welcome!

Jukebox Paradise

Technical Design Menu

Today's Special

Version 3

Design By:
Alberto Mastretta
Janel Jolly
Maria Lee
Frankie Fasola

Art By:
Alberto Mastretta
Janel Jolly

Written By:
Alberto Mastetta
Maria Lee
Frankie Fasola

VFS

Jukebox Paradise

Contents

Game Overview

High Concept

Jukebox Paradise is a 3D local multiplayer brawler for two to four players, where players collect milkshake ingredients on a spinning vinyl record in a 1950s diner. Their goal is to build as many milkshakes as they can before the songs end. In the process, players have basic melee combat abilities to fight for the milkshake ingredients! Since the gameplay area is constantly changing as the record spins, players also have to skilfully avoid changing environmental obstacles while simultaneously engaging in close combat with one another.

Game Changing Opportunities

Short Term: Fight nearby players by punching, dashing and throwing items.

dMid-Term: Collect ingredients and power-ups to gain advantages over the other players.

Long Term: Build the most milkshakes before the song is over.

Core Technology

JukeBox Paradise is developed using Unity 4.3 Engine. It is a PC game with Xbox 360 Controller support.

Platform

Unity is the chosen game engine due to the ease of use, the inspector, and various plugins available which will be utilized. Unity allows the team's artists, programmers, designers and level designers to make changes in game simply due to the user friendly interface and control options. Team members working on their respective areas inside of Unity can access the inspector to make real time changes without ever touching the code and causing another team member to get involved.

The plugins JukeBox Paradise will use are Audio Tool Kit, NGUI and Xbox Control Input Master. These plugins contribute to an increased pipeline and game development.

Alternate engines that were considered include Unreal Three Development Kit and Flash, but compared to Unity these engines to not offer the same advantages and are not optimal for JukeBox Paradise.

Feature Set

General Features

This section is a summary of the features for JukeBox Paradise.

Dynamic Levels

Time estimation for the following features: 30 Hours of Programming

Overview

There is one level in JukeBox Paradise. Through code obstacles will be created and spawned on the record during gameplay to provide a variety of challenges and situations for players.

Technical Implementation Plan

Step One: The songs in JukeBox Paradise will be broken down into three sections, intro/outro, verses and chorus. The level designer will plan obstacles and player interactions for each section of the song.

Step Two: A level Manager will be created to conduct the changes to the level during gameplay. This class will utilize the singleton design pattern.

Step three: A new generic level class will be created that uses the factory design pattern. This class will be used to create instances of the three level sections. The class will check which section of the level is called, it will then create a new one from the original class and return it.

Step Four: Three classes for each section of the level will be created. Inside of these classes the corresponding level functions will be created. The classes will spawn obstacles on the record based the tempo of the song and duration of the section.

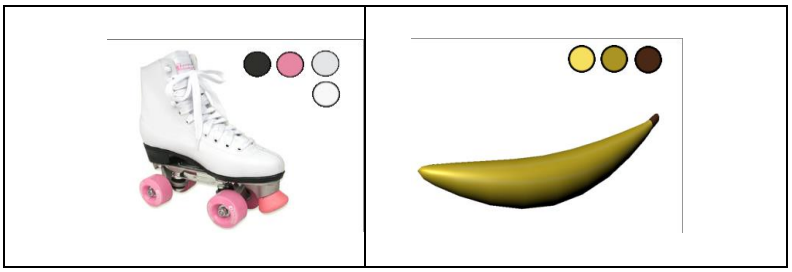
Objects will be spawned at designated spawn points on the record. A particle will play before the item is created to alert the player. After the particle plays the object will be created on the record and become part of the environment.

Step Five: Using the Level Manager and Level Generic class the level will be changed through code. After a section of the song has played, the level manager will switch to the next section and call that class to control the level.

Collectible Objects

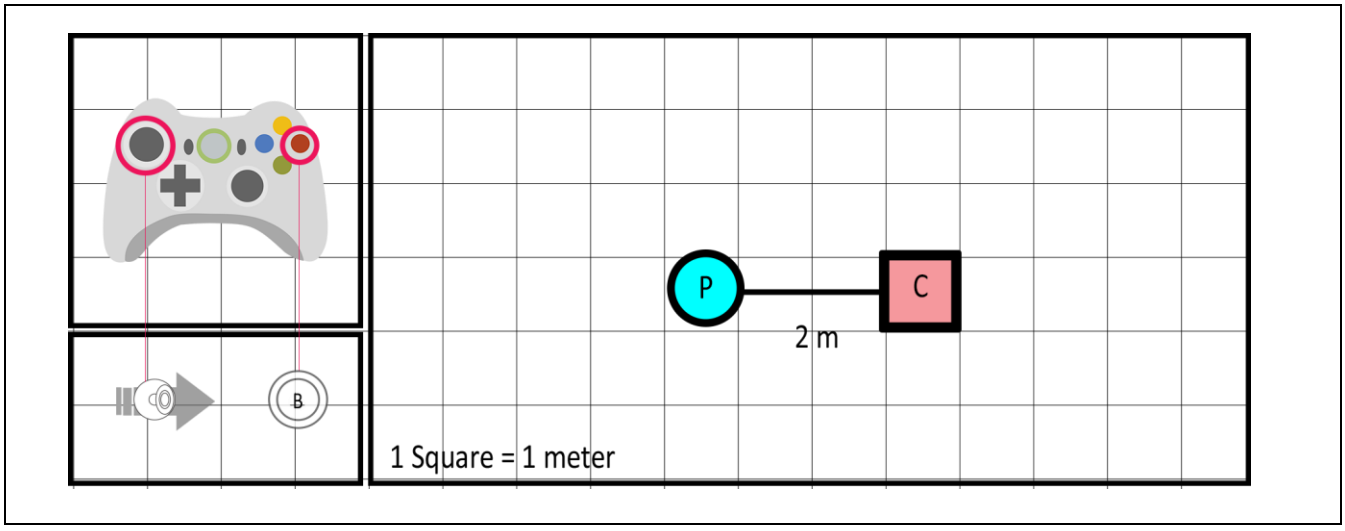
Overview

There are ingredients to pick up in the game. These objects are placed in the level for the player to pick up.



Collectible Objects: Collision

This section describes the collision of the collectible objects with the player.



Technical Implementation Plan

Step One: The ingredients are created by the artists in Maya and then are imported into Unity. They are scaled to x, relative to the base unit of the player.

Step Two: The texture is applied and then they are turned into prefabs.

Step Three: A rotation script is attached to the game objects so that they rotate over the Y-axis.

Step Four: A pickup script is attached to the game objects so that when the player collides with the object, its alpha becomes 0%, making it transparent.

Step Five: A script with the particle system is also attached to the game object. Its position is set to the game object's position, and when the player collects the object, the particle effects are enabled.

Step Six: Once the lifespan of the particles have been completed, the collectible object is destroyed.

Power-Ups

Overview

Time estimation for the following features: 30 Hours of Programming

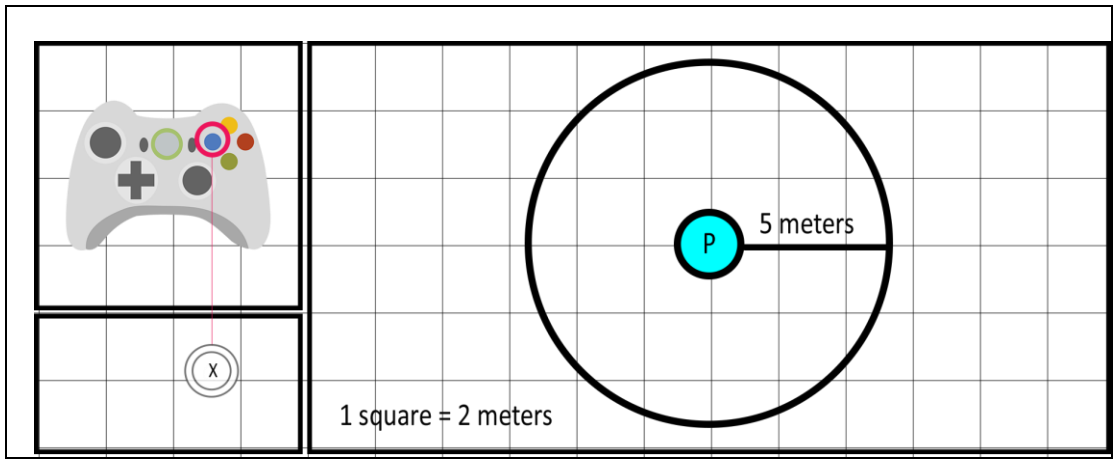
Power-ups are placed throughout the levels along the track. There are four power-ups in JukeBox Paradise. Power-ups grant player a unique ability for a set period of time.

All Power-ups are different functions or variables located inside the Power Up script. Inside of the Player class, we will add additional collision checks for the power-ups.

Power-Up: Milkshake

The Super Punch Power-Up allows players to punch their opponents in 360 degrees. Once a player picks up the Super Punch Power-Up with the 'B' button, a 0.1 second animation of the player's hand changing into a large (5x times its regular size) red boxing glove is shown.

For three seconds, the player can move the avatar by using the left analog stick and whenever the 'X' button is pressed, the avatar punches the ground and attacks in 360 degrees. The animation for this is 0.2 seconds



Technical Implementation Plan

Step One: Changing the Punch Force and Radius

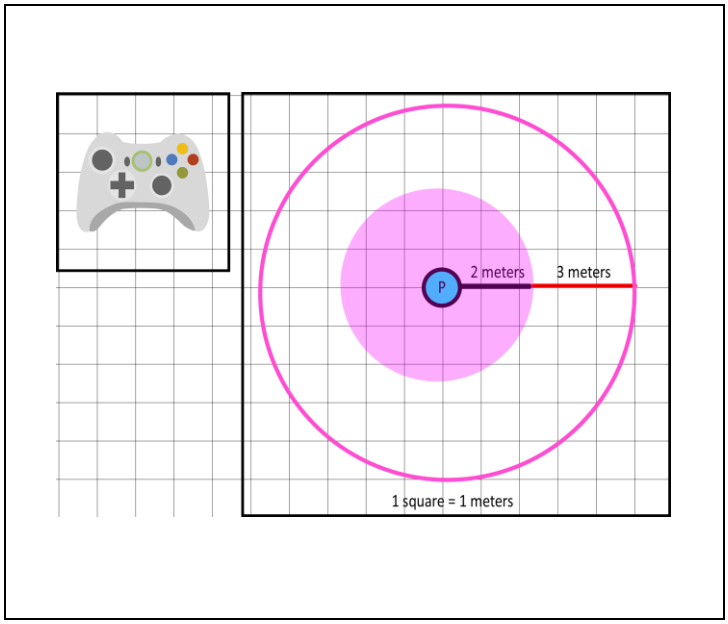
When the player collides with the power milkshake, a Boolean is set to true. Inside of the player punch function, an if statement checks if the Boolean is true. When this Boolean is true, the force and distance of the punch are changed. These variables are then used for the punch function. The punch power and radius are increased.

Step Two: Reverting the Force and Radius

When the Boolean becomes true a timer is set to three seconds. The timer counts down and when the timer reaches zero, the Boolean is set back to false.

Power-Up: Shield

The Shield power-up puts a shield around the player for a brief period of time. If the player is attacked while they have the shield power-up, the attack is directed back at the attacker. The player inside the shield cannot attack while inside the shield.



Technical Implementation Plan

Step One: Create the InShield Function

The InShield function will check to see if it is colliding with anything on the player layer. If the shield collides with a player, it will push the player back using Vector3.back. A static Boolean called shield is also created.

Step Two: Calling the InShield Function

The player class has to be altered to work with the Shield power up. First, inside of the Update function, we create a new if statement that checks if the shield Boolean is true. This prevents the player from attacking. While the Boolean is true the player class will call PowerUp.InShield. A timer is created inside the function that will count down until the Boolean becomes false, and the player loses the Shield power-up.

Power-Up: Speed Runner

Summary of Speedy Runner	
Power-Up: Ability	Increases a player's running speed
Duration	5 seconds
Run Speed	15 meters per second

Technical Implementation Plan

Step One: Creating the Running Object

The Running object will be a game object. The object will be modeled in Maya and imported into the Models folder. Inside the OnCollisionEnter function inside the Player class, a new collision check will be created for the Running power-up.

Step Two: Running Collision

Inside the Controller script a new static Boolean, run power up will be created. When the player collides with the Running object, the PowerUp.Running function will be called. The function begins a timer for the power up and turns the Boolean run power up to true.

This will increase the movement of the player for a brief time. In the Controller script a new if statement will be made to check if the Boolean is true and change the movement speed

Game Modes

Overview

This section is a summary of the two game modes for *Jukebox Paradise.*

Milkshake Battle: Free-for-All

Free-for-all has all 4 players compete trying to build the most milkshakes before the song ends.

Milkshake Battle: Team-based

Team battle has players work together in two person teams to build the most milkshakes before the song ends.

Technical Implementation Plan for Game Modes

A Game Modes class will be created and put on the milkshake glasses in the scene. The class will count down a timer that is the duration of the song. When the song is over a function will be called that plays the next song in the jukebox and communicates to the level manager, telling it to switch leverls.

Step One: Create a Goal Script

A new script will be made called Goal. This script will be placed on the empty game objects inside of the milkshakes. This class will track which ingredients have entered the milkshake glass and communicate them to the game modes script.

Step Two: Gameover Function

When a milkshake glass receives all the ingredients it will tell the game mode script which player or team won the game. A bool will become true and gameover will be called, triggering the ending sequence of the game. The players will lose control of their players and particles will play. Using NGUI text will be displayed on the screen telling which player or team won. The winning team will play an animation of the cheering, the losing team will play an animation to show they are upset they lost.

Free-For-All:

Step One: Load a scene with four milkshake glasses.

A scene will be created by the level designers and contain four milkshake glasses, one for each character. The glasses are modeled in Maya, imported and scaled to one. Inside of the glasses an empty game object is placed and turned into a trigger.

Team:

Step One: Load a scene with two milkshake glasses.

A scene will be created by the level designers and contain two milkshake glasses, one for each character. The glasses are modeled in Maya, imported and scaled to one. Inside of the glasses an empty game object is placed and turned into a trigger.

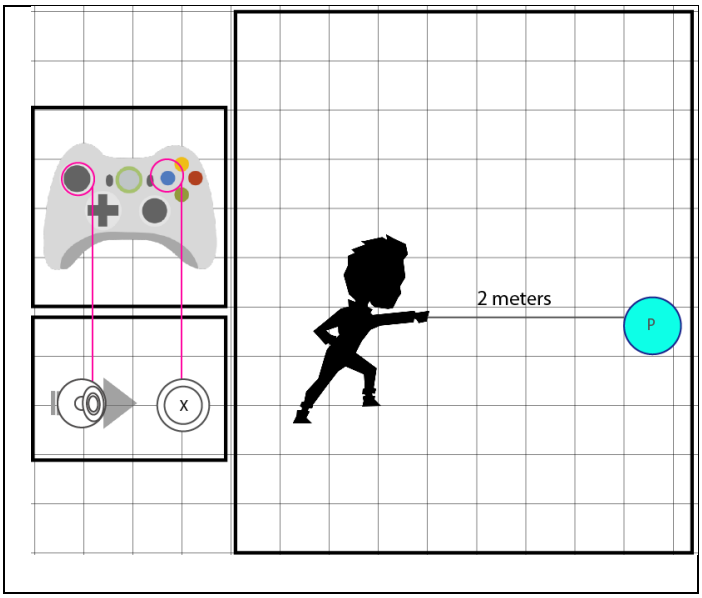
PUNCH

OVERVIEW

Time estimation for following features: 35 Hours of Programming

Jukebox Paradise features simple melee combat. Players can punch each other. There are no health bars or percentage points for players. Melee combat is used to knock the other players off the record and into the death zones around the level.

There are no combos, or special combat moves, but melee combat is very difficult to get right. Most of the time is going to be spent on tweaking to make sure the combat feels fluid and balancing the kickback from attacks.



WEAK ATTACK

Button Press (seconds)	**0 – 0.3 seconds**
Force	15 N
Knockback	1 meter

If the 'X' button was pressed for 0 to 0.3 seconds, the player automatically initiates a Weak Attack. A force of 15 N is applied and the knockback distance for the opponent is 1 meter.

MEDIUM ATTACK

Button Press (seconds)	**0.31 – 0.7 seconds**

Force	23 N
Knockback	3 meters

Table 1. Medium Attack Breakdown

If the 'X' button was pressed for 0.31 to 0.7 seconds, the player automatically initiates a Medium Attack. A force of 23 N is applied and the knockback distance for the opponent is 3 meters.

Strong Attack

Button Press (seconds)	**0.71 – 0.99 seconds**
Force	30 N
Knockback	5 meters

Table 2. Strong Attack Breakdown

If the 'X' button was pressed for 0.71 to 0.99 seconds, the player automatically initiates a Strong Attack. A force of 30 N is applied and the knockback distance for the opponent is 5 meters. If the 'X' button was pressed for more than 0.99 seconds, then the player completes a knockout move, disabling its opponent for one second on the Step.

Technical Implementation Plan

Step One: Creating Variables for Attacking

The players are given one attack that can be charged up to increase power. We start by creating Floats for the distance and force. A Layer Mask is created set to the player layer.

Step Two: Pressing the X Button

An Attack function is inside of the players class. The Attack function uses XCI Get Button Down to check to see if the player presses the X button and it checks which player pressed the button. Player numbers are assigned in the inspector through an int. When the player presses the X button, a timer starts counting the duration of the button press.

XCI Get Button Up is called next and calls another function called Check Attack.

Step Three: Attacking Other Players

An array of colliders is created. Using an overlap sphere cast and a for each loop we will check to see if anything is within the attack distance. Next the function checks to if any of the colliders are in front of the player. The first collider in front of the player is the attacked.

Step 4: Applying Kickback

The function looks at the timer from XCI Get Button Down. The length of the press determines the power of the attack. There are three strengths of an attack, weak, medium and strong. A weak attack is a button press under .3 seconds, a medium attack is a button press between .31 and .7 seconds and a strong attack is anything over .71 seconds.

Using Vector3s the direction of the attack is calculated and the player is pushed in the direction using add force and multiplying the attack strength by the direction of the attack.

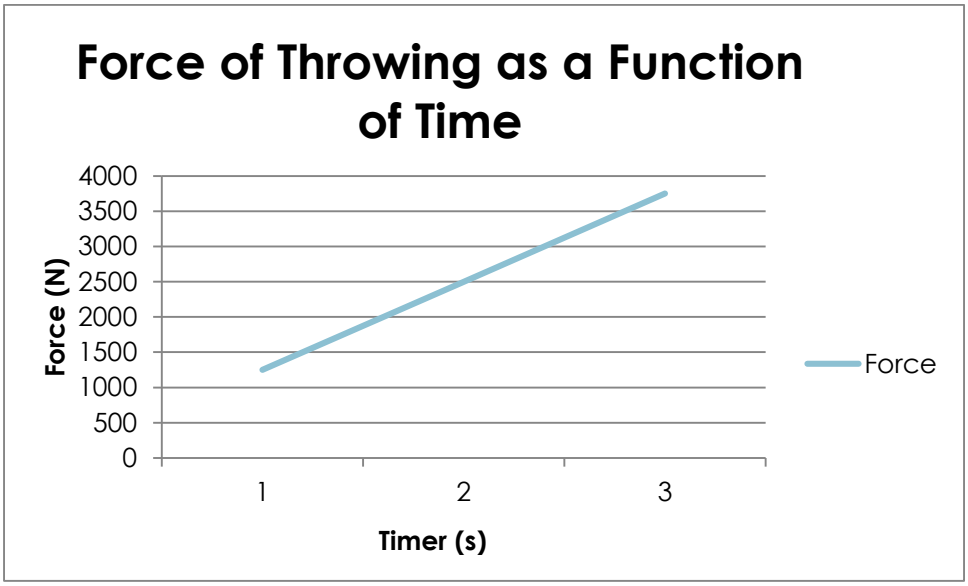
Throwing Objects

Overview

Time estimation for following features: 20 Hours of Programming

When a player has an ingredient, the player presses ‘B’ again to throw it. The direction of the object being thrown is determined by the direction that the left analog stick is pressed. If the left analog stick is not being pressed in any direction and the player presses the ‘B’ button, the object is thrown to the player’s position.

The force and the distance of a throw can vary depending on how long the ‘B’ button is pressed for. The impulse of throwing is calculated by multiplying the force – 1250 Newtons – and the number of seconds that the ‘B’ button has been pressed for. The maximum force applied to throwing is when the player holds the ‘B’ button for three seconds.



If the ingredients are thrown at other players, they bounce off of them and they are moved back according to the power of the throw. The farthest that they move back is 2 meters, which occurs when the player presses the ‘B’ button for three seconds.

Technical Implementation Plan

Step 1: Pick Up Class

In the scene an empty game object is created and called pick ref. A class is created called Pick Up. In the start function a game object, pick Object, will find the pick ref from the scene and a layer mask will be bit shifted to the object layer.

Step Two: Searching for Objects

A Vector3 and Quaternion are created and set to the transforms rotation and position in the update loop. Using the XCI Get Button Down function Unity will check if the B button is pressed and which player pressed the button. Player numbers are assigned in the inspector through an int.

An Array of colliders is created and is populated with an overlap sphere cast. Only game on the object layer are checked. An int i is created and set to zero. If i is equal to zero the game object pick object will become equal to the first collider.

Step Three: Picking Up Objects

The collider is changed to ignore gravity, become kinematic, and a trigger. The collider is then set to be a child of the object picking it up. The position and rotation of the collider is set to the Vector3 and Quaternion created earlier. The rotation script on the collider is turned off and a bool becomes true saying the player is picking up an object. Finally i is incremented to allow only one object to be picked up.

Step Four: Throwing Objects

When the player releases the B button the picking bool becomes false and a separate bool becomes false which allows the player to pick up objects. The player now has the object and can move with it.

An if statement checks if the player has the object and press the B button using XCI Get Button function. A timer starts counting and a bool becomes true that says the player is throwing.

Using the XCI Get Button Up function and checking to see if the player is throwing, the player can throw the object. First the throw strength is set to the min or max throw depending on the duration of the button press. Button presses under one second will be set to one and button presses over two seconds will be set to two seconds.

A float is created and is equal to the length of the throw press multiplied by the throw power. The throwing bool then becomes false. The object picked up was stored inside of pick object so it can be referenced here. The object is set to use gravity, not be kinematic, not a trigger and its parent becomes null. Using add force and the inputs from the left analog stick, the direction and force of the throw is calculated. Pick object is set back to pick reference and the player can pick up objects again.

Step Five: Dropping Objects

Inside of the player class a function is created called Hit Counter. Every time a player is hit and they are carrying an object the counter goes up. When the counter reaches three a bool is set to true which makes the object the player is carrying use gravity, no longer kinematic, no longer a trigger and lose its parent. The throwing bool is set to false and the player can pick up objects again.

If the player is hit with an attack that has been charged for over 1 second, they will drop the object regardless of the hit counter.

Step Six: Objects Inside of the Milkshake

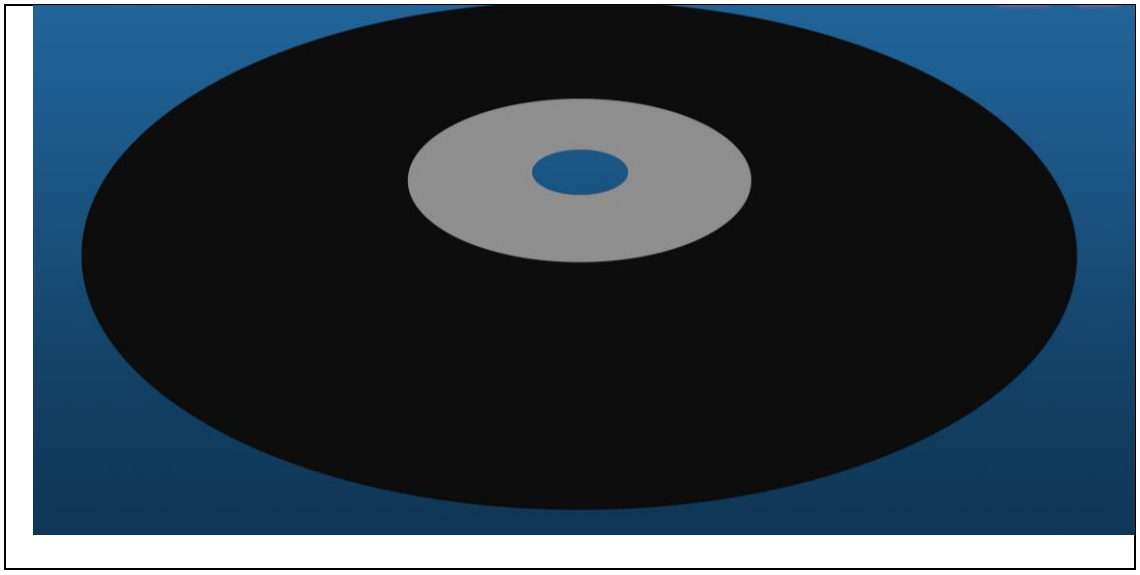
When an object lands inside of the milkshake it is moved off screen. The object is set to not be rendered or use gravity and is set to kinematic. A random number is chosen between one and ten. The number begins to count down, when it reaches zero the object is moved to a spawn point. The object then becomes rendered, set to obey gravity and no longer kinematic.

Camera System

Overview

Time estimation for following features: 15 Hours of Programming
The camera does not move, but features a shaking feature when players land critical hits when they punch or tackle.

Jukebox Paradise uses a 3/4 top down, static camera. This camera was chosen to give players an area to fight each other on, and allow players to see approaching obstacles. The camera has shaking effects when players land critical hits.



Technical Implementation Plan

Step One: Moving the Camera to Position

First the camera will be moved to the desired position in the editor. After this the camera will not be moved again.

Step Two: Creating Camera Controller

The camera controller will control when the camera should shake. The camera will shake when one of the players lands a critical hit, or when a player touches an exploding obstacle in the environment. The first part of the Camera Controller is to create a Transform for the camera. In the Awake function the transform is checked to see if it is null, if it is null, the Transform will get the Transform of the object it is attached too.

Next creating a Vector3 to store the original position of the camera is needed. In the OnEnabled function the Vector3 is set to the local position of the Transform that was created first. The final variables created are three Floats used to determine the length of the shake, the force of the shake and how the shaking should decrease.

Step Three: Shaking the Camera

To have the camera shake, inside the update function, we call the CameraShake function. Inside the CameraShake function we check if the Float for length is above zero. If the Float is above zero, take the Transform we made and set it equal to the Vector3 we made for original position, added to Random.insideUnitSphere, to get a random value within a radius of 1, multiplied by the shake force. The Float for shake length is then set to be equal to Time.DeltaTime multiplied by the Float for decreasing the shake.

When the Float for shaking reaches zero, the Transform is set back to Vector3, which contains its original position.

Step Four: Attaching the Script and Causing the Camera to Shake

The Camera Controller script will be attached to the main camera in the scene. Inside of the class for Players, a CameraController Script will be created. When players land a critical hit, the CameraShake function will be called to make the camera shake.

Game World

Overview

Time estimation for following features: 35 Hours for Programming

World Layout

Scale

All of the game objects in *Jukebox Paradise* are measured in meters. Any art asset that is imported from Maya will be scaled within the Import Settings in Unity to ensure that manual scaling does not occur on Unity's Scene editor.

Object Type	Width	Height
All playable characters	0.6 meters	2 meters
Milkshake Ingredients/Power-Ups	0.4 meters	0.7 meters
Milkshake Glass	2 meters	4 meters
Record	30 meters	3 meters
Record Needle	15 meters	3 meters
Lint Ball	2 meters to 6 meters	2 meters to 6 meters

Coins	5 meters	5 meters
Oil Residue Areas	5 meters	10 meters
Scratch Holes	4 meters	8 meters

Weather

The weather is always sunny in Groovsville; there is never a rainy day! Although the game takes place indoors in the diner, it is a warm spring day around 2:30 PM after school. Why else would milkshakes be so popular in Groovsville?

Levels

The record spins clockwise, according to the tempo of the song that is playing in the background for each level. It rotates on the y-axis from its pivotal point which is located at the center of the record.

All of the three background music for the levels feature a popular dance music genre from the 1950s – Doo Wop, Rockabilly and 'Dirty' Rock 'n' Roll. The speed of the record turning is determined by the BPM (Beats per Minute) of each song. The background music for Level 1 is Doo Wop because it has a 12 8 Compound Time Signature. As this type of time signature is used for slower songs, players have a better opportunity to learn character movement and their abilities in the first level. The rotational speed of the record increases as Level 2 and Level 3 transition into songs with a common time signature, used for songs with faster tempo. The variation between Level 2 and Level 3 comes from the BPM increase in Level 3 at 180 BPM.

Unless the background music has speed changes, the rotational speed of the vinyl record does not change. Players cannot change the rotational speed of the vinyl record.

Level	Music Genre	Time Signature/BPM
Level 1	Doo Wop	12 8 Compound Time Signature/83 BPM
Level 2	Rockabilly	Common Time Signature/120 BPM
Level 3	Dirty Rock 'n' Roll	Common Time Signature/180 BPM

Technical Implementation Plan

Step One: Import the Assets

The artist will model the environments inside of Maya and import them to Unity. The objects will then be set to a scale of one.

Step Two: Create Prefabs

Levels will be designed in sections to correspond with the song. A song has an intro/outro, verses and a chorus. The levels will have sections for each part of the song. These will be made into prefabs so they can be used as modular pieces.

Step Three: Use Code to Spawn Prefabs

The levels will be created dynamically in code. The prefabs will be instantiated during run time, and destroyed when they leave the camera.

Step Four: Rotating the Record

The record will be set to rotate on the Y axis using Unity's transform rotate. Using the plugin Audio ToolKit we will find the beats per minute. When the beats per minute are increased or decreased the record will speed up or slow down to the tempo of the song.

Movement

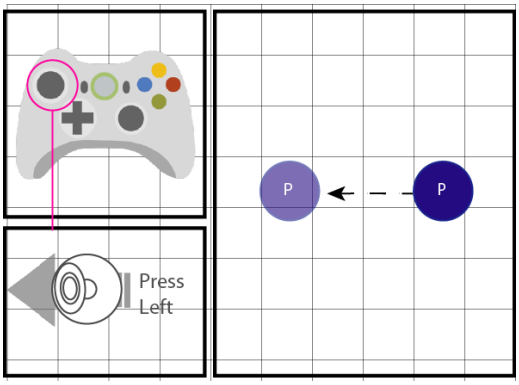
Overview

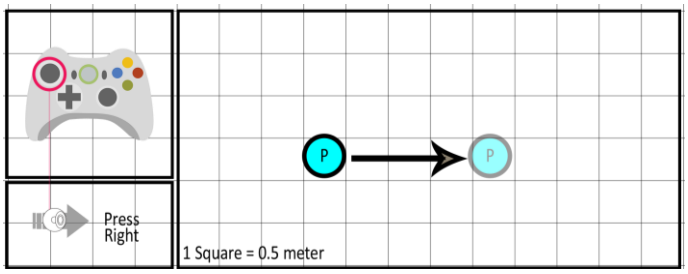
Time estimation for following features: 10 Hours for Programming

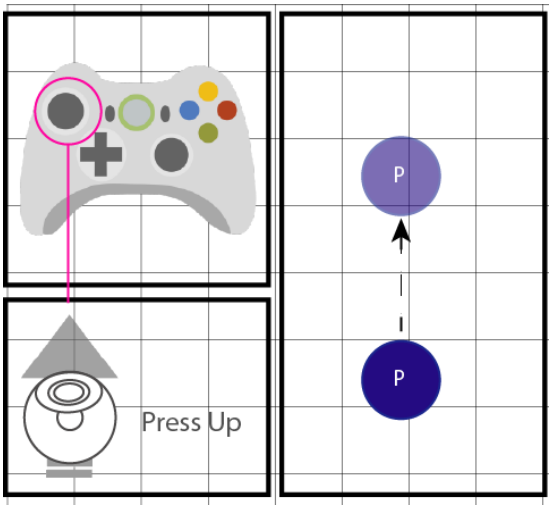
The playable characters in *Jukebox Paradise* move along the x- and z-axes by using the left analog stick of the XBOX controller. They do not move if the left analog stick is not pressed.

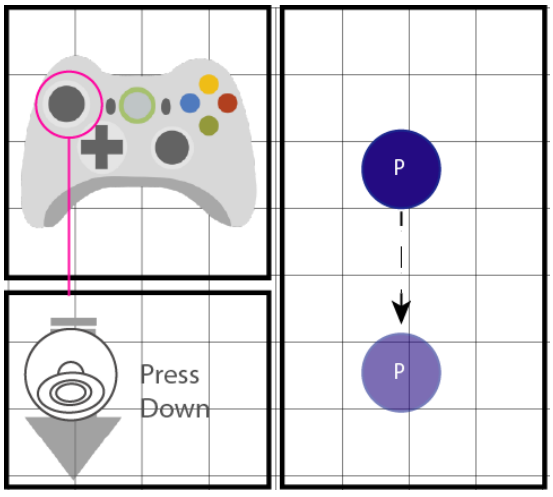
All of the players on the arena have gravity applied to them with rigidbodies. The frictional gravity force is 10.

When the left analog stick is pressed once, the avatar moves at a speed of 10 and moves 1.5 meters towards the direction that it is facing along the 8-axes. The player direction follows whichever direction the left analog stick is pressed. For example, if the left analog stick is pressed once in the upwards direction, the avatar moves up on the screen by 1.5 meters.





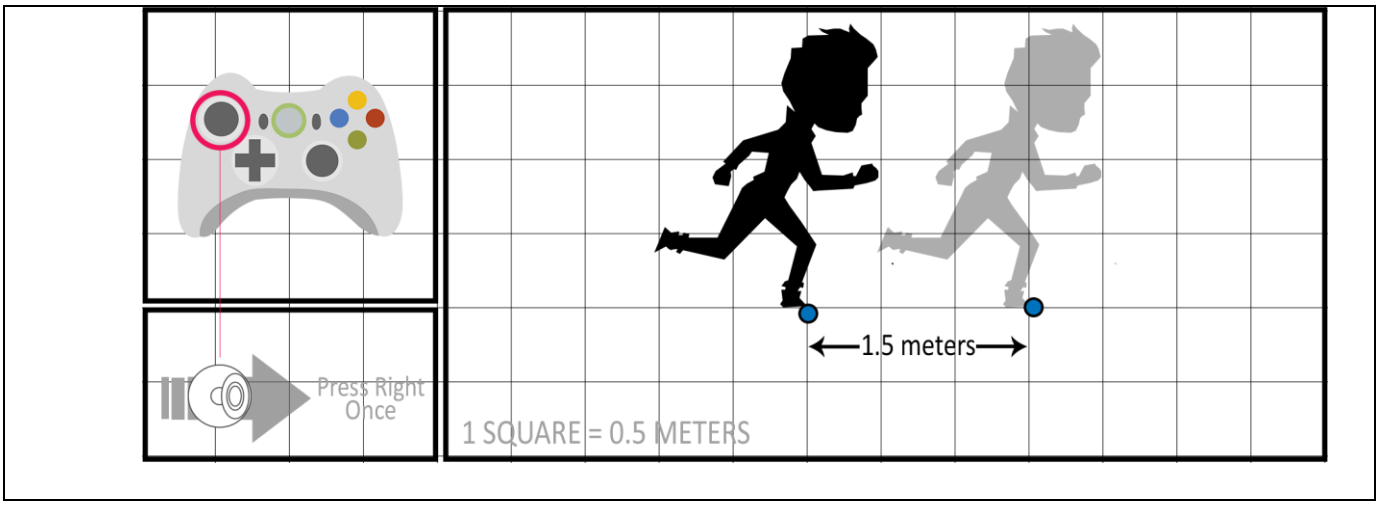


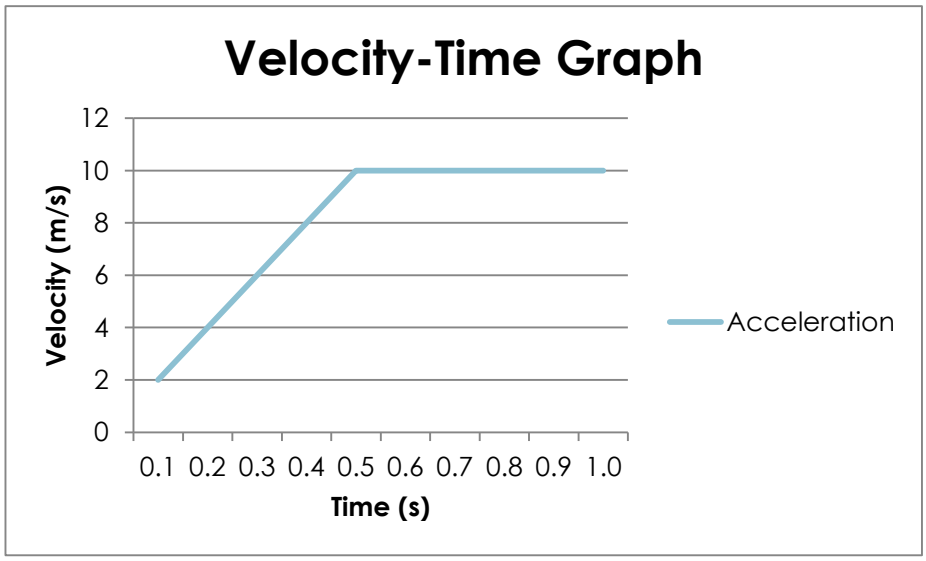


RUNNING

Time estimation for following features: 2.5 Hours for Programming

When the left analog stick is pressed once, the avatar moves at a speed of 10 and moves 1.5 meters towards the direction that it is facing along the 8-axes. The player direction follows whichever direction the left analog stick is pressed. For example, if the left analog stick is pressed once in the upwards direction, the avatar moves up on the screen by 1.5 meters.



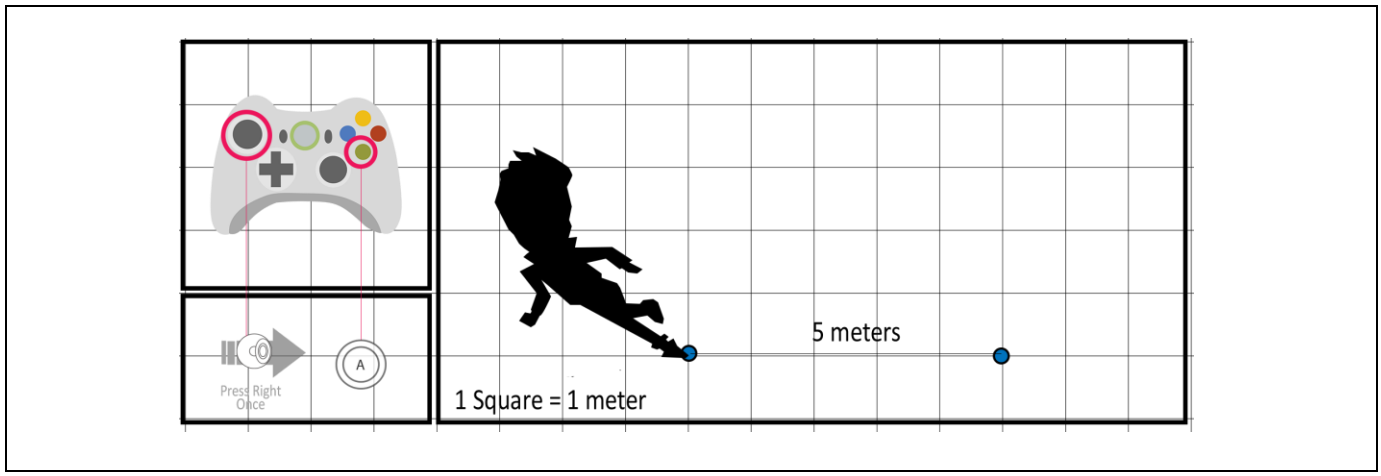


If the left analog stick is pressed fully for five seconds, the avatar reaches the maximum velocity of 10 meters per second. Once the maximum velocity has been reached, the acceleration remains constant and positive.

SLIDING

Time estimation for following features: 5 Hours for Programming

When the player moves the left analog stick and presses ‘A’, the avatar performs a slide. Sliding moves the player forward by 5 meters in the direction that they are facing. When sliding, players cannot change their direction. They only move forward five units in the direction that they are facing.



The sliding distance varies on the tilt of the left analog stick. If the analog stick is pressed from 0.6 to 1, then the player’s maximum velocity is multiplied by the maximum velocity – 10 and the sliding distance – 5 meters.

Tilt of Left Analog Stick (0 – 1)	Acceleration of Slide
0.2	15
0.4	25
0.6	50
0.8	50
1	50

Sliding can be used as an offensive or as a defensive move; while sliding, the player cannot punch. Sliding can be used to dodge attacks, catch up to players, or to avoid obstacles. If a sliding player hits another player, the other player trips, disabling them for 0.1 second. If a player slides into another player who is holding an ingredient, he/she trips but does not lose the ingredient.

TECHNICAL IMPLEMENTATION

Step One: Creating the Controller Class

A Controller class will be created in place of the character controller already present in Unity. Controller class will be used for all four players. The script is assigned to the players and the player number is set.

Step Two: Moving the Character

The characters will be moved by using Vector3s to find the speed and position at which they are moving, and then adding force to the players. One Vector3 will be used to get the axis the payers are moving on. This Vector3 will use the Xbox Control Master Plugin. This plug in gives us an XCI (Xbox Controller Input) class. Using XCI.GetAxis in place of Unity's Get Axis. A second parameter is used which is the player number. Player numbers are set in the inspector.

Step Three: Sliding

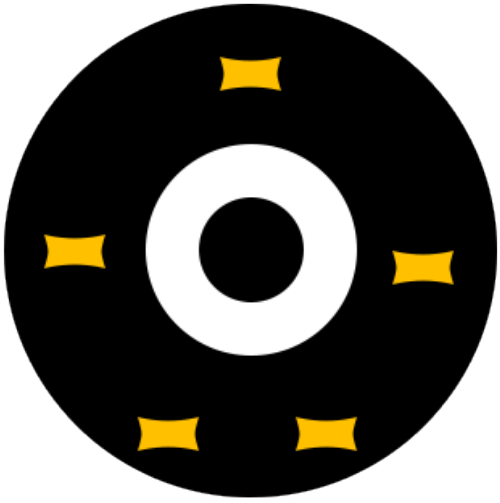
An if statement checks if the player presses the A button and the player number. If the dash timer is less than or equal to zero the Vector3 used to get the players axis input is multiplied by five. This jets the player forward in a quick dash. The dash timer is set to .5 and at the bottom of the update loop the timer is set to decrease if it is greater than zero.

Death and Spawn

Overview

Time estimation for following features: 10 Hours for Programming

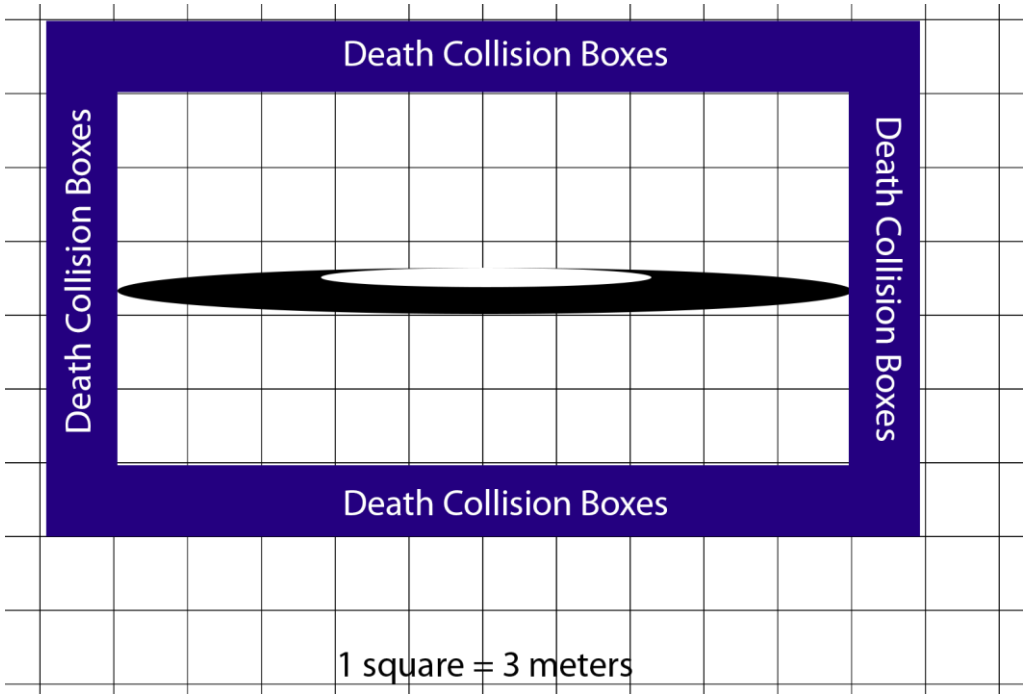
JukeBox Paradise will use a pooling system for better performance. When players are killed they will be moved off screen and then moved to a spawn location after a period of time. There are five re-spawn locations on the record.



All of the playable characters do not have any health properties. However, there are four death collision boxes that surround the stage. The collision boxes that are located on the top and bottom of the record are 30 meters wide and 3 meters high. They are six meters away from the record. The collision boxes that are located on the left and right side of the record are 21 meters high and 3 meters wide. They are located three meters away from the record.

When the player hits one of the collision boxes, the death event is automatically triggered. There is no death animation for the player but when the player collides with a death trigger, then the character's death SFX plays. After 0.5 seconds, the player's is no longer visible to the player. After two seconds, the player is re-spawned at one of the five re-spawn locations on the record.

If the player possessed an ingredient, once the ingredient hits the death collision trigger, the interactable object acts as if they were thrown off the stage. They are re-spawned on the stage after a random interval between 1 to 10 seconds at one of the five spawn locations. The random interval is set by using Unity's Random.Range() function.



Technical Implementation Plan

Step One: Hiding the Player

When a death condition is met the player is moved off screen and no longer rendered. The player is set to not use gravity and becomes kinematic.

Step Two: Respawning the Player

After a second passes the player is moved to one of the spawn locations using an array of game objects and the random range function. The player then becomes rendered, uses gravity and is no longer kinematic.

TAUNTING

OVERVIEW

Time estimation for following features: 5 Hours for Programming

The characters in JukeBox Paradise have four unique taunts for each character.

TECHNICAL IMPLEMENTATION PLAN

Step One: Button Press

An if statement will be put into the Players class. The statement will use the **XCI.Get Button Down** and then check the Player's number. When a player presses the left trigger, a taunt animation and sound effect will play.

Step Two: Stopping Other Actions

A bool will be created to check when the player is taunting. This will stop the player from performing actions other than moving while taunting.

Environmental Obstacles

Overview

Time estimation for following features: 20 Hours for Programming.

Spinning along the record are dangerous obstacles that harm the players.

Technical Implementation Plan

Step One: Import the Assets

The artist will create environment hazards in Maya. The assets will be imported into Unity and set to a scale of 1.

Step Two: Creating the Object

On the record there will be spawn locations for the objects. These locations will be empty game objects. The object will randomly pick a location check if there is currently an object there. If there is no object a particle system will player to alert players. After the particle system plays the object will be created at the spawn location.

Step Three: Moving the object

A new class will be created for the objects. Inside of the script the objects will spin on the record using the rotate around function and setting the point to Vector3 zero.

Step Four: Collision with Player

Inside of the player class the collision is handled. The players will use On Collision Enter and check for collision on the layer for obstacles. If the player collides with an obstacle, the players death and respawn functions will be called.

Step Five: Removing the Object

When the object is created a timer will be set using random range. The timer will count down between a min and max time for the obstacle to be on the record. When the timer reaches zero the object will be destroyed.

Game Characters

Overview

There are four playable characters in *Jukebox Paradise*. All four characters have the same abilities. The only difference in the characters is the appearance. Players choose one of the characters before the game begins and plays as them the duration of the game. Players cannot choose the same character. They are all in their late teens, and they all work at the *Jukebox Paradise* diner after school. There are two females and two males in *Jukebox Paradise.* There are no NPC characters in *Jukebox Paradise.*

All of the players can run, punch, slide, pick-up objects and throw objects while on the arena.

Sophia

Sophia is 17 years old. She likes to hang out at Jukebox Paradise after school with her friends and enjoy a milkshake. In her spare time she practices hula hopping and watching Tom & Jerry with her younger sister.

Louis

Louis is 18 years old. Louis spends his Friday nights playing guitar at Jukebox Paradise for the Groovy Boys. He aspires to be the next Elvis or Chuck Berry.

GINGER

Ginger is 18 years old. She can be found at the back of Jukebox Paradise smoking with Johnny. Ginger is a beauty school dropout.

JOHNNY

Johnny is 19 years old. He is seen smoking behind Jukebox Paradise with Ginger. Johnny always tries sweet talking the waitresses. Johnny does not know where he is going, but he knows he is going to leave Groovsville one day.

TECHNICAL IMPLEMENTATION PLAN

Step One: Import the Asset

The artist will model, rig and animate the characters in Maya and they will be imported into Unity.

Step Two: Replace Capsules

The capsules used as place holders for the characters are now replaced, with the character models.

Step Three: Test Characters

The characters will need to be tested for their animations and movement.

NPC Characters

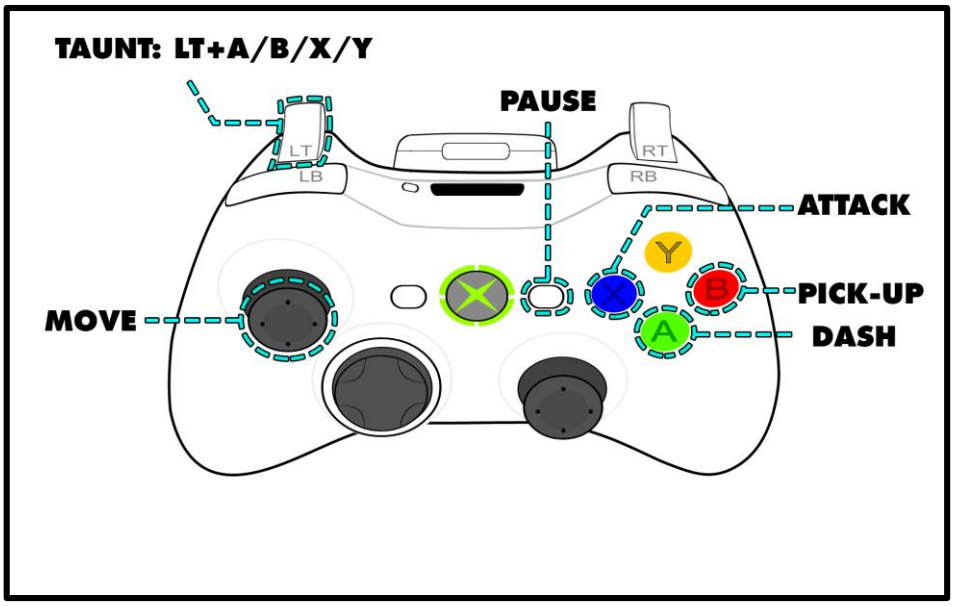
Enemies and Monsters

USER INTERFACE – CONTROLS

OVERVIEW

Time estimation for following feature: 10 Hours for Programming

JukeBox Paradise can be played with both a gamepad and a keyboard. All the main actions in the game are mapped to a single button, except for the taunts that use a combination between two buttons. The reason for this is that the main features needed to be simple to perform, while the taunts are just complementary to the gameplay experience in an aesthetic way.



XBOX CONTROLLER

ACTION	GAMEPAD
Move	Left analog stick
Punch	X
Pickup	B
Dash	A

Taunt	Left trigger + A/B/X/Y
Pause	Start button

Technical Implementation Plan

Using the Xbox Control Input Master plugin we will move an XBLControll.dll file to the projects folder, and replace Unity's Input Manager with a new one within of the plugin.

Step One: Creating a Controller Script

A new class will be created in place of the Character Controller already in Unity. The script will use a Vector3 that gets the X and Y axis.The Vector3 and it is multiplied by the movement speed of the character.

Step Two: Moving the Character

A new Vector3 is created and set to the rigidbody velocity. A new Vector3 is created and subtracts the first from the second. The X and Z variables are clamped and the Y is set to zero. The add force function is the parameters are the new Vector3 and force mode velocity change.

Step Three: Gravity

Gravity is set using the add force function. Gravity multiplied by the mass is used for the Y axis.

Step Four: Rotation

An if statement checks the magnitude of the Vector3 used for movement. When the magnitude is greater than .01 the player will rotate. A new Vector3 will be created and set equal to the movement Vector3 normalized. The transform rotation will be set to Quaternion Slerp and move from the transforms rotation to the Quaternion Look Rotation of the new Vector3 we created. The final parameter time will use Time delta Time.

Step Five: Moving With the Record

When the magnitude of the movement Vector3 is less than .01, the player will move around the record using the Rotate Around function and setting the point to Vector3 zero.

Audio

Overview

All of the audio assets for *Jukebox Paradise* will be implemented with Unity's Engine using an external plug-in, Audio ToolKit by ClockStone.

Format

For all SFX, we will be using 44.1 K 16-bit mono WAV tracks and 44.1K 16-bit stereo WAV tracks for background and ambience music.

Sourcing

We will outsource audio assets to external audio collaborators from the Sound Design Campus. We assume that we will be receiving all of the audio assets – character SFX, environment SFX, and background music for all three levels.

In the case that the project does not receive any collaborators, we will either outsource audio assets to other external collaborators that we have available. Or, we will re-scope the game design to allow one of our team members to produce audio assets from 5Alarm Music under the guidance of the audio mentor, Steve Royea.

In the case that the audio assets do not meet the project's standard of quality, then we will work with the audio collaborators to guide them in the right direction.

Sound Detail

Music

There are three background music tracks for the three levels in *Jukebox Paradise,* and one background music track for the Front End Menu.

All of the background music tracks are required to be looped in the case that an endless game mode is added on the production floor. The music tracks for the levels are all the base tracks and they are always triggered o start when the level loads.

The Front End background music can be shorter in length but it must be looped track for about 45 seconds to 1 minute.

Sound Asset	Description
Level 1	• 1950s Dance Music • Doo Wop

Level 2	• 1950s Dance Music • Rockabilly
Level 3	• 1950s Dance Music • Dirty Rock 'n' Roll

AMBIENT SOUNDS

ENVIRONMENT SOUNDS

The environment sounds are looped tracks that play in the background along with the background music tracks.

Category	Sound Asset	Description
RecordSpinning	Low whoosh	Follows the record as it spins
Gears Turning	Clicking	Clicking noises in the background, plays intermittently
Diner Ambience	Voices, glasses, cutlery, drinks being poured	Low pass filter, resonance, indoors

CHARACTER SOUNDS

Both vocalized and non-vocalized character sound effects are triggered when corresponding actions occur. Vocalized sound effects have a 0.1 – 0.2 second delay after their corresponding action sound effects play.

Category	Sound Asset	Description
Character (All)	Power-up: Punch	Hitting the floor, bigget impact, booming thud sound
Character (All)	SFX: Slide	Cartoonish, slipping on ice
Character (All)	Run	Cartoon footsteps on a hard surface
Character (All)	Dash	Speedy, short, almost like a car
Character (Women)	Voice: Cheer	Woohoo, cheering noise, female
Character (Men)	Voice: Cheer	Cheering, male

Character (Women)	Voice: Ow!	Cartoon, short
Character (Men)	Voice: Ow!	Cartoon, short
Character (All)	Tackle	Body check, hit, quick
Character (Men)	Voice: Tripped	Surprised gasp, short, not fearful?
Character (Women)	Voice: Tripped	Surprised gasp, short, not fearful?, higher pitched
Character (All)	Collision: Punched	Hit, punch with impact, like in the face
Character (All)	Collision: Tripped	Body thud with a whoosh?, Cartoonish
Character (All)	Powered Up Punch	Big IMPACT, ground breaking shatter
Character (Women)	Voice: getting hit by an object	Quick vocalization, "ow?", surprised pitch
Character (Men)	Voice: Getting hit by an object	Quick vocalization, "ow?", surprised pitch
Character (All)	Faster running	Quickened sound of footsteps

Object Sounds

Object sound effects are played when players collide with interactable objects in the game world – power-ups and milkshake ingredients.

Category	Sound Asset	Description
Interactable Object	Collision: Picking up Power-Up	Sparkling, short burst
Interactable Object	Collision: Picking up Ice Cream	Slurp like someone's eating ice cream
Interactable Object	Collision: Picking up Fruits	Squish sound like you've stepped on a fruit
Interactable Object	Collision: Picking up Special Ingredients	Zing?
Power-Up	Indicating shield on/off	High-pitched beep, short

HUD and Menu Sounds

Audio: Menu

All of the audio assets for the Front End UI will be implemented into Audio Toolkit's AudioController, and they will be called when players are selecting their characters and choosing their game modes.

The background music for the menu will be looped until player leave the Menu. When players begin the game, the background track will stop with a fade-out by using the method, *StopMusic(Single).*

For the sound effects in the menu system – mouse over and mouse click – they will play at the specified location of the cursor. The sound effects for the menu system will be called with the method, *PlayMusic(String,Vector3,Transform,Single, Single, Single).*

Category	Sound Asset	Description
Mouse Over	Option Moused Over	Button press, click
Preloader	Coin inserted into machine	Coins being fed into vending machine
Menu Switching	Whirring	Motor whirring
Mouse Clicked	Option Selected	Button press, click
Front End Music	1950s Dance Music, BoogieWoogie	Plays in the background

Audio: HUD

The sound effects for the HUD will play according to their specific audioID. For the goal sound effects, they will be called to the specified transform of the game object with the method, *Play(String, Vector3, Transform).* The volume for the goal sound effects will be louder than the background music to ensure that players receive clear auditory feedback. Using the *SetCategoryVolume* method, the volume for goal sound effects will be slightly higher than the background music tracks.

The level complete sound effect is triggered when players finish a level.

Category	Sound Asset	Description
Level Complete	Cheers, Trumpet playing	Plays when level is complete
Goal	Ding, ding, ding!, bell like a boxing match	Plays when an ingredient is successfully in the milkshake!

Technical Implementation Plan

- Step One: AudioController
  With the Audio Toolkit package imported into the Unity project, we drag an Audio Controller prefab into the hierarchy. There will be one Audio Controller for the entire project, and all of the audio assets will be called from it.

- Step Two: Import Audio Assets into Unity
  All of the audio assets for the game will be imported into Unity into their appropriate folders: Background Music, Character SFX, Object SFX and Ambience Tracks.

- Step Three: Implement them into AudioController
  Within the Inspector, all of the audio assets will be added onto the Audio Toolkit's AudioController as sub-items. All of the SFX and background tracks will be called from the AudioController. The audioID will differentiate between the background music, character vocalized SFX, character non-vocalized SFX and object SFX.

- Step Four: Call Background Music (Scripting)
  All of the background music for the three levels will be put into a *musicPlaylist* and each song will play for each level. Once a song ends, then it will move onto the next level. It will not be a *loopPlaylist*. After players select their characters and the game mode, the background music tracks will play by calling the *PlayMusicPlaylist* method.

- Step Five: Call SFX (Scripting)
  The sound effects for characters and interactable objects will play according to their specific audioID. For the sound effects, they will be called to the specified transform of the game object with the method, *Play(String, Vector3, Transform).*

Testing

Overview

Play testing will begin when we move to the second floor. We will ask at least one new person to play test our game every week. At the end of each week we will create a build for play testers.

After M1 we will begin to seek new play testers. We feel M1 is the best time to increase play testers because the game will have most features in and we will be able to receive feedback to see what works, and what does not. The project Manager Maria Lee is in charge of finding play testers and the Programmer Frankie Fasola is in charge of conducting the play tests, as well as documenting feedback.

Every morning during our stand up meetings at 10 A.M., we will discuss any feedback given by play testers. During these meetings the team will review the feedback and decide if the game needs to be fixed, changed or if the feedback should be ignored.

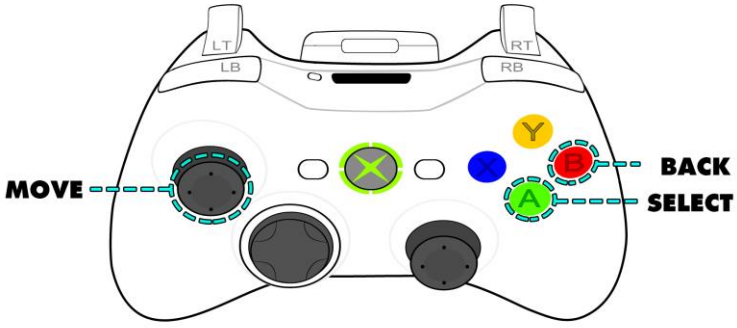
GUI & HUD

Overview

This section details the layout and elements of the game's interface. The game has a screen aspect ratio of 16:9 at a resolution of 1920 (width) pixels by 1080 (height) pixels. All the User Interface and HUD in the game are designed to that resolution and aspect ratio.

Menus

The menus in jukebox paradise are designed to get the players into the gameplay as fast as possible, while setting up a proper mood for the game. For this, we want to have strong visuals on the menus but clear selections and easy navigation. The navigation is designed to be used with the Xbox 360 controller.

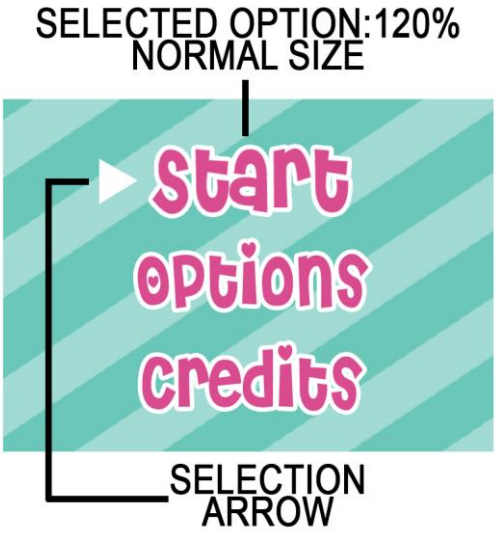


For navigation, the menus will highlight an option as a default, and players can move the highlighted option with the left analog stick. To select the option highlighted, players can press the A button. All screens in the game allow players to go back to the previous screen by pressing the B button, except the main menu.

All typography for the menus is set to cherry, although the final choice will be decided on the execution of the interface.

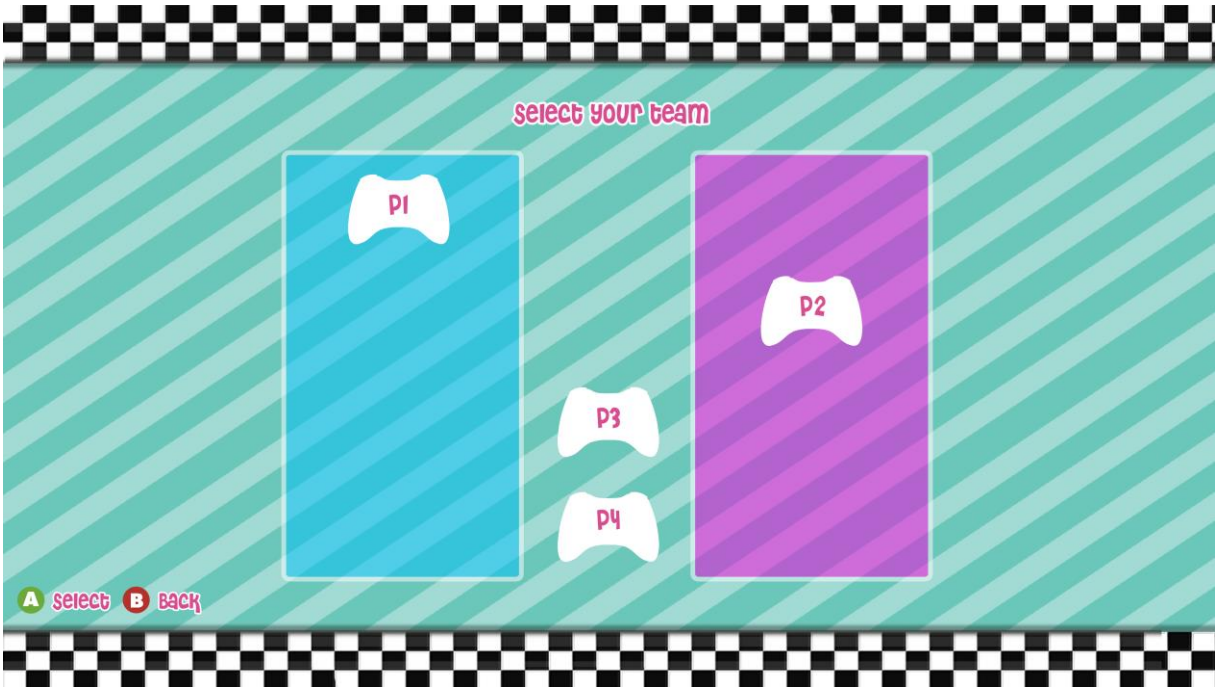
When the game starts, the VFS slate screen will appear, and after that, the Maun Menu will welcome the players.

The main menu shows 3 options presented at the center of the screen, below the Jukebox Paradise logo The first option highlighted is the Start Option since we want the player to get into the gameplay as fast as possible.



Highlighted options grow 20% from their normal size, accompanied by a triangle arrow displayed at the left of the text.

Only Player 1 can control the menus, the rest of the players can only interact in the menus for the team selection and character selection screens. When Player 1 presses the A button on a highlighted option, the current screen will slide to the left and the next menu screen will come in from the right.



The team selection screen shows the players the two sides that they can pick. Using the left analog stick players can move their controller icon to any of the 3 places available, which are the blue team space, the pink team space, and in the middle. When the Player 1 presses the A button, the menu proceeds to the character selection screen. If two controller icons are occupying the same team space, the other two controllers won't be allowed to be moved onto that team space, forcing them to stay on the middle or the other team. In case a controller is left in the middle, the game shows a message asking the players to move their controller to a team space in order to proceed.

The character selection screen will allow players to choose their character. Every player has its own box where the characters are shown for them to choose. When they decide to pick one, the character gets highlighted by a glow and becomes unavailable for other players to choose, showing them a grayscale version of the character's image. When all players have picked a character, the game proceeds to the level select screen.

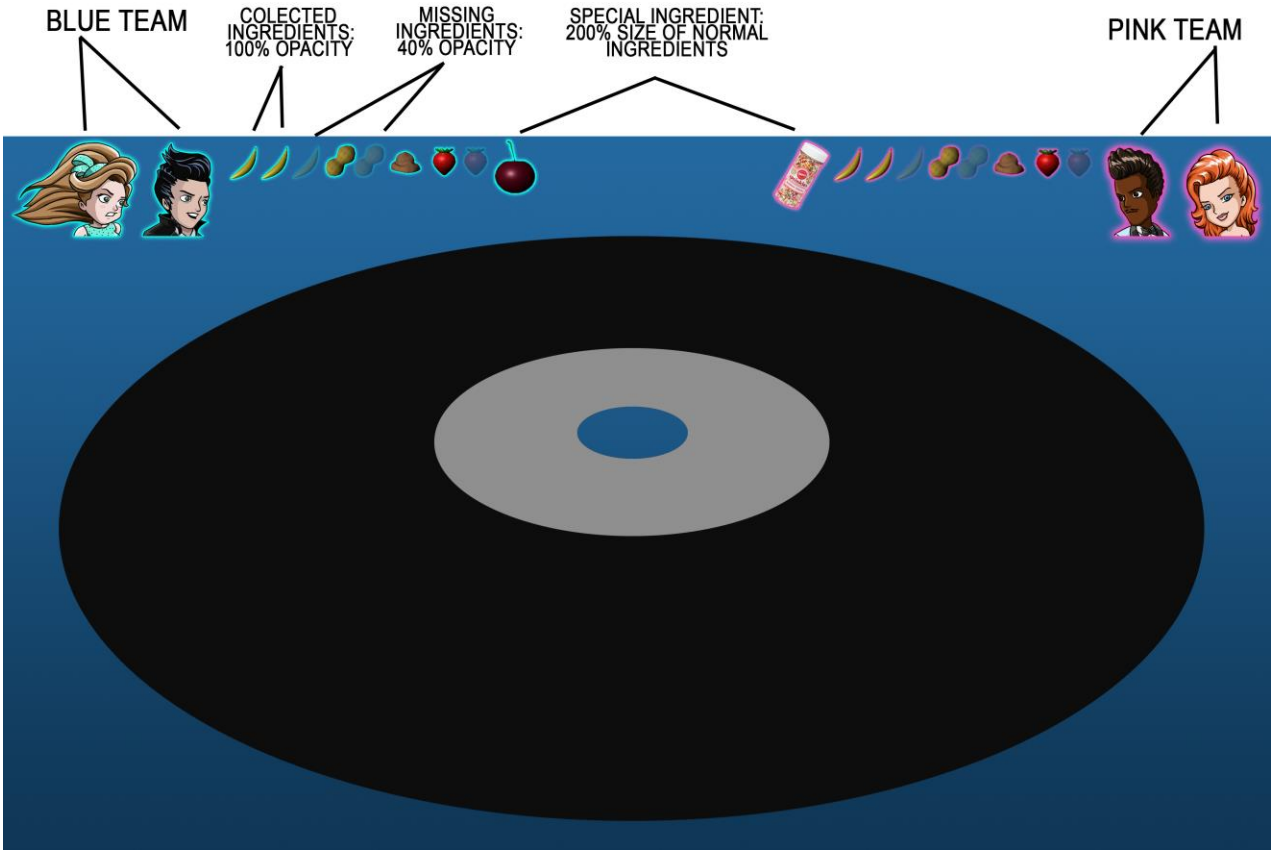
On the level select screen, the three available levels are shown. Depending on the option highlighted by Player 1, a short music loop related to the level plays for them to feel the mood of the level and pick the one they like want the most at that given moment. When the level selection has been completed, the game starts.

HUD

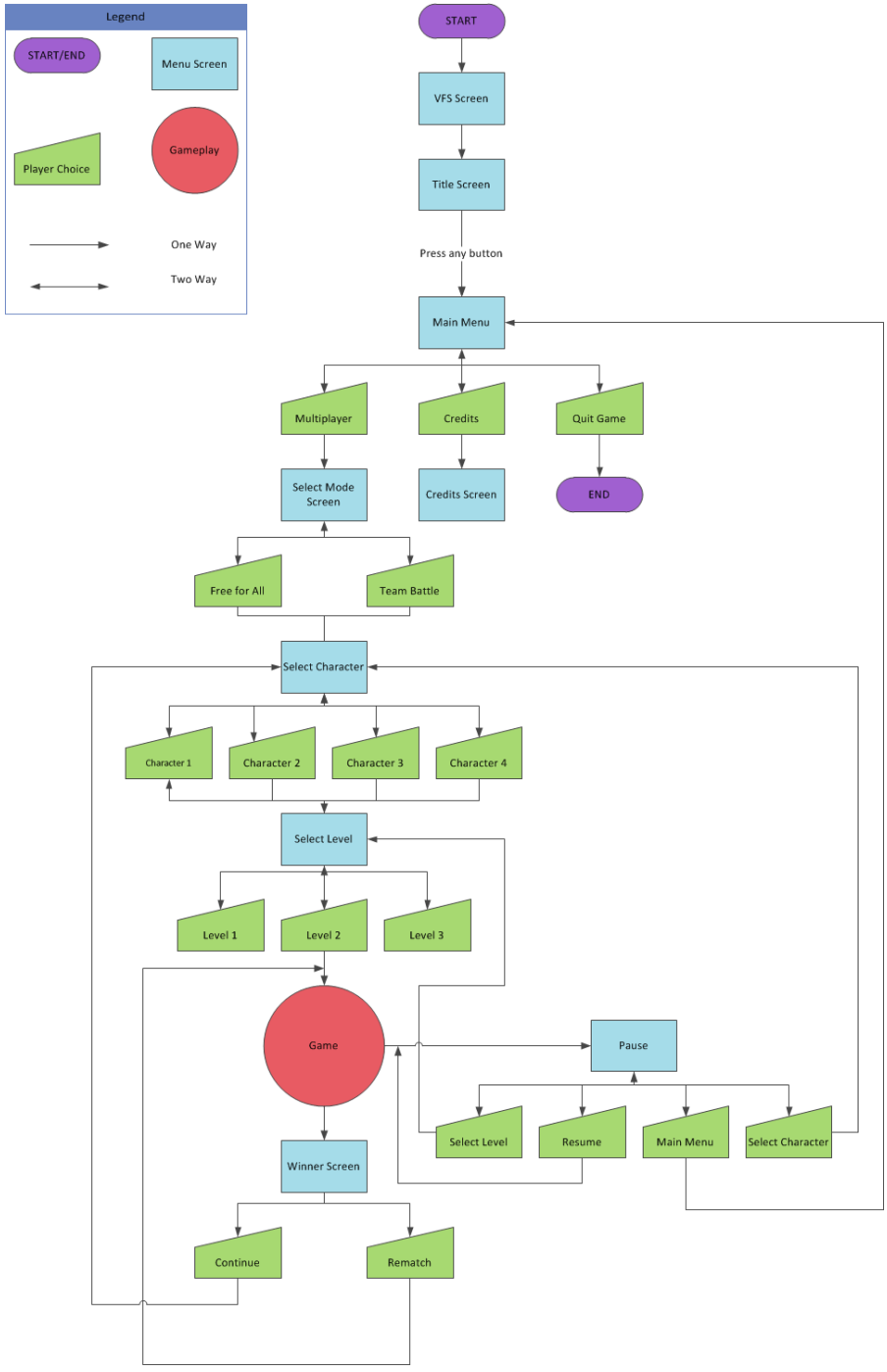
The HUD in jukebox paradise is designed to aid the players on their mission to create a milkshake by showing them the ingredients that they need on the top of the screen. To help the players identify which side they are on, an image of their chosen character is displayed on the top corners of the screen. The Blue team is on the left side of the screen, while the Red team is on the right.

To help the players identify themselves even more, a glowing aura of the color of their team surrounds their character's image, as well as the ingredients that they need. These ingredients are displayed with 50% opacity on the screen, and as soon as players collect that ingredient, the image becomes 100% opaque.

Each team has one special ingredient that they need to collect, and it is displayed by the closest item to the center of the screen. It is also 200% the size of the rest of the ingredients, so that players can identify it easier and keep it in mind.



User Interface Flowchart



Technical Implementation Plan

Time estimate: 32 hours for design and 32 hours for programming
Jukebox Paradise's user interface will be done using the NGUI plug it for unity available on the asset store.

MENUS

Step One: Create the Menu Camera
A different camera will render the game menus.

Step Two: Create the screens layout with placeholder art
The layout for the different screens of the menu will be created before any art or functionality. This helps us to get a solid base to work on both functionality and art.

Step Tree: Create controller functionality for each screen
The controller navigation to move the highlighted selection between the menu options will be done by creating identifiers for each option and then linking them to the other options. This way, the game will ask itself when it receives controller input if there is any option in the direction that the analog stick is pointing. If there is something, it will move the selection to the item referenced in that direction.

Step Four: Create functionality between screens.
The next step is to actually get the game moving between the different screens of the game. This will be done by scripting the conditions for each button and adding the behaviour for it to move the UI.

Step Five: Store values of the player's choices for the game
The next step is to get the player input to actually set values and assign the proper game elements for when the game starts. This will be a simple script that identifies the player number and what option they chose, and then that character will be assigned to their input in the game. The choice for the levels will decide which scene we load.

Step Six: Test functionality
Once everything is set to work, we'll test the functionality to see that everything is working properly

Step Seven: Place final art and effects
At this point, we replace the placeholder art with the final art and incorporate any effects and special animations that we want. The art will also be assigned to an atlas texture to reduce the amount of draw calls. Text will be rendered on a single draw call while the images will be rendered by another one.

Step Eight: Test and Polish
To finish the UI we will test the functionality and look of it and make any changes that we need to refine it.

HUD

Step One: Create the HUD Camera
A different camera will be rendering the HUD and it will be rendered on top of the game camera so that we can lock down the position for the HUD elements as well as having the possibility for the game camera to move.

Step Two: Create placeholder art and create layout with it
We will create the layout for the HUD with simple art so that we can test the functionality first.

Step Tree: Create functionality
The HUD behaviour will be referencing to the game state to know what to display. Any values and events that the game has will be fed to the HUD for it to react on it and show the user their actions reflected on the state of the game.

Step Four: Replace placeholders with final art
The final art for the HUD will be placed and any adjustments to the functionality and placement of it will be done here.

Step Five: Test and Polish
The functionality and look of the HUD will be tested and tweaked with any necessary changes to give it the final look and functionality.

APPENDICES

GENERAL INFORMATION

During production, all team members will be using the Unity Asset Server for source control on a daily basis. At the end of each working day, all local changes to the project will be stored on the server by using the **Commit** tab. To minimize the number of conflicts between the local and server changes, all team members will specify working tasks at the daily stand-up meetings.

Frankie Fasola will be responsible for backing up the game to the “last playable” build. This will occur at the end of each working day.

All other assets and documents will be uploaded onto Perforce as needed.

APPENDIX A: PIPELINES

NAMING CONVENTIONS

Jukebox Paradise will adhere to the following naming convention:

AssetType_AssetName_Author_Version.FileExtension

Example #1	Char_Model_Juan_AM_V1.ma
Example #2	Char_Texture_JuanDiffuse_AM_V1.png

FOLDER STRUCTURE

- Pre-Production
  - Concept Document
  - GDD
  - TDD
  - MDD
  - Mentor Pitches
  - Audio Pitches

        - Prototype Archives
    - Production
        - Build Archives
            - Daily Backups
            - M1
            - M2
            - Alpha
        - Audio
        - Art
            - Character
            - Environment
            - Logos?
    - Project Management
        - DAC + Release Forms
        - Software Requests
        - PM Assignments
        - Document Templates

Appendix B: Notes on Prototypes

Build 1:

The first build of Jukebox Paradise was a 2D multiplayer game with punching. Through the prototype we discovered the idea of multiplayer was good, but we did not like the 2D aspect.

Build 2:

The second build of *Jukebox Paradise* was an online multiplayer platforming game, but this time, the mechanics included skating, punching and collecting objects. The camera was third person non controllable. Through play testing the prototype, we came to the conclusion that the best part of the game was punching. The skating mechanic was compared to a gimmick and the camera system did not fit the game well.

Build 3:

The third build introduced the idea of a Jukebox and the players on the record. The camera was moved to an isometric view. The players had the ability to jump, punch, tackle, dash and collect power-ups. The camera is static for the game so the idea of networking multiplayer was dropped. There was no fun in the game outside of punching each other and it felt as if the game was not quite there yet.

Build 4:

The fourth build added the milkshake glasses on the sides of the record and collect milkshake ingredients. The players could now pick up and throw objects, as well as drop them. After the dry run for the mentor pitch, the instructors mentioned there was an excessive amount of controls and the same game could be achieved with two buttons. Redesigning the controls was the focus for the mentor pitch.

Build 5:

The fifth build was built twice with two control schemes. The first had three buttons, eliminating the jump tackle and dropping of objects. The second build had two buttons and cut the same mechanics. The punch was improved to allow the players to charge up hits. Through a charged up punch, the same effect of tackling could be achieved.

Jumping was seen as not important for the game because the slide could be used to avoid obstacles and players. It also could be used as an offensive maneuver. Throwing was changed to incorporate the same system as punching. There is a base throw power and a max throw power. Throwing was also modified to behave like a basketball with an arc trajectory. Finally, the players and objects were set to move with the record.

Appendix C: Software Request Forms

Tool/Application Name

- **NGUI:** A UI system for Unity
- **Audio Toolkit:** Audio Manager for Unity
- **Xbox Control Input Master:** An open-source plug-in that supports four XBOX controllers in Unity

Number of Students Who will Use It

- Frankie Fasola
- Alberto Mastretta
- Maria Lee
- Janel Jolly

IDs of computers where the software where need to be installed

- To be determined

How it fits into the pipeline

- **NGUI:** Due to the multiplayer nature of *Jukebox Paradise,* our game requires several front-end UI screens that allow players to choose their individual characters. We will also have a game mode selection screen.
- **Audio Toolkit:** Since our game has numerous audio assets and relies heavily on gapless looping of background music, we will be utilizing Audio Toolkit's AudioController.
- **Xbox Control Input Master:** Unity's Input Manager is not optimized for multiple controller inputs. The Input Manager gets confused between controllers and responsive control is difficult.

Brief description of how a demo been tested to verify with 100% certainty the product does what you need.

- **NGUI:** In Unity 2 class we had a demonstration about NGUI and how it works. The artists took time after class to test out NGUI and decided it suits our needs for user interface. In class we were able to add health bars to enemies and create an example menu.
- **Audio Toolkit:** Using the plugin we were able to control the audio through one source and it was categorized for us. Calling audio, filtering sounds and audio effects were accessible and easy to implement.
- **Xbox Control Input Master:** Using the control input we were able to achieve four player multiplayer for our prototype. The plugin has simplified coding the controls and is reliable for players during the game.

WHAT TOOL DOES VFS CURRENTLY HAVE INSTALLED FOR THIS WORK.

- **NGUI:** Unity's OnGUI system.
- **Audio Toolkit:** None.
- **Xbox Control Input Master:** Unity's Input Manager.

STATE WHY THE EXISTING PRODUCT UNSUITABLE.

- **NGUI:** OnGUI is expensive for memory during gameplay and hard to use. NGUI simplifies the UI process.
- **Xbox Control Input Master:** Unity's Input Manager is unreliable because it mixes up the controllers input when multiple are plugged in.