**CMP208 Coursework Report   
Cameron Thustain – 1900594**

**Tumble Chaps**

# Introduction

Tumble Chaps is a simple yet fun catch game that takes inspiration from the unique 2020 hit game Fall Guys by Mediatonic. Many catch games have been made over the years so Tumble Chaps only takes inspiration from the characters, art, and fun child-friendly music of Fall Guys (see Figure 1). The overall gameplay and mechanics do not replicate the levels of its inspiration. The general purpose of the application is to demonstrate the use of box2d physics and body transformations upon collision.

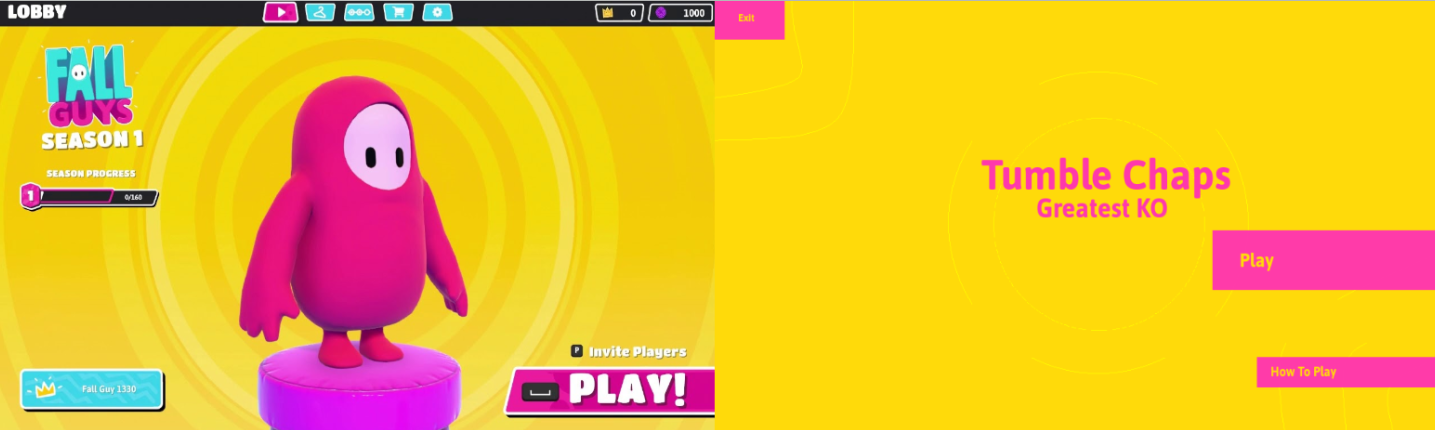


Figure 1 Fall Guys Menu - Tumble Chaps Menu

Tumble Chaps involves a player who can navigate along green terrain. The players' main goal is to catch tennis balls that fall from the sky (see Figure 2). For every ball, the player catches a point is gained. If the player misses the ball it falls to the ground and the player loses confidence which is the game's take on lives. When the player catches the required amount of balls they win. However, if too many hit the terrain, they lose! The application includes a vary of different states, an options system that allows the player to stop the music and pick a difficulty for the level. Furthermore, the application includes a variety of different audio which is played upon collision, button presses, or state changes.

The randomness of the balls spawning creates a fun feel game, that keeps players on their toes, with the potential for further progress, features, and input.

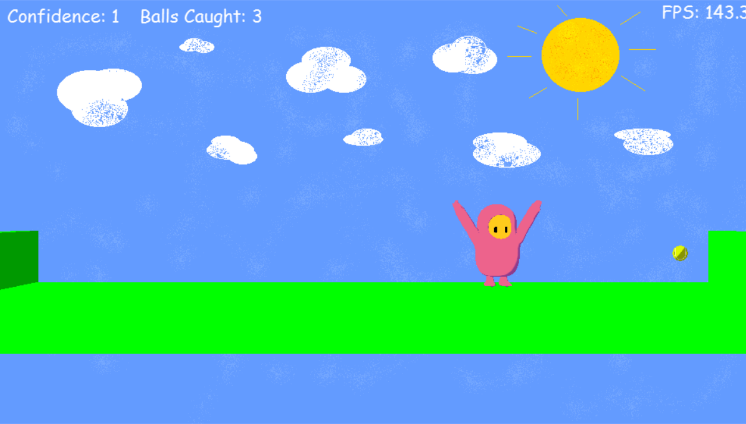


Figure Tumble Chaps Level

# Application Design

The application itself uses object-oriented programming so the code is separated into classes for better understanding, better troubleshooting, and problem-solving.

The application sets up global instances of the gef classes such as platform, font, and input within our main state manager class which are mentioned below:

- Font

- Platform

- Sprite Renderer

- Input Manager

- 3D Renderer

- Physics World

- Pointer to the audio class which sets up audio manager within.

# Diagram Description automatically generated States

Most of the program is spent within the state manager class where a global pointer of that class is created within scene\_app. The individual states are stored with an enumeration which is then created within the state manager (C++ Enumerations | Learn C++ Online | Fresh2Refresh.com, n.d.). Each possible state then has its own initialise, update,

 render and release function within the state manager. This design was chosen because it’s simple and there is only a small number of states within the application.

The applications states go as follows (see Figure 3):

* Init (Splash Screen)

Figure 3 Flow Chart of Games States

* Menu
* How to Play
* Options
* Level
* Win
* Lose

The states are changed using a function that will switch the enumeration to the next state. The separate state's function is called depending on what the current active state is. After each state is changed the objects that were used within that state are appropriately deleted for better memory management. Only the current state functions are called because of this, memory is saved by not having every individual function called at once but also makes troubleshooting the code much easier (C++ Enumerations | Learn C++ Online | Fresh2Refresh.com, n.d.).

# Level Assets

Initially, the terrain and assets within the application were in their separate classes. Later it was decided to move them into one to help with the setup of collision and if the application was to be expanded on, no more than one class would be used per level. The level asset class is responsible for creating the terrain, loading in the player and ball model, and rendering these assets when appropriate to do so.

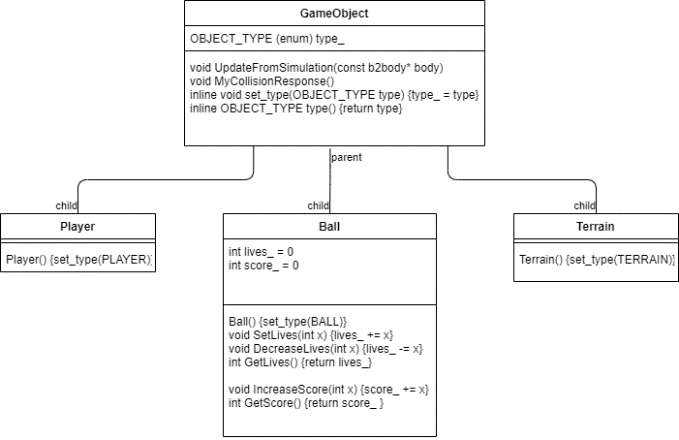


Figure GameObject class & subclasses

# GameObject Subclasses

Within the game object class subclasses of each object type are created (see Figure 4). Because of this, specific object types such as Ball could be given the score and live variables and functions instead of the parent class. This was decided as not every object will have or need these values. The ball was given a score and lives due to the style of the game. If the ball was to increase with any terrain lives would be decreased or if the ball collided with the player, the score would increase. This works fine due to their only being one ball on the level. However, if to be expanded on this would be an area that could be redesigned if more balls were implemented. Furthermore, each subclass of the game object gets to set a type within their constructor which comes in useful when setting up collision in the application.

# Audio

Audio is used frequently throughout the application so it was decided that it would have its class where sound and music samples could be initialised, played, and cleaned. The only time an instance of Audio Manager is created is in the audio class, this was to save memory. Audio is only initialised within that class for better troubleshooting if something was to go wrong. When other classes required audio a pointer to the audio class is created so the sound sample doesn’t need to be reinitialised. In many cases for playing audio, a sample had to be specified to make this less costly an int32 was passed within the function which plays the sounds. From there whenever a sound had to be played whether it had to be after a collision, button press or a change of state only one value had to go into the function this saved a lot of time and was very simple and quick.

# Reuse of classes

Many of the objects with the program are only created once. Within the gef classes such as input, sprite renderer, 3D renderer, and more are passed by reference around into the different classes of the program. Most of the game states require these classes for basic functionality therefore they are used all over and are only deleted if necessary. Because we are passing classes, passing by reference was most beneficial here.

Firstly, it saves recreating instances of classes in every single class which requires it. Doing this saves memory and they only must be initialised so there is no worry about any null values. Furthermore, due to the state manager being a large class passing by reference is fast. However, it can be difficult or at least very easy to make mistakes when passing them around classes which can cause null pointers if not done correctly.

# Reuse of code

One important aspect of the application was the attempt to reuse a lot of code due to the number of similarities in some areas. With each state having a background in the same position and size or the use of the same button icons between states. The application was designed so there wasn’t lots of repetition with code and a sprite or text could be rendered or changed with one line or one variable. This was also used further when setting up the difficulties for the game with both having the same variables but of a greater or smaller value much like the players' lives or the gravity scale of the ball physics body. This made scalability a lot easier and overall a lot simpler and quicker. Furthermore, adding new states or new features such as difficulties could be achieved much quicker and easier.

# Techniques

Enumerators

Throughout the application, enumerators were used to hold the application states and different object types within the game object class. The use of enumerators meant that each value in the enumerator could have its fields and methods. Each object was given their type apart from the ground and walls in the level, this was done because they don’t have many differences, so they were set to a type called terrain. As well as the objects each state was put into an enumerator as well. This was decided purely for type safety and because the state manager was set up to use a switch case (C++ Enumerations | Learn C++ Online | Fresh2Refresh.com, n.d.).

States were stored within an enumerator called ActiveStates which held the following:

* NONE
* INIT
* MENU
* HOWTO
* OPTIONS
* LEVEL
* LOSE
* WIN

Like mentioned prior these were then used in a switch case. When the state would change if it was tested against the values within the switch statement the appropriate function for that state would be called.

Within GameObject the type enumerator held:

* PLAYER
* BALL
* TERRAIN

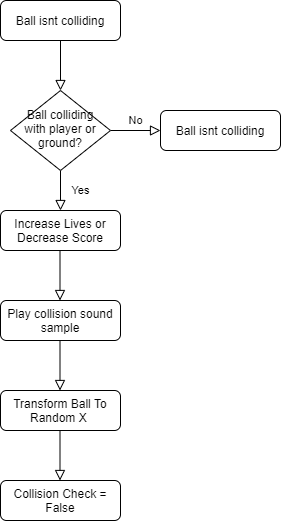
 This small enumerator came in very helpful when setting up collision firstly each type was created into a subclass of GameObject and within their constructor’s their appropriate type was set. From there we could give subclasses members and methods. Collision checks when one object collides with another, so these object types were used thoroughly throughout in collision checks and setting the assets within the type of the level.

Figure Ball collision flowchart

# Ball Spawn

A main mechanic within the application is the randomness of the ball spawning after each collision with either the player or terrain. This is done using a function within the level assets class. The function is set up, so the ball gets transformed to a fixed value on the y axis and a random position along the x-axis (rand - C++ Reference, n.d.). From there the body of the ball is set to awake so the physics continues to act on the ball even after the transform. The function is then called after the collision is checked between the ball and player or terrain (see Figure 5).

# Difficulty

One feature of the application is deciding the difficulty of the level within the options state. This was decided so the user can experience a spike in difficulty if they find the first time playing easy. When changing the difficulty, it changes the number of lives the user has and the overall gravity scale of the ball. To do this a function was created inside the level assets class which takes a float variable which is then passed into the gravity scale function. When the user picks a difficulty the gravity scale is increased. For example, if the user was to pick hard the scale would be set to 0.04, this is only an increase by 0.03 from normal mode However, it’s enough to make the game slightly more challenging for the player.

# Collision

A small yet important mechanic added was a collision check. This involved a Boolean variable which is initially set to true. This Boolean is used within the collision function, the for loop iterates through each contact while it's less than the contact count and that the collision check is true. When creating the if statements checking if ball and player or terrain has collided it sets the Boolean to false. This was implemented due to an issue with the ball falling close to the ground and wall which would count as a collision between both objects. This caused two lives to be lost instead of one however, once this Boolean was implemented this issue was resolved.

# Repetition of sprites and text

Within the application, many sprites are rendered for button icons or state backgrounds. This meant a lot of repetition would be involved. To save memory and time multiple functions were created to either render a state background, a button icon, or text. Each of these functions would take in float variables for values such as width, height, and the position of the sprite or text. Due to each state background having the same size and position their render function only takes a pointer to the gef texture class, which are all individually initialised in their appropriate initialise functions. For example, the menu texture within menu init.

# User Guide

Running exe

The program is very simple to run and navigate. For the program to work an executable need to run with a controller plugged into the computer before it is run. Tumble Chaps supports control input, mouse input or touch controls could become an addition if the application was to be worked on further. The best controller to use for this would be a PS3/4 controller although a PS5 controller will work, with the new built-in speakers into the controller the audio of the application outputs through them meaning audio is very quiet and hard for the user to hear. An Xbox controller may work with the application as well however the controls would have to be worked out by the user. If the user doesn’t plug in a controller before running the exe input won’t work and the application must be restarted for it to work.

In-Game-Controls

States:

* **CROSS -** go through menu or pick normal difficulty
* **TRIANGLE -** how to play or pick hard difficulty
* **CIRCLE -** exit application or go back to previous state
* **DPAD DOWN -** stop music

Level:

* **DPAD LEFT -** moves player left
* **DPAD RIGHT -** moves player right

Within the application, there is a how-to-play section that also reiterates the objective of the game as well as the controls (see Figure 6). Furthermore, button icons are shown throughout to guide the user through the application.

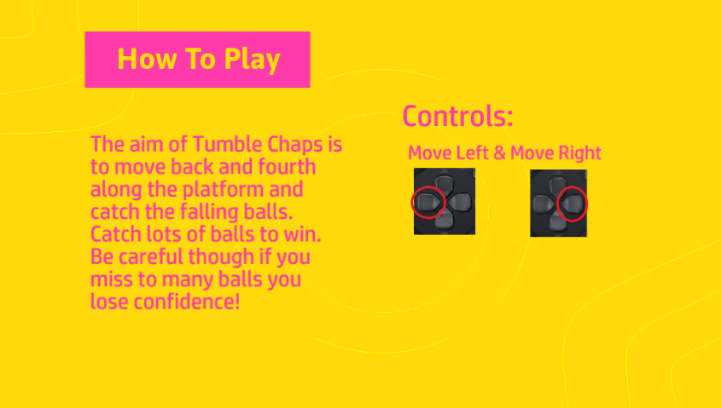


Figure How to Play Screen

# Data-Oriented Design

Many programs including this application use OOD this is because objects can be established, and it can be determined what sets of data belong to that object. However, using OOD means parts of data are restricted meaning they cannot be reused and concerns over memory speed arise. Overall using OOD does increase the performance of a CPU but it can’t be said the same about memory.

On the other hand, Data-Oriented takes a different route unlike OOD Data-Oriented uses data, not objects. This means everything can be acted on. With Data-Oriented the data is structured as closely as possible to the output data, so functions use minimum effort (Mines, 2018). One major benefit is code with easier implementation or parallelization. This means more than one core of a processor is used to execute a task. Unlike OOD there isn’t the risk of threads attempting to access the same data (Mines, 2018). The other obvious benefit of Data-Oriented Design would be an efficient use of the memories cache. Because the same functions are being used all the time the memories cache doesn’t have to keep storing instructions (Mines, 2018).

If DOD was to be implemented into the application a lot of changes would be made to most of the code. For example, the use of subclasses in the GameObject class does not work well with DOD. Furthermore, any containers used much like the arrays for some of the terrain within Tumble Chaps would need to be changed.

If Tumble Chaps were to be expanded on into a larger application DOD should be considered in many different aspects. Tumble Chaps currently only renders one ball with physics which is transformed off-screen to seem as if there are endless balls within the level. If it was decided that the difficulty feature was to be changed so more balls spawn the higher the difficulty, DOD should be used for better memory. Arrays of components could be used to render the balls as well as applying the individual physics to each ball instead of pointers.

When expanded upon it may be decided that the player model has a particle effect when moving. Particles could be set into a continuous array within a class for a single type of object. From there the particles could be kept track of (Nystrom, 2014).

Currently, the applications states are managed in a big switch. If later more features were implemented and that meant an increase of states DOD could be used instead of OOD. With OOD it is inflexible and using a switch can be untidy (Nystrom, 2014). Instead have a class for states with pointers to the components. The components can be store in an array that can be iterated over. Furthermore, it would be easy to get the components of an entity as it would only involve a pointer (Nystrom, 2014).

Design Oriented overall all has its benefits especially with memory caching and is something to be considered when constructing such a large application. However, if this was to be implemented within Tumble Chaps it would take a lot of time and a lot of overhauls, and only then would you see its true potential.

# Reflection

All in all, this task has been very invoking which I have learned a lot from. The module has been a great first experience for creating an application that supports console controls. Throughout the creation of Tumble Chaps, I understood box2d physics and gef classes more. The gradual process of my application coming together helped me with aspects of C++ that I wasn’t too confident with before, such as switch statements and passing objects around classes by reference. The steps I took when creating this application helped with structure and helped me set goals for each day, I worked on it. Firstly, the game states were worked on using switch statements every time the active state changed. If there was anything, I was to do differently next time it would be making use of a parent class with inheritance for the game states rather than using a switch statement. The next focus was loading in the assets within my level. Parts of this were a challenge but were overcome after looking closely into each step of my code. One main issue was the z-order, every so often I was finding my background not loading or the terrain however, after looking into z-order more it was a breeze. When it came to audio within my game this was done by creating an audio class in which a pointer to the audio manager was created. From there, audio could be initialised and played when I chose. The collision came as a bit of a challenge but for this, I maximised the use of the GameObject class creating subclasses of each object type I had made from there it was deciding how id set up each subclass. Next time I’d make the player my main collider rather than my ball simply for logical reasons. If I was to make further progress or content on my application, I would implement features like screen shake or a penalty for losing confidence to make the level that bit more exciting. Overall, the task at hand and module have been a great first experience at making a 2.5D game. The things I have learned or became more confident with over this module will most definitely help with my future work at Abertay University and is certainly a great turning point for somebody who isn’t as confident as they can be as a programmer.

# References

The following are references to any sources which were used when constructing this report and any sounds, models or music used within the application.

2021. *Alert, prompt, win, positive tone 001*. [online] Zapsplat. Available at: <https://www.zapsplat.com/music/alert-prompt-win-positive-tone-001/> [Accessed 14 May 2021].

2021. *Baseball land on grass 2*. [online] Zapsplat. Available at: <https://www.zapsplat.com/music/baseball-land-on-grass-2/> [Accessed 14 May 2021].

2021. *Button, click*. Zapsplat. Available at: <https://www.zapsplat.com/music/button-click/> [Accessed 14 May 2021].

fresh2refresh.com. n.d. *C++ Enumerations | Learn C++ Online | Fresh2Refresh.com*. [online] Available at: <https://fresh2refresh.com/cpp-tutorial/cpp-enumerations/> [Accessed 14 May 2021].

2021. *Designed mysterious warm musical twinkle 1*. Zapsplat. Available at: <https://www.zapsplat.com/music/designed-mysterious-warm-musical-twinkle-1/> [Accessed 14 May 2021].

Sketchfab. 2020. *Fall Guy Base Rig - Download Free 3D model by cwthomasart (@cwthomasart) [4f7e29e]*. [online] Available at: <https://sketchfab.com/3d-models/fall-guy-base-rig-4f7e29e10b6d420e9100ffe3f6ce731b> [Accessed 14 May 2021].

2021. *Game music - action, retro 8-bit style bouncy, hard dance track with electronic synths and drums*. [online] Zapsplat. Available at: <https://www.zapsplat.com/music/game-music-action-retro-8-bit-style-bouncy-hard-dance-track-with-electronic-synths-and-drums/> [Accessed 14 May 2021].

2021. *Game sound, chime, hit or collect item 2*. [online] Zapsplat. Available at: <https://www.zapsplat.com/music/game-sound-chime-hit-or-collect-item-2/> [Accessed 14 May 2021].

2021. *Game sound, fun arcade organ tone, short negative fail or lose tone 3*. [online] Zapsplat. Available at: <https://www.zapsplat.com/music/game-sound-fun-arcade-organ-tone-short-negative-fail-or-lose-tone-3/> [Accessed 14 May 2021].

2019. *Grass step*. [online] Available at: <https://freesound.org/people/Hawkeye\_Sprout/sounds/469155/> [Accessed 14 May 2021].

Mines, J., 2018. *Data-Oriented vs Object-Oriented Design*. [online] Medium. Available at: <https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056> [Accessed 14 May 2021].

Nystrom, R., 2014. *Data Locality · Optimization Patterns · Game Programming Patterns*. [online] Gameprogrammingpatterns.com. Available at: <http://gameprogrammingpatterns.com/data-locality.html> [Accessed 14 May 2021].

Cplusplus.com. n.d. *rand - C++ Reference*. [online] Available at: <https://www.cplusplus.com/reference/cstdlib/rand/> [Accessed 14 May 2021].

Sketchfab. 2017. *Tennis ball - Download Free 3D model by ingscappatura (@ingscappatura) [15bd608]*. [online] Available at: <https://sketchfab.com/3d-models/tennis-ball-15bd6087617a4a379953550c56edb839> [Accessed 14 May 2021].