLives'));  **await** Patches.inputs.setScalar("NF\_BrickLives", BrickLives.pinLastValue()); }  **async** **function** **IncreaseLevel**() {  **let** LevelIncreasing = (Patches.outputs.getBooleanOrFallback('BIncreaseLevel'));  **if** (LevelIncreasing.pinLastValue() == true) {  **await** Patches.inputs.setPulse('PF\_IncreaseLevel', Reactive.once());  } }  (**async** **function** () {  **async** **function** **Update**() {  **if** ((Patches.outputs.getScalarOrFallback('setGameState')).pinLastValue() == 1) {  *//ball*  checkcatchball();  RebounceWall();  *//brick*  UpdateBrickLives();  updateBrickTarget();   GeneratePosition();  IncreaseLevel();  }  }   **const** intervalTimer = Time.setInterval(Update, timeInterval \* 1000); })(); |
| --- |

CheckCatchBall is a function that handles the response of the collisions done via collision detection and viability of collision, i.e. whether the ball is in place and movement has been completed.

RebounceWall is a function that spawns the ball at the brick and triggers forward movement, this is using a pulse that triggers animation in reverse. This reverses the progress value outputted, which can be used for a bouncing-like

UpdateBrickTarget is a function that checks if the brick has broken, usually with 0 HP remaining. This then triggers an increment in the target number.

UpdateBrickLives is a function that checks for a collision in the brick and updates the HP of the brick, which also updates the UpdateBrickTarget function if applicable.

IncreaseLevel is a function that follows the above function behavior, but reflects the level instead. When all the bricks in the level are broken, the next level is created.

Lastly, the Update function is triggered every frame while there is a GameOver check as a breakpoint.

BossManager.js

| **import** Time **from** 'Time' **import** Patches **from** 'Patches' **import** Reactive **from** 'Reactive' **import** Diagnostics **from** 'Diagnostics'  **const** Scene = require('Scene');  **const** targetFrames = 20; **const** timeInterval = 1 / targetFrames;  **async** **function** **StartBossFight**() {  **let** startbossfight = (Patches.outputs.getBooleanOrFallback('BStartBossFight'));  **if** (startbossfight.pinLastValue() == true) {  Patches.inputs.setPulse('BF\_RandomizeBossDestination', Reactive.once());  Patches.inputs.setPulse('MoveBoss', Reactive.once());  } }  **async** **function** **CheckBossProgress**() {  **let** BossProgress = (Patches.outputs.getScalarOrFallback('BossProgress'));  **let** FinishedAttacking = (Patches.outputs.getBooleanOrFallback('BossFinishedAttacking'));   **if** (BossProgress.pinLastValue() == 1 && FinishedAttacking.pinLastValue() == true) {  Patches.inputs.setPulse('ResetBossAnimation', Reactive.once());  Patches.inputs.setPulse('BF\_RandomizeBossDestination', Reactive.once());  Patches.inputs.setPulse('MoveBoss', Reactive.once());  } }  **async** **function** **GetBossHealth**() {  **let** N\_BossHealth = (Patches.outputs.getScalarOrFallback('N\_BossHealth'));  Patches.inputs.setScalar('FBossHealth', N\_BossHealth.pinLastValue()); }  **async** **function** **CatchFireBalls**() {  **let** fireballstatus0 = (Patches.outputs.getBooleanOrFallback('CatchFireBall\_0'));  **let** fireballstatus1 = (Patches.outputs.getBooleanOrFallback('CatchFireBall\_1'));  **let** fireballstatus2 = (Patches.outputs.getBooleanOrFallback('CatchFireBall\_2'));   **let** fireballprog0 = (Patches.outputs.getScalarOrFallback('FireBallprog0'));  **let** fireballprog1 = (Patches.outputs.getScalarOrFallback('FireBallprog1'));  **let** fireballprog2 = (Patches.outputs.getScalarOrFallback('FireBallprog2'));   **if** (fireballstatus0.pinLastValue() == true && fireballprog0.pinLastValue() == 1) {  Patches.inputs.setPulse('ReverseFireBall\_0', Reactive.once());  Patches.inputs.setPulse('PlayFireBall\_0', Reactive.once());  }   **if** (fireballstatus1.pinLastValue() == true && fireballprog1.pinLastValue() == 1) {  Patches.inputs.setPulse('ReverseFireBall\_1', Reactive.once());  Patches.inputs.setPulse('PlayFireBall\_1', Reactive.once());  }   **if** (fireballstatus2.pinLastValue() == true && fireballprog2.pinLastValue() == 1) {  Patches.inputs.setPulse('ReverseFireBall\_2', Reactive.once());  Patches.inputs.setPulse('PlayFireBall\_2', Reactive.once());  } }  (**async** **function** () {  **async** **function** **Update**() {  **if** ((Patches.outputs.getScalarOrFallback('setGameState')).pinLastValue() == 1) {  StartBossFight();  CheckBossProgress();  GetBossHealth();  CatchFireBalls();  }  }   **const** intervalTimer = Time.setInterval(Update, timeInterval \* 1000); })(); |
| --- |

StartBossFight is a trigger function and a randomizer. The boss is spawned in a random part of the screen, then moves everytime it shoots a fireball. This makes it seem like the fireballs curve while flying, but instead it follows a straight movement relative to both positions of the boss and target.

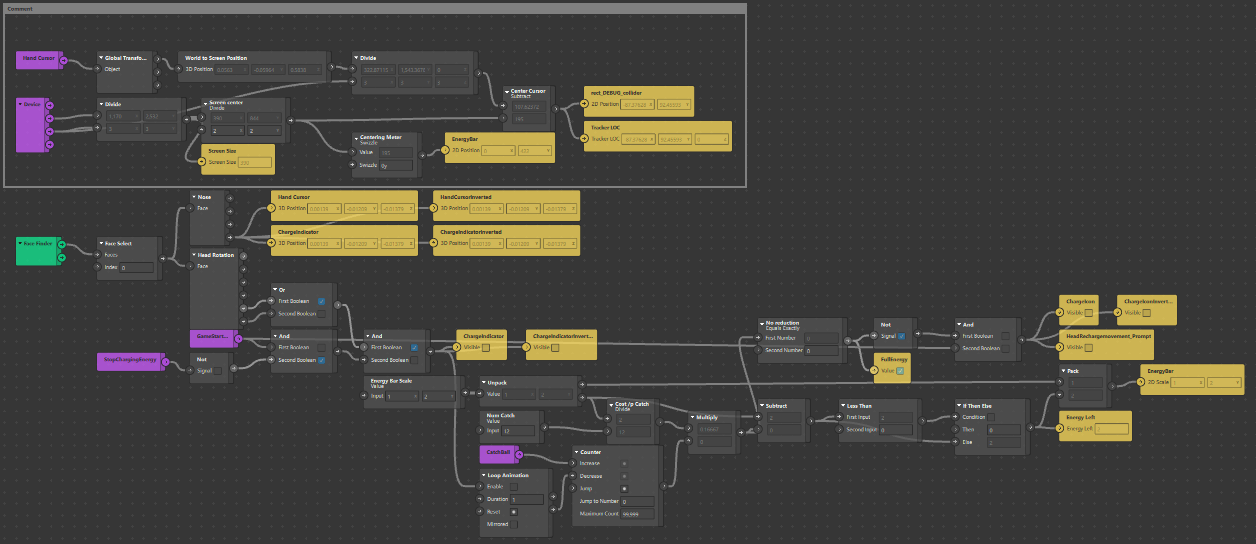
CheckBossProgress is a function that is updated similarly to the above function, in that it is triggered during the boss fight, as opposed to the “starter trigger” nature of StartBossFight.

GetBossHealth is a simple getter/setter for the boss health, which delegates the boss health to a global value, bypassing recursive calling.

CatchFireBalls is both a check and trigger function for collision response. The fireball used contains collision requirements, as long as both are triggered, the fireball will “bounce” back at the boss to damage it.

Lastly, the Update function also has a breakpoint in GameOver.

Hand Movement:

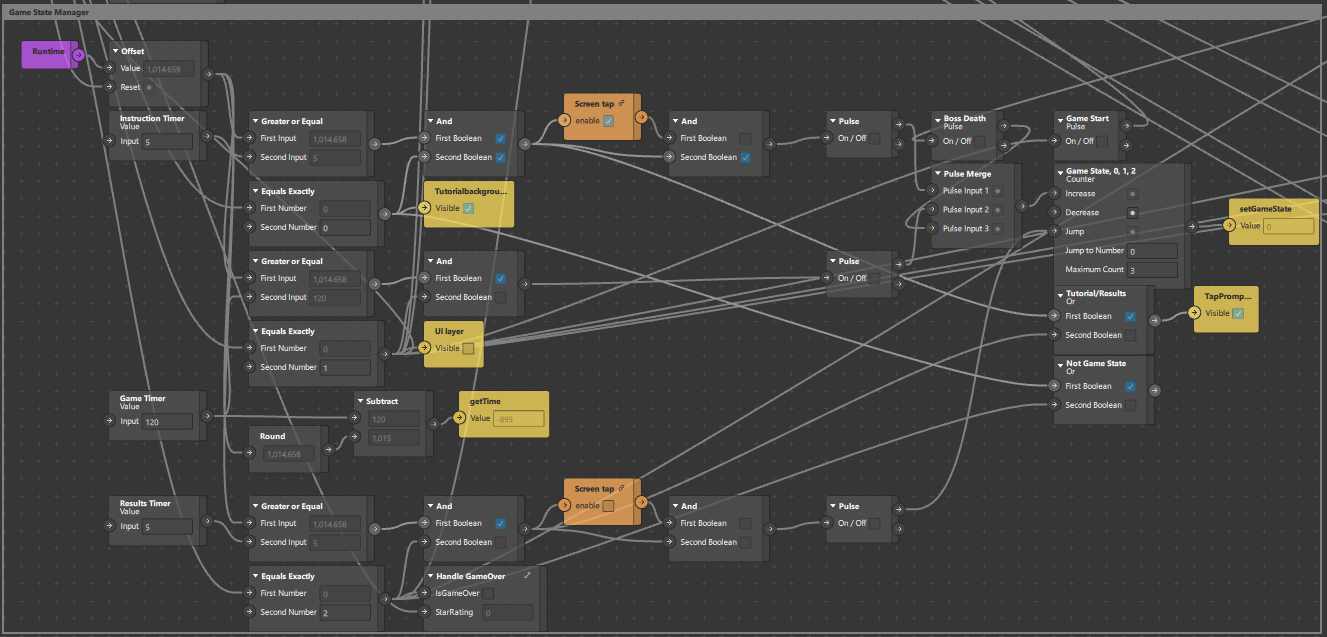


This is a patch group that handles the input of the face tracker, then imposes a cursor onto the face. This is primarily nose tracking, with a secondary tracking in the head position, focussed on leaning forwards and backwards.

Note: the original only allowed for leading back, but field testing has shown that those with neck injuries either have a brace that blocks back leaning, or have a stiff neck. It is, thus, easier to account for forward leaning, such that the user can lean their entire body forwards.

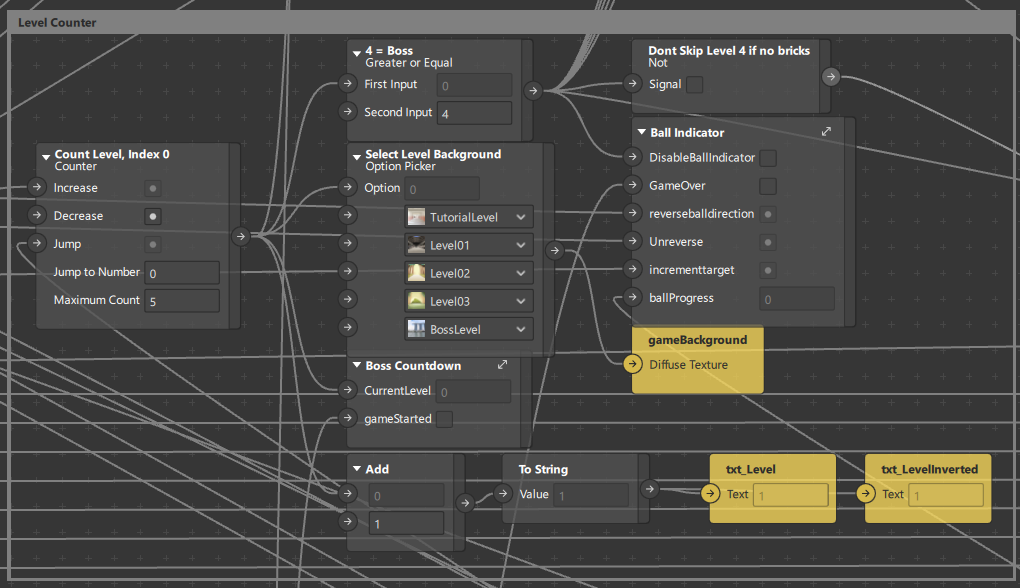
This patch also handles recharging of the energy meter, preventing collision if the meter is empty.

Gamestate manager:



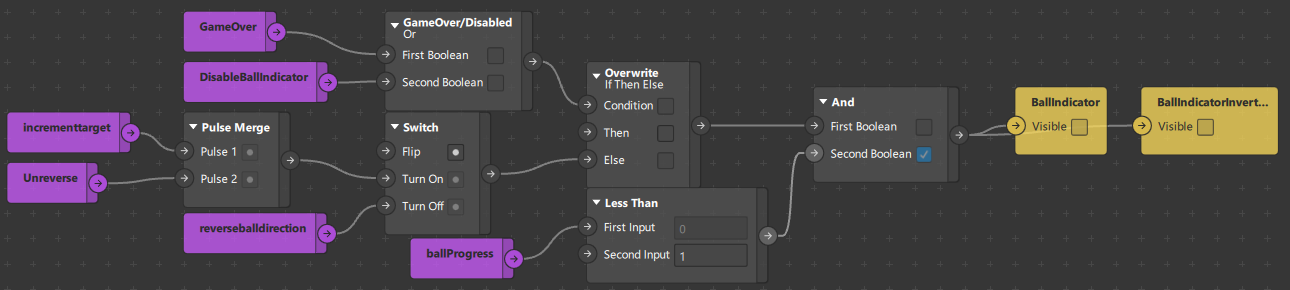
Using a runtime patch to track time since the effect has been launched, then using an external pulse called \_ResetTime to trigger offsets within the time. This allows for the time tracker to reset every time the game state has changed. This means that each state has a maximum time or an optional early exit. The optional exit is either a screen tap or the early death of the boss. This is fed into a counter patch to track the current game state, with a jump to reset to state 0 when the results page is complete.

Level Counter:



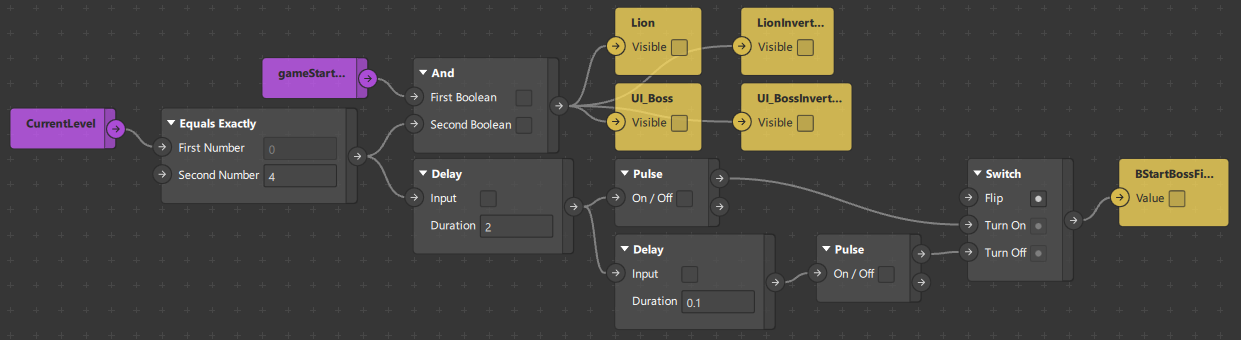
This group is a simplified switch case with a global numerical output. The output triggers various values such as changing the background for each level, each being a texture imposed on a material. The levels are (tutorial, level1, level2, level3, boss fight). The output value also triggers a bossfight as well as incrementing itself. When all bricks are broken, the game will automatically increment levels, which in this case, ignore the value 4, as it is the boss fight, i.e. there are no bricks, which means there is a required exception if not the boss fight will not be triggered.

Ball indicator:



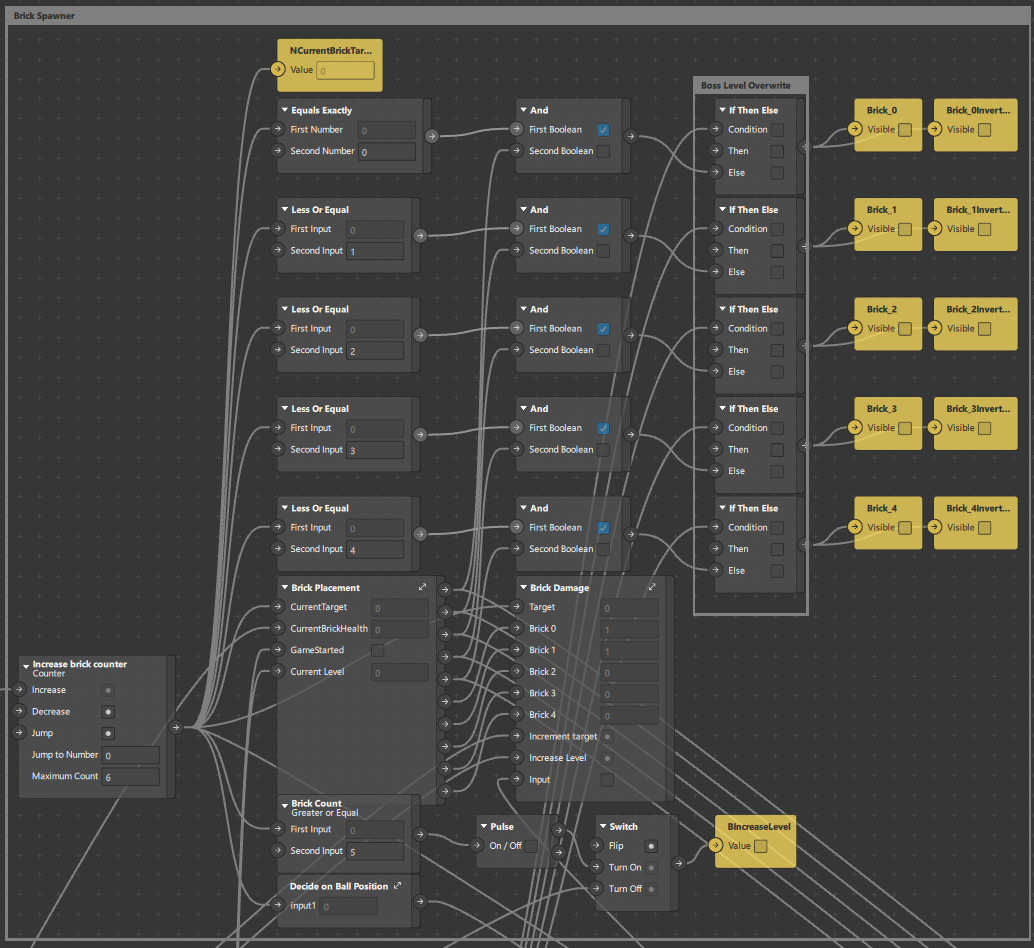
A ball indicator is shown whenever the ball reaches the screen, during the movement. When the ball reaches its destination, the indicator disappears, if not it will be rendered over the ball itself. When the ball is moving back to the screen, the reverse direction is triggered, which also prevents the indicator from being rendered. This is also overwritten by a GameOver/Disabled check.

Boss countdown:



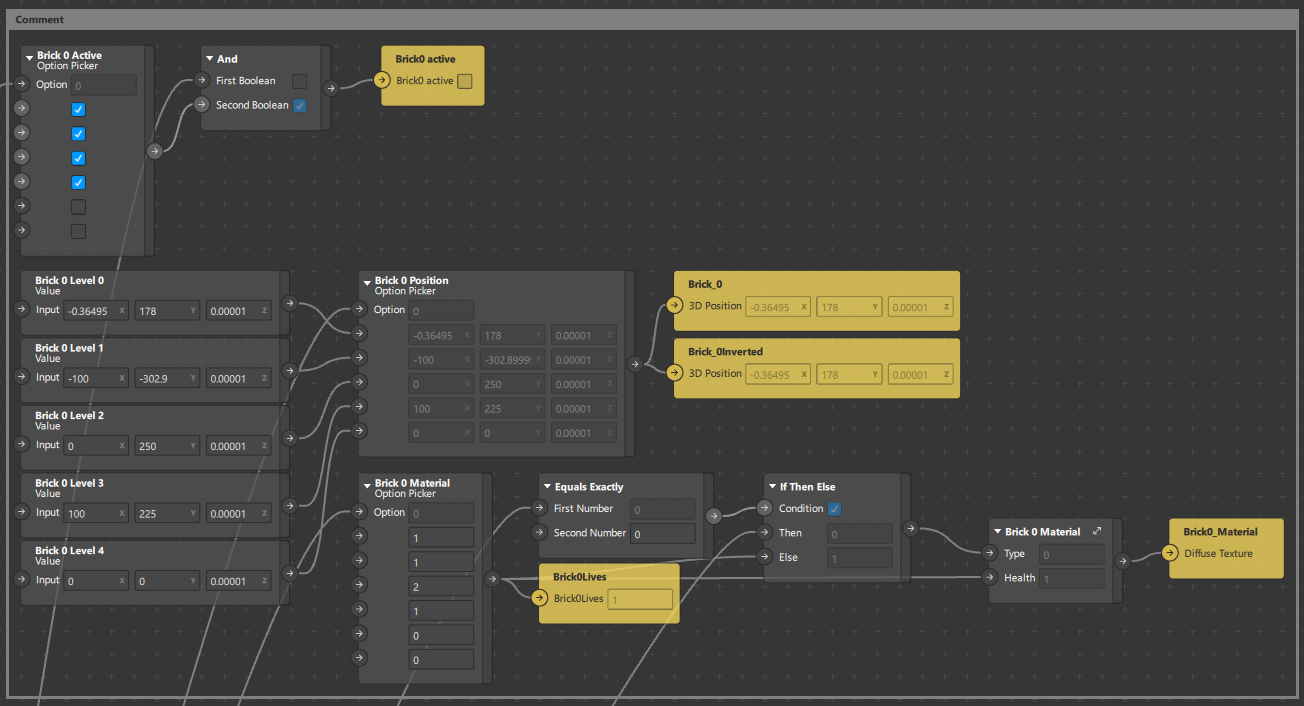
When the correct level is triggered, in this case (4), the boss spawns, then awaits the sequence to trigger the boss fight. There is a countdown of 2s, which then sends a pulse, then waits for another 0.1s to disable. This is due to the output being a boolean instead of a pulse.

Brick Spawner:



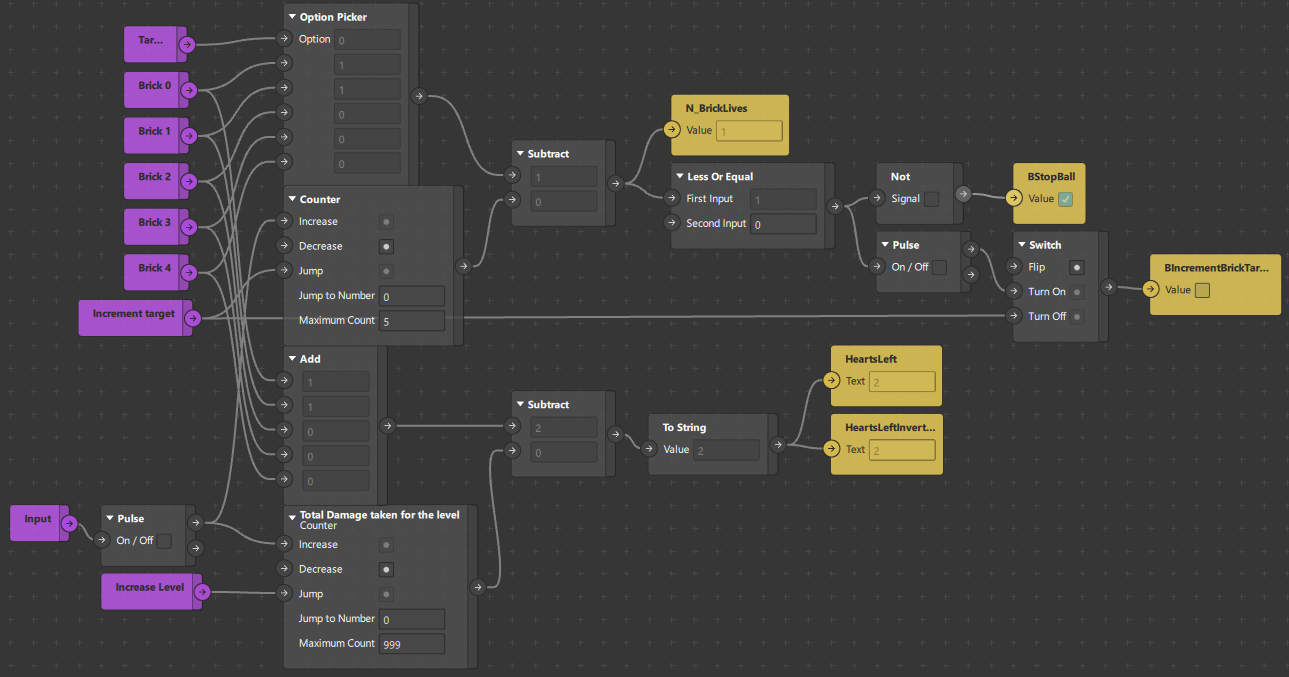
There are 5 bricks, 0-4, which are automatically looped back upon exceeding the value. The counter then sends a pulse to enable each of the bricks, such that brick placement and brick damage can continue to update the rest of the values. When all the bricks are destroyed, the level changes, which requires an exception to prevent skipping over the boss level(mentioned above).

Brick Placement:



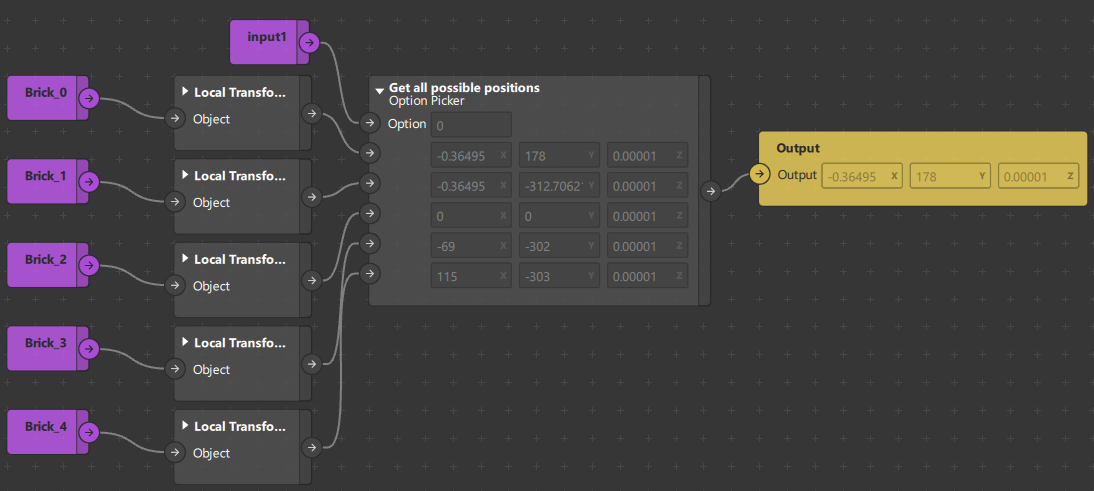
This is a single block within brick placement, the rest of the blocks follow the same format, with the exception of data. This is all hard coded for the position of the bricks, changing for each level. There is a material patch as well, to change when it has been hit, such as increasingly damaged versions of the “full health” brick.

Brick Damage:



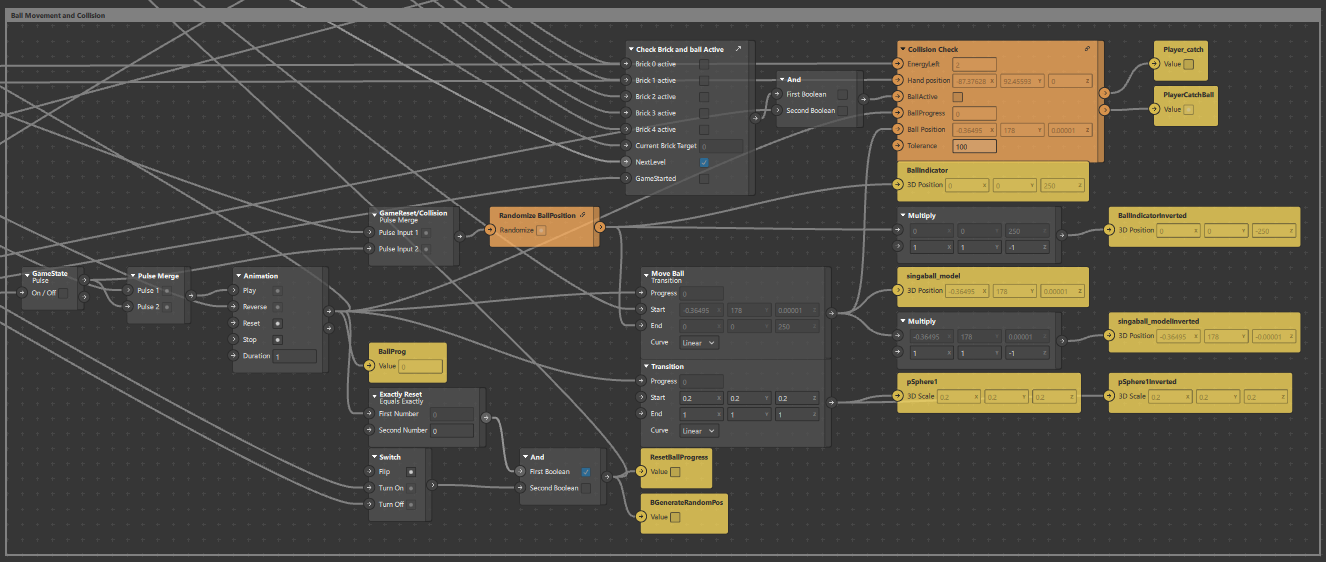
This patch tracks the damage taken for the level, which shows up as an UI element during gameplay. This also outputs a value for the current brick health, which helps to increment the brick target outside of this patch.

Decide on Ball Position:



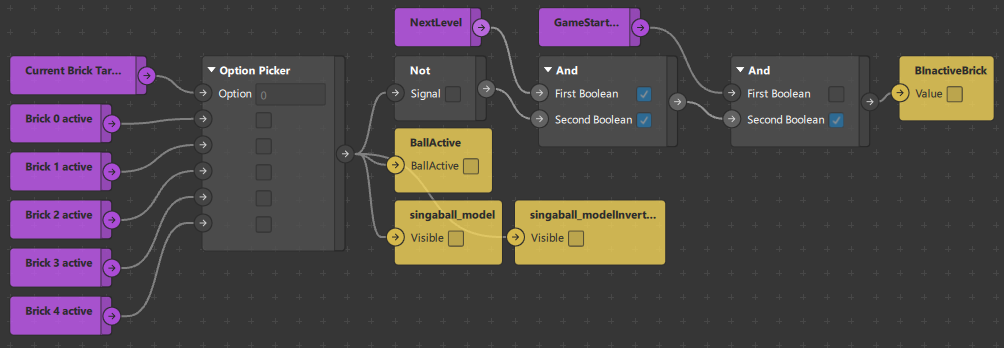
Since the location of the bricks are immediately acted upon, we can use the subsequent data to update the target location of the ball. The input value would then be the target index while it returns the coordinates of the bricks it targets.

Ball Movement and Collision Detection:



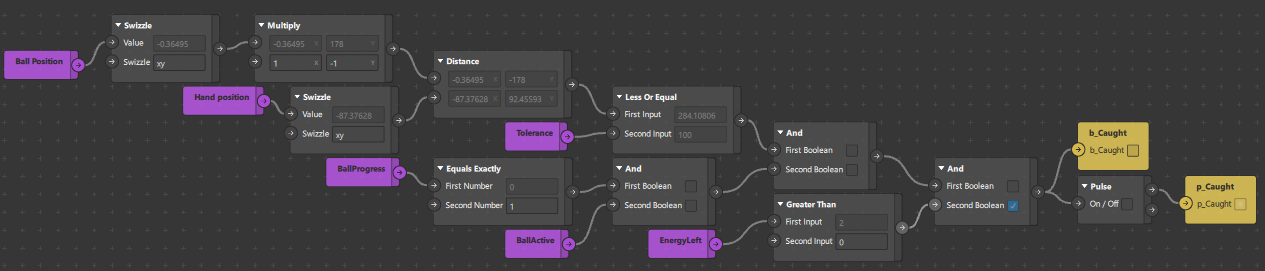
This is a medium sized patch sequence that handles the movement of the ball. Given a target, it randomizes the destination on screen, then awaits its spawn. After spawning, it travels to the destination with a duration of 1s. The return value is “ball progress”, which is used to trigger the indicator. It then awaits collision with the player, refer below for the patch asset used. When it catches, it triggers the subscribed event (caught ball), which subtracts the energy cost of the collision.

Check brick and ball active:



This patch group triggers the render of the ball and whether the brick is inactive, in case of breaking it completely.

Collision Check(Patch Asset):



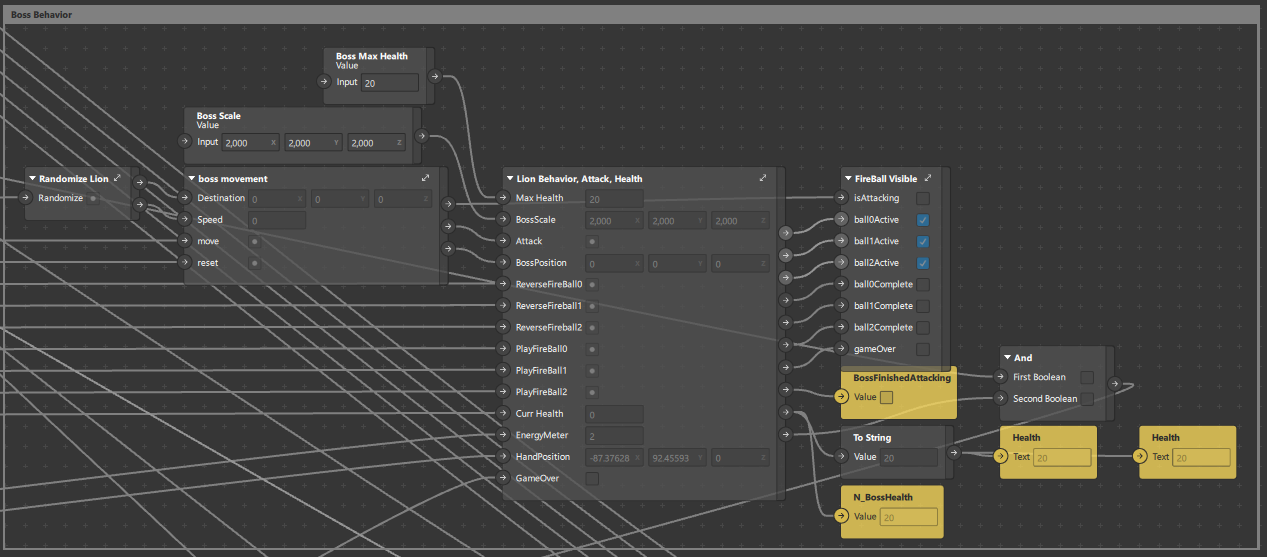
The code first flattens the XYZ value to the XY only, then inverses the y position, due to a conversion between screen origin and world space coordinates. Next, it converts the hand position, the cursor location. It converts it from XYZ to XY, with the same rationale as the ball.

The screen origin is in the top left of the screen, which means Y is downwards. The world space coordinates have the origin at the center of the world, thus the Y value is upwards. It is more reliable to convert the world space coordinates of the ball into the screen coordinates.

Comparing both in vector distances thus ignores depth, with a tolerance of 100. It also ensures that the ball has completely moved, with a progress of 1. Next it compares the energy required to “catch” the ball.

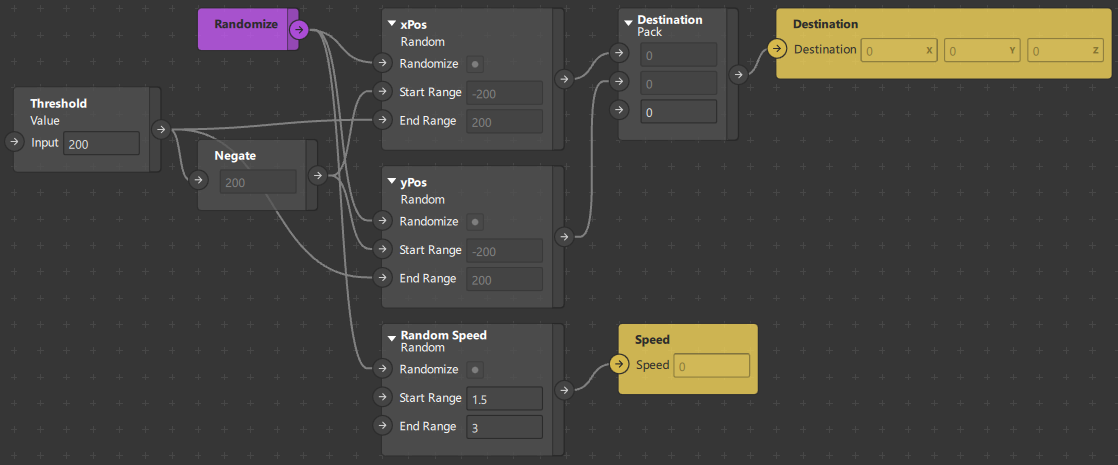
When all the conditions are met, it sends out both a boolean and a pulse to subsequent patches. This is easier done in a single location rather than putting only 1 output then converting it each time.

Boss Behavior:



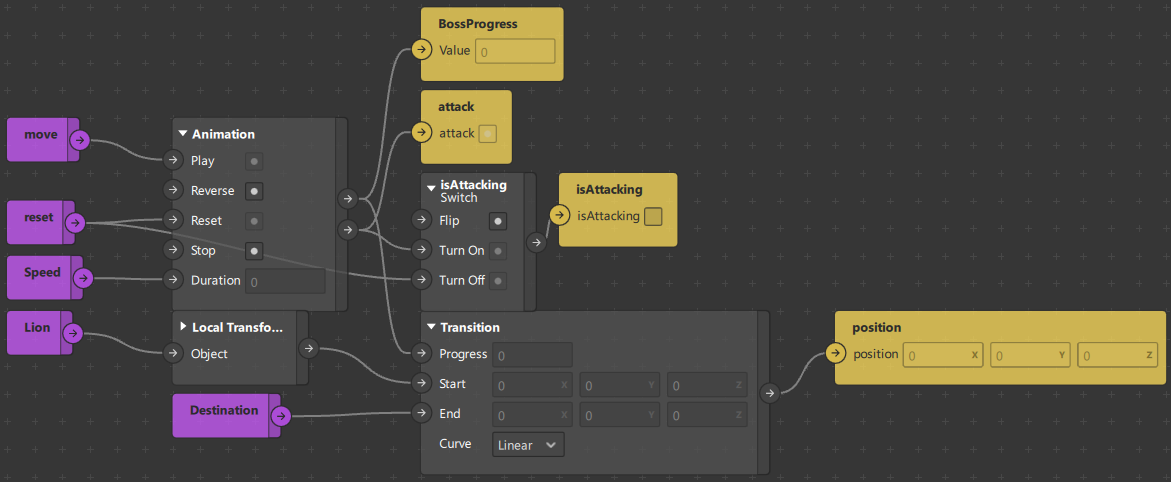
This group is simply a shell for organization purposes, refer to documentation below for the complete explanation.

Randomize lion:



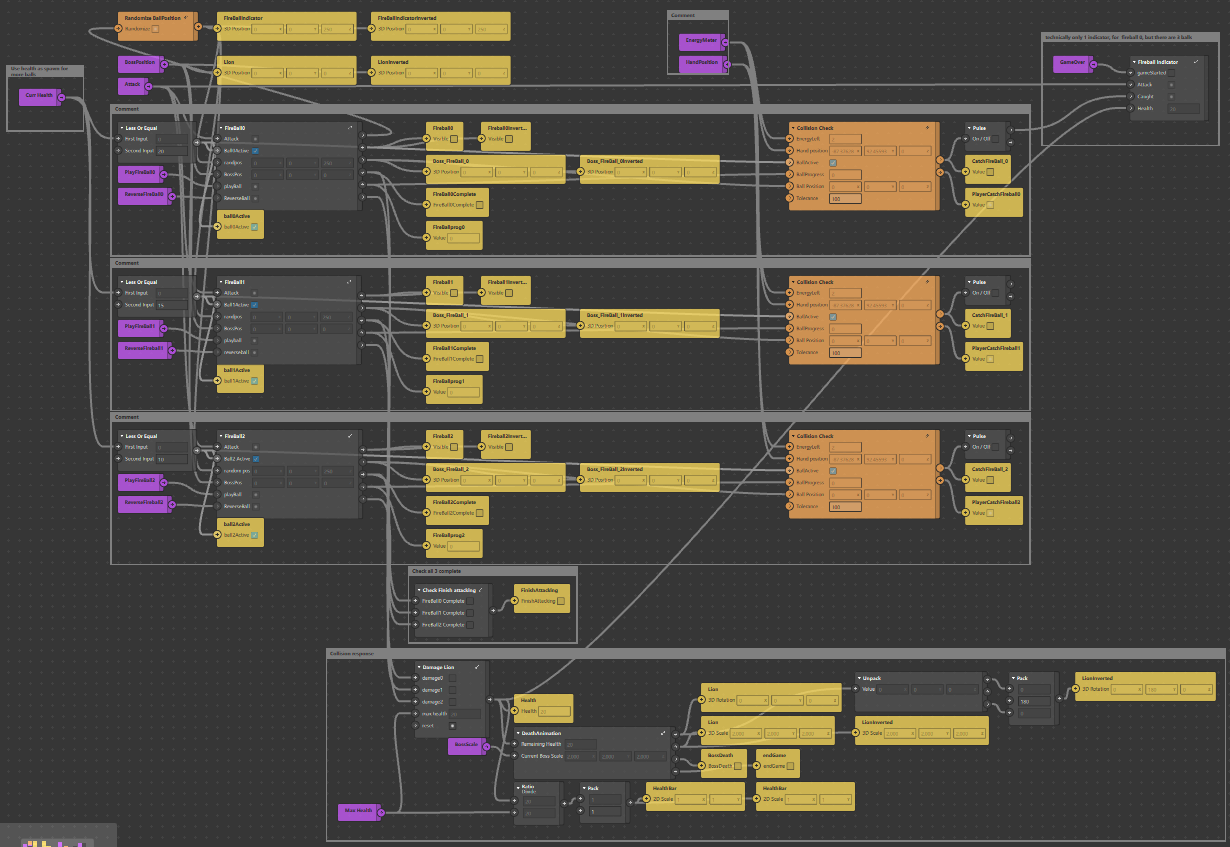
With a value of 200, it randomizes a random destination of -200 to 200 in XY for the boss. With a range of 1.5-3, it also randomizes the speed of the boss. Note that the speed is more aptly named duration of travel. The smaller the value, the shorter the duration of movement, thus faster.

Boss movement:



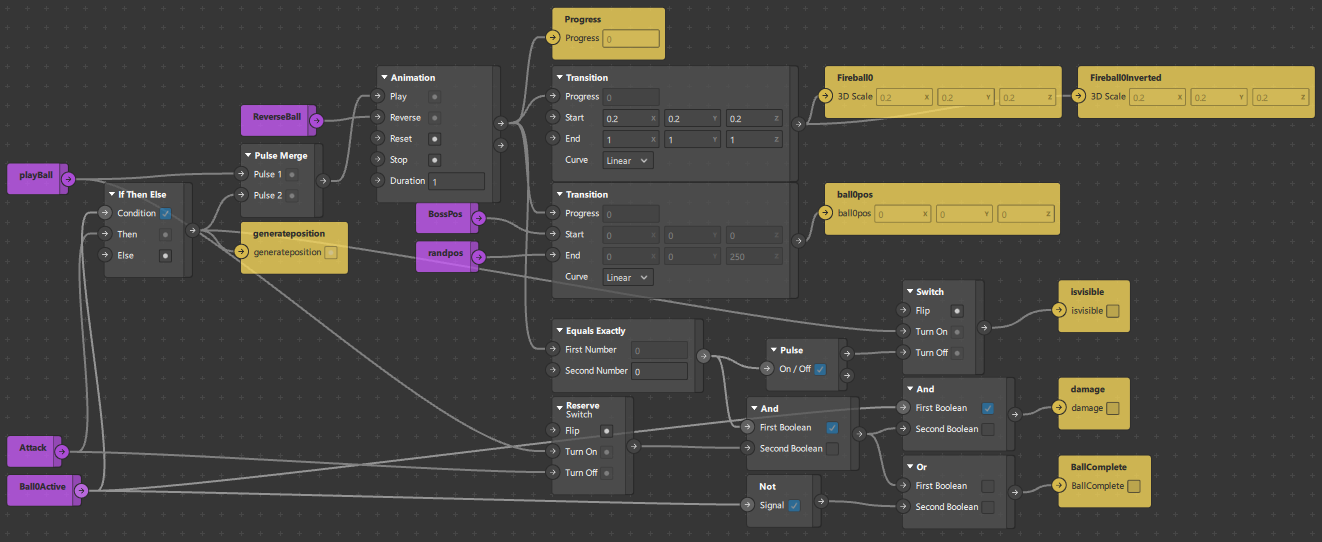
The boss follows a real time equivalent of turn based combat. It moves to a target location before firing a projectile. Then waits for the projectile to be hit back, then repeats by randomizing another location.

Lion behavior, attack, health:



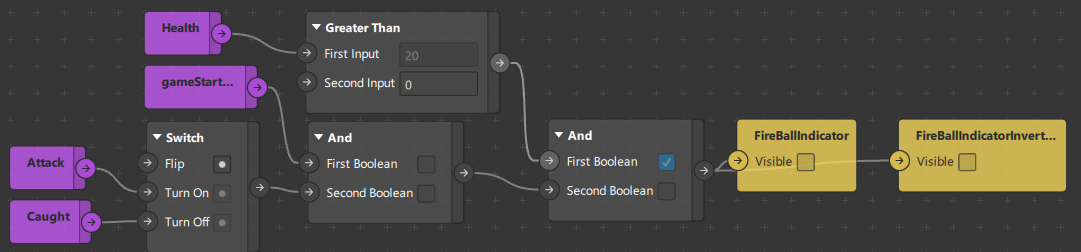
This is a large patch, but handles 3 main roles, spawning of fireballs in increasing amounts when health is low, handling collisions between the fireballs and boss, lastly, tracking the health of the boss, triggering death animation and state changing when needed.

Fireball Update:



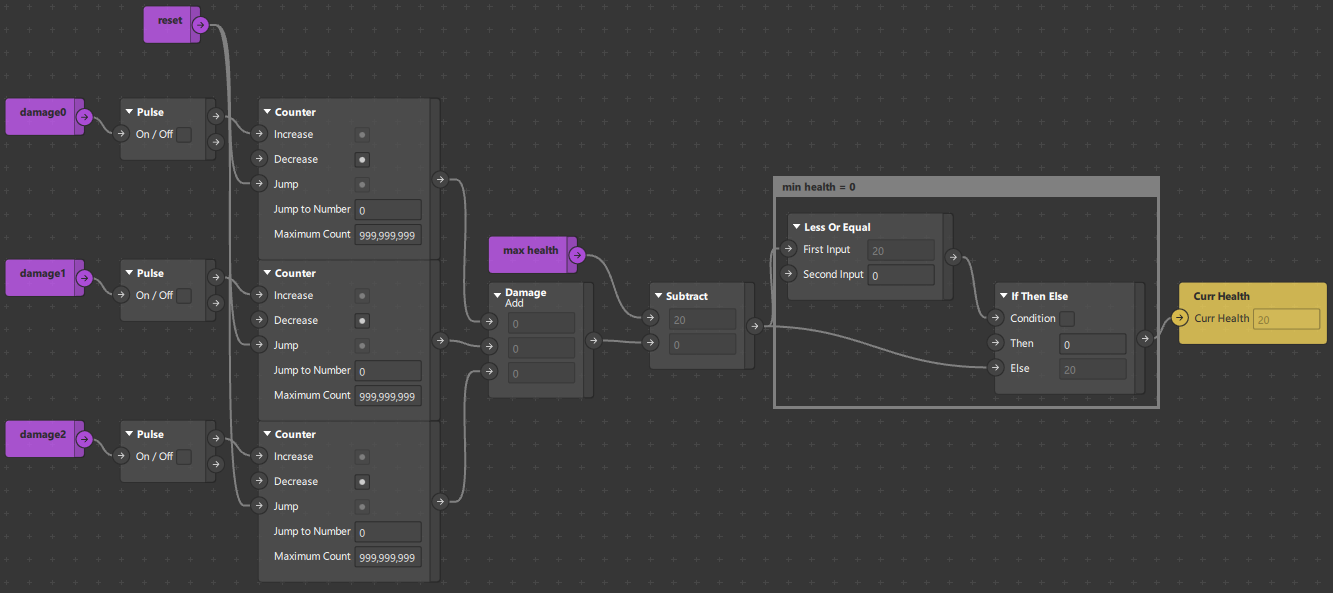
Update is not a patch asset, but the structure is the same as the original ball in regular levels. This is just condensed for all of them to be updated while the boss fight is in progress.

Fireball indicator:



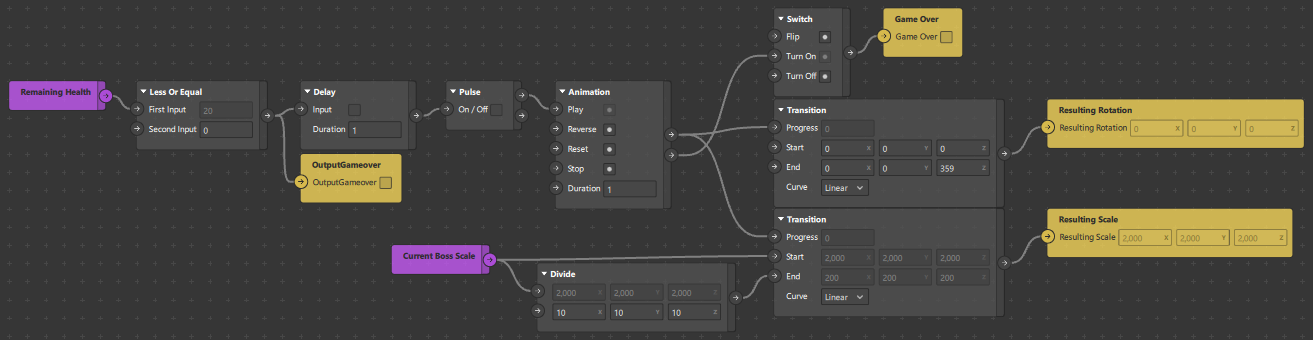
If the boss is alive, it checks for whether the fireball has been fired or caught. If the ball is fired, then display the indicator, this runs on a similar principle to the original indicator, but slightly more primitive.

Damage Lion:



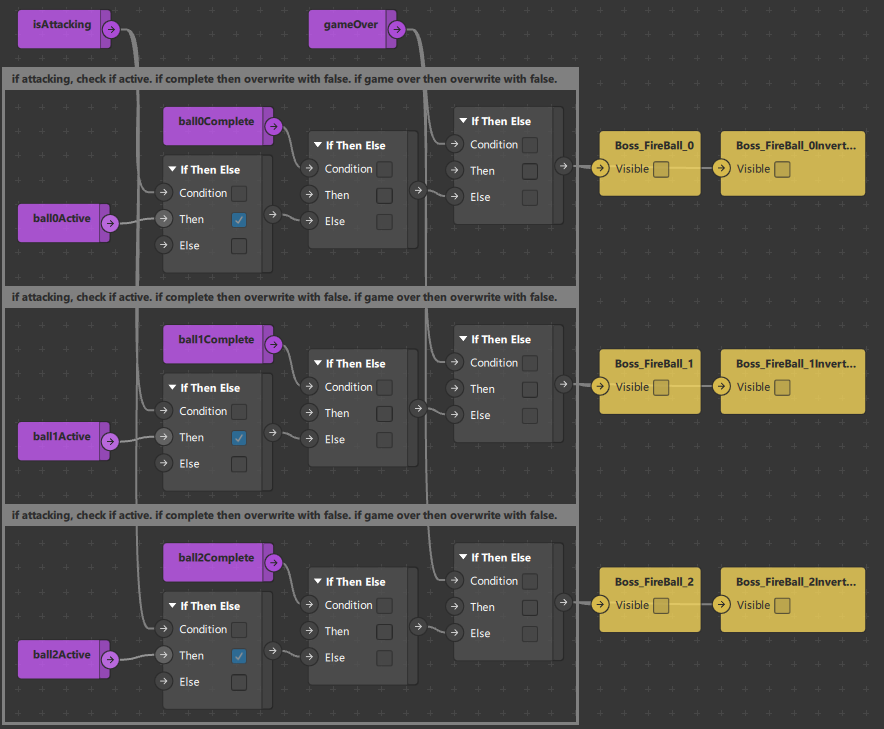
Since the fireballs themselves track the number of collisions, we add all the respective damages and subtract from the boss health pool. If the boss dies, there may still be damage incoming, dubbed Overkill, the health has a negative exception, forcing the output value to be 0.

Death Animation:



When the boss dies, it triggers a simplified animation of the boss spinning and shrinking. The duration of death is 1s. Before it triggers the results screen.

Fireball visible:



When the fireball is “attacking”, it is being rendered, but there are also additional checks, such as its active state, gameOver, whether the attack is complete. This means that the fireball is only rendered during the bossfight, attacking, as well as flying back.

### Additional Notes

Current implementations of scripts are used as global variables allowing for unofficial loops, thus suppressing loop errors and preventing recursive calling. The scripts use get and set functions to pull data from existing patches or set the values. This is an effective but primitive way to handle data, especially when there is no added logic or transformation of the data within the code. I have taken to rename more functions as well as document them for further use with a clearer understanding in the future.

# Development Details

## Current Tasks

Such empty

## Bugs & Issues

The current game UI is scaled to iPad Pro 11”. No other device will look or perform the same as this one, due to 3d space to 2d space conversion. Other engines will account for this by having a virtual camera, but unfortunately due to the face tracking capabilities of the Meta Spark AR, the engine uses a real camera instead.

Woup(Word Soup):

Current ball is set to be 250 units off the canvas, as well as a 100-200 pixel range of randomness within the safe touch zone. This is configurable but also try not to exceed this range. A larger range will increase the difficulty of the game, while accidentally leaving the confines of the screen. The 250 pixel depth is risky as the camera is NOT a set distance from the canvas, scaling to its pixel size and screen scale. This means that the game is set in 1 phone size, as other sizes can make the ball either behind the camera(bad), or too close to the canvas, thus making the ball move less, i.e. lethargic. Collision is also not in 3d space. Since the ball is closer to the camera, the ball is not only larger, but also sheared towards the edge of the screen, courtesy of lens curvature. This means that the collision flattens the Z coordinate but also uses the approximate “ray cast” vector from the camera to the ball to find the 2D Approx Coordinate of the ball. Then using a simple vector distance of the cursor to the ball, a tolerance of 100 Virtual Coordinate Units as a buffer.

Similar to step up right, the locations of the bricks are hardcoded to each level, meaning there is no replayability despite all of the interference from the logic states. This also means that we are unable to create patch assets for the locations due to no math or logic, just an array of values. This also means that each change of the level layout would require a large amount of adjustment. The movement of the ball and the boss is thus randomized, but repeated. Thus no matter how the movement of the ball is changed, the target remains static, thus getting stale over time.

# Test Plan

Instructions for setup & deployment. Use of video call & screen record for data collection. Only 1 iPad should screen record, preferably the host.

1. Set up 2 ipads with gmail within the messenger app, gmail should not be the same.
2. Add the ipad to a chat group, if not done prior.
3. Use 1 of the ipad as a host & video call the chat. This will call all participants within the group.
4. Enable screen share
   1. Within the messenger video call, there is a sharing button at the bottom of the screen.
   2. Recommend features include games & watch parties. Screen sharing is located on the rightmost button.
   3. Select screen share option & follow instructions given.
5. The host of the call will enable screen recording
   1. The screen record function can be found within settings, control center.
   2. Pull down the notification center & long hold the record button. This will expose the detail menu, click the microphone to enable audio recording, this will make the microphone red.
   3. Clicking through the record button will now record the screen & audio of all applications.
6. On both ipads, open the mobile player for meta sparks
   1. Click the hamburger menu located on the top left or top right of the app & select the most recent version of the target game.

Notes: the ipad can record a limited amount of videos, make sure to export the video in between play sessions. This is unlikely to present an issue if alternating ipads record the video. Name the video in concise terms as soon as possible as the name will be generated to the date-time of recording.

## Test Strategy

Session Count: 3(1 for each of the games)

Session Duration: 30 ~ 40 minutes each

Participants: 10 elderly with differing demographics, no particular specification

Alt: (Refer to Differing levels of technical proficiency) Provide explanation and tutorial phases before playing to reduce confusion, frustration and emotional fatigue towards gameplay. (not survey)

## Test Environment

Testing Facility: Elderly Activity Centers/Active Aging Centers

Technical Support: ARISE@NTU, Remote & Physical Presence needed

Equipment: iPad with preloaded Meta Spark AR games, Meta Spark Mobile Player, Messenger, Screen Record(Built-In)

## Test Cases

Collection of Data:

1. 1st measurement, before the start of the 1st session
   1. Demographic questionnaire
   2. Intergroup anxiety
   3. Intergroup attitude
   4. Loneliness (older adult only)
   5. Well-being (older adult only)
   6. Ageism
   7. Quality of interaction
2. 2nd measurement, after the 2nd session
   1. Identical as 1st measurement, except for demographic questionnaire
3. 3rd measurement, after the 3rd session
   1. Identical as 1st measurement, except for demographic questionnaire
   2. Post-test interview on feedback on exergames and intergenerational bonding

Expected Outcome and Response: hope to see a decrease in intergroup anxiety, a better intergroup attitude, lower loneliness (in older adults), higher wellbeing (in older adults), lower ageism, and positive feedback towards exergaming.

End Goal: To examine the effect of video-mediated communication and simultaneous gameplay on intergenerational communication and bonding.

# Asset Management

Assets are split between textures & materials. Most materials are generated by the software when a texture is used. Most scene entities will not accept textures, but require material instead. This means there are 2 layers to compression to reduce the size of the final product at the cost of slightly harder development. When deleting a texture, the attached material will turn into a checkerboard pattern. When deleting a material, there will be a separate prompt to delete the attached texture. If the prompt does not show up, then manually go into the root project to delete it. This is due to the software no longer referencing the asset, thus unable to clean it despite prior deletion.

Temporary assets are used to be able to change the sprite/texture during runtime, such as the textures of the bricks, stars and projectiles. These textures are assigned when needed while the rest of the game does not interact with the textures.

## Art Assets

Previous batch made them

# Appendices