­­­Computer Science Programming Project – PoolGame

*Internal Deadline: end of Autumn Term 2024 (Friday 20th December)*

Contents

[Notes 1](#_Toc179444788)

[Key 1](#_Toc179444789)

[Writeup 1](#_Toc179444790)

[Program 1](#_Toc179444791)

[Analysis of the problem 2](#_Toc179444792)

[Problem identification 2](#_Toc179444793)

[Stakeholders 2](#_Toc179444794)

[Research the problem 2](#_Toc179444795)

[Specify the proposed solution 6](#_Toc179444796)

[Design of the solution 9](#_Toc179444797)

[Decomposing the problem 10](#_Toc179444798)

[Describe the solution 10](#_Toc179444799)

[Describe the approach to testing 10](#_Toc179444800)

[Developing the solution 10](#_Toc179444801)

[Iterative development process 10](#_Toc179444802)

[Testing to inform development 10](#_Toc179444803)

[Evaluation 10](#_Toc179444804)

[Testing to inform evaluation 10](#_Toc179444805)

[Success of the solution 10](#_Toc179444806)

[Describe the final product 10](#_Toc179444807)

[Maintenance and development 10](#_Toc179444808)

[Bibliography 10](#_Toc179444809)

# Notes

## Key

Notes (to remove in the final version)

First draft (to expand upon)

## Writeup

* USE TABLES FOR THINGS
* Use Big O for measurements
* Identify success criteria that can be definitively tested and used to determine the success, or otherwise, in developing that feature of the solution
* Abstraction (easier to code, easier for CPU to run)
* Bibliography at end
* Add table of contents after it’s done
* Assign each success criteria a name, e.g.:

1. [placeholder]
2. [placeholder]

## Program

* USE ASSERTS
* Tag-in system to keep track of everyone’s scores even if they aren’t playing (e.g. at the start of each game, it asks the names of the players (allowing them to choose from a list of pre-existing names (maybe created at start-up) or make a new name))
* Tutorial
* Supports default controllers (Xbox, PlayStation) with a setting (maybe with a popup on the start screen) to add controllers by recording inputs and assigning them to certain controls
* Persistent data storage (maybe using .txt or .csv)
  + Settings
  + Scores
  + Different ‘user profiles’ have different saved settings
* Popup when the game is first installed (or when a new user enters) to enable various a11y features
* <https://prime31.github.io/Nez/docs/> for UI

# Analysis of the problem

## Problem identification

There currently are few group activities that are available to most people; most are either too expensive, slow to pick up on, or too repetitive. A solution to this problem could bring people together more (where other methods may be out of their reach) or just provide an easier and simpler way for groups to be entertained.

A computational approach is perfect for this problem as it can easily be customized to accommodate whoever’s using it, whereas analogue entertainment methods often can’t meet everyone’s needs (e.g. someone with mobility issues might have a hard time with sports, someone with very little money might not be able to afford a lot of entertainment methods that already exist).

My solution could be taken almost anywhere if it can run on a laptop, and there would be less risk of losing/damaging parts of it in transit. It’s also cheaper for end-users since they just need to install it rather than buying parts or paying fees, making it accessible to a wide range of people regardless of their financial situation (with some exceptions). In addition, my solution can be more scalable than a real-world game because no more equipment or space is needed for extra players (especially if only one keyboard and mouse/controller is needed).

## Stakeholders

### Casual gamers (***STK.1***)

These will likely be a large portion of my solution’s demographic, so they should be accommodated for as well; they range in age from children to adults. They play games to socialise and have fun, rather than to compete or challenge themselves, so my solution needs to be intuitive, relaxed, and have local multiplayer. Among them are some pre-teen children, so my solution will need to be suitably kid-friendly as well.

### Casual competitive gamers (***STK.2***)

These are still a large portion of the demographic so suitable mechanics should be introduced so that they can be entertained as well. They will need some sort of score-keeping system and a leader board to be implemented.

### Individuals with accessibility needs (***STK.3***)

A part of the problem my solution is trying to solve is the lack of availability of games to some groups of people, so my solution should be able to accommodate most their needs. They may need my solution to be able to run on low-end machines, they may also need colour-blind modes, etc.

### Notes

Due to the range of conflicting requirements, toggles may need to be available to alter how some mechanics work for different audiences.

I will be using ***STK.1***, ***STK.2***, and ***STK.3*** to refer to my stakeholders. If something is applicable to a particular stakeholder, I will write (***STK.x***) after it. (Later on, these will be replaced with ***SC.x*** for success criteria)

### Testers

My solution will be tested by its stakeholders where possible to reveal how suitable it is for them and for any other feedback they may have. Where this is not possible, I will be sure to research other solutions to problems with similar demographics to accommodate their needs. This will help me improve my solution by adding, tweaking, or removing features.

Some testers may have computer science experience and as such will be able to provide direct feedback on the source code. They may be able to evaluate my solution with a more technical lens, increasing the chance of finding bugs or exploits.

## Research the problem

[Add Images of each existing solution]

### Existing solutions

I will be looking at existing solutions to parts of my problem (or similar problems) and noting features to apply to my solution and ones to be mindful of (i.e. features that could be added but might not be suitable). If a similar feature is present multiple existing solutions, I’ll only write it down in the first game it appears to remain concise.

#### Gang Beasts

|  |  |
| --- | --- |
| Feature which could be applied to my solution | Justification |
| Simple menus and a “ready up” system | These ensure that players can start games quickly and when everyone is ready. The simple menus also make it easier for players to change settings and exit games, reducing the need for online guides and streamlining player’s experiences. (***STK.3***) |
| Simplistic environments and a minimalistic graphical user interface | These make it clearer for players to identify what they should be focusing on during a game, without being distracted by anything meaningless. (***STK.3***) |
| Environmental hazards | These create a more interesting and chaotic environment for players to engage with, reducing the boredom that comes from replaying the same thing over and over again. (***STK.1***)  They also provide different chances for players to combat each other in simple, unique ways without requiring new controls that may be hard to input and/or learn. (***STK.2***) |
| No inappropriate imagery | This makes the game suitable for all ages, allowing younger children to play. (***STK.1***) |
| Each player is assigned a colour and name and are allowed to pick an outfit for their character | This helps players avoid confusing who’s who mid-game even if they have some vision difficulties. (***STK.3***) |
| Using a keyboard, players can slow down or speed up the game with + or – respectively. | This make it more manageable for players with slower response times to be on an equal footing with faster players. (***STK.3***) |
| Both controller and keyboard/mouse support, with intuitive controls (e.g. space bar/A as jump and WASD/joysticks for movement) and an option to remap them in the settings. | This makes it easier for players to learn to play the game without needing time to remember specific controls. (***STK.1***)  While it may give less freedom to extreme competitive gamers (not casual competitive gamers), they aren’t a part of my stakeholders so I shouldn’t compromise the overall playability of my game for them. |
| Up to 4 players able to play at any one, with heavy interactivity between players and a score system. | This encourages players to be more competitive and obstruct each other, while keeping the playing field fair. (***STK.2***) |

|  |  |
| --- | --- |
| Features to be mindful of in my solution | Notes |
| It has simple and complicated combo moves | They are not explained to player very clearly, so most people either don’t know they exist or stumble into them accidentally. However, the game isn’t built off of these, so the simple moves are enough to play. |
| It revolves around fighting | It could be seen as unsuitable for some audiences. It skirts around this by using cartoon violence (i.e. no blood or gore) and it is presented in a light-hearted way. But players can still be ‘knocked out’ in more gory ways like by being pushed into a meat grinder or eaten by a shark. |

#### Jackbox Party Pack Series

|  |  |
| --- | --- |
| Feature which could be applied to my solution | Justification |
| No reliance on reaction time, only having a timer to make sure players submit their answers relatively quickly | This solves the problem of game speed having a greater impact on those who have lower reaction times. (***STK.3***)  It also bypasses the need for a game-speed modifier option, saving on development time and effort. |
| Each ‘Party Pack’ has different games to choose from when playing | This ensures a fresh experience for returning players, keeping them entertained even if they’ve played it before.  (***STK.1***, ***STK.2***) |
| Simple controls (via a mobile device) to submit answers with an on-screen mobile keyboard or the touch screen (e.g. for drawing) | This makes it easier for players to play the game even if they have limited gaming experience (since they only need to use their fingers to tap or draw). (***STK.1***)  It also reduces the need for players to re-map controls because they are so simple (although it could be troubling for players with impaired motor skills). (***STK.3***) |
| Option to suppress potentially inappropriate content | This allows much younger groups to avoid seeing any inappropriate content while still allowing older groups the freedom to have as much fun as they want. (***STK.1***) |
| Up to 8 players can play most of the games within each pack | This means that large groups don’t need to rotate out players. Less people are on the side-lines waiting for their turn, so more people can have fun together. (***STK.1***) |

|  |  |
| --- | --- |
| Features to be mindful of in my solution | Notes |
| A connection to the internet is required for players to submit their answers | With no in-person alternative due to the nature of the game, it makes playing it in-person require every player to have a mobile device. If players are in a place with no internet, it can’t be played (especially if some players can’t use mobile data). |

#### Golf With Your Friends

|  |  |
| --- | --- |
| Feature which could be applied to my solution | Justification |
| Builds off an existing and well-known game (crazy golf) | This means most new players are already familiar with the rules, so the time to learn how to play is reduced. (***STK.1***) |
| Players can interact with each other (by knocking into other player’s golf balls to send them off course). | This allows players to compete with each other and lets them actively disadvantage each other as well. (***STK.2***) |
| A player’s golf ball is always in view of them and is customisable, with colours, a name, and a trail. | This helps players avoid confusing who’s who mid-game even if they have some vision difficulties. (***STK.3***) |
| The game is scored in terms of how many shots it takes for a player to sink their golf ball | This gives players with slower reaction times an equal chance of winning since the time taken to finish isn’t taken in to account. (***STK.3***) |
| The settings menu has options to modify gameplay (e.g. toggle ball collisions, alter gravity) | This allows players to customize their experience to how they’d prefer for the most fun. It also keeps games fresh even after a few replays. (***STK.1***) |

|  |  |
| --- | --- |
| Features to be mindful of in my solution | Notes |
| Some courses can be quite challenging to complete (e.g. precise balancing or aiming) | Easier courses are available, so newer players or those with accessibility needs can still have fun while playing.  Another fix would be to have a sort of handicap/assistance mode, but it may be hard to tweak it so that it’s fair. |

#### Mario Kart 8 Deluxe

|  |  |
| --- | --- |
| Feature which could be applied to my solution | Justification |
| Assistance modes (e.g. auto-accelerate and smart steering) | This might help players who have accessibility needs keep up with other players. (***STK.3***)  It also levels the playing field for less-skilled players or those who are newer to the game, allowing them to play with more experienced players. (***STK.1***) |
| Item boxes | These add an element of randomness to the game and enable players who are behind to still have a chance to catch up (especially with items like the blue shell, which temporarily stuns the player in first place). Similar to Assistance modes, this shortens the gap in skill between players. (***STK.1***) |
| Difficulty options (e.g. item balancing and bot difficulty levels) | These allow for a more customisable experience and more fun for groups of players who aren’t looking for too hard of a challenge. (***STK.1***) |

|  |  |
| --- | --- |
| Features to be mindful of in my solution | Notes |
| It requires a Nintendo Switch to play | Most people don’t own a Switch, so it limits the number of people who can play the game. If it was available on multiple devices, it would be more accessible to more people. |
| It relies on reaction time | It may be challenging for those with certain physical disabilities or coordination difficulties to play on an even footing to those without. |
| The randomness of item boxes may seem unfair | This is particularly evident if a player is hit multiple times with detrimental items. It fixes this by varying the chance of certain items occurring. |

#### Minecraft: Bedrock Edition

|  |  |
| --- | --- |
| Feature which could be applied to my solution | Justification |
| Cross-platform multiplayer | This allows for more people to play together even if they don’t all have the same devices. They don’t need to buy expensive new hardware to play together, so the financial situation of prospective players matters less. (***STK.3***) |
| Variable graphics settings | These allow both for players with lower-end machines to play without issue and for those with higher-end machines to fully utilise theirs for improved graphics. (***STK.1***, ***STK.3***) |
| The menus are large and well-spaced with an easy-to-read default font | This makes them easy to navigate and understand without even needing to enable any specialised settings. (***STK.3***) |

|  |  |
| --- | --- |
| Features to be mindful of in my solution | Notes |
| It has a steep learning curve and is open-ended | This could be overwhelming for newer players, since they are thrust into a world with no clear objective.  Also, many of the mechanics aren’t explained in-game (e.g. potion brewing, finding the stronghold). While this doesn’t impact long-time players, newer players could miss out on a lot of what it has to offer. |

#### Pool (video game and analogue versions)

|  |  |
| --- | --- |
| Feature which could be applied to my solution | Justification |
| The objective and rules are simple and well-known, it also relies on real-world physics | This means that newer players are more likely to be able to pick up the game quickly, as I mentioned before. It also makes how objects move in-game more intuitive to learn. (***STK.1***) |
| Turn-based play | This allows players to take their time when planning their moves and removes any reliance on reaction time, meaning those with lower reaction times aren’t disadvantaged. (***STK.3***) |
| Players can disrupt each other’s strategies (e.g. by leaving the cue ball in hard-to-shoot-from locations at the end of their turn) | This adds to the strategy of the game and increases interactivity (and therefore compactivity) between players without direct player-verses-player combat. (***STK.2***) |

|  |  |
| --- | --- |
| Features to be mindful of in my solution | Notes |
| Its gameplay loop can feel repetitive, with little variation. | Variations of the classic eight-ball pool like one-pocket or nine-ball pool are meaningfully different, but they can still have the same premise and can feel repetitive if played enough. |
| It relies on player’s precision for some difficult shots | This may make it harder for less experienced players to hold their own against experts, reducing the availability for certain groups to play together. |

#### Accessibility Guidelines

While this isn’t an existing solution outright, it’s a guideline to one. I’ll keep it in mind when designing my game, so that it is built with accessibility in mind rather than having it patched on later (which could be difficult and/or time-consuming). For this I will be predominantly using [gameaccessibilityguidelines.com](https://gameaccessibilityguidelines.com/) because it lists a lot of features that I might not think of while designing. It separated features into categories (e.g. motor, hearing) and by how easy they are to implement.

On top of using that website, I’ll try to avoid needing to patch accessibility problems by not causing them in the first place. For example, instead of implementing a colour-blind mode, my game won’t rely on only colour for players to differentiate different objects from each other.

### Features of my proposed solution

My solution will be a simulated game of pool so that most new players can play without having to learn a new game. I have chosen pool because its premise is simple (i.e. hit more balls into pockets than your opponent to win) so it is easy to understand for new players. Due to its simplicity, it can be expanded upon to suit certain group’s preferences (e.g. adding obstacles, adding an ‘items’ equivalent, having more than one cue ball, etc.). Also, it will be a turn-based game, meaning that people with slow reaction times or other similar difficulties won’t be at a meaningful disadvantage while playing.

It will be able to be played by 2+ players at a time so that larger groups have less people watching on the side-lines being bored, and smaller groups can play too. A single player mode isn’t relevant to the stakeholders, so it isn’t likely to be added.

It will have simple menus to help players start games and find any relevant settings quickly. Where possible, there should be multiple ways for players to receive information and differentiate objects in the game (e.g. colours and patterns to differentiate between different object balls). There should also be controller support (on top of keyboard and mouse) because it helps people with needs re-map the controls to use with their specialised hardware.

#### Limitations

For larger groups, if it’s a turn-based game and people spend a while waiting for their turn. Or if not it’s a real-time game and, since it’s local multiplayer, it would have too many inputs into 1 device to function. It may work if online multiplayer was an option, but the main problem my game is trying to solve involves players being together in-person.

[placeholder]

## Specify the proposed solution

With the above idea of how to approach the problem in mind, I have outlined the requirements and features of my game. I have categorised the features to help keep in the scope of the project.

### Requirements

#### Functional

|  |  |  |
| --- | --- | --- |
| Requirement | Justification | Success criteria |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |

#### Non-functional

|  |  |  |
| --- | --- | --- |
| Requirement | Justification | Success criteria |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |

#### Hardware

|  |  |  |
| --- | --- | --- |
| Requirement | Justification | Success criteria |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |

#### Software

|  |  |  |
| --- | --- | --- |
| Requirement | Justification | Success criteria |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |
| Fill this in | Fill this in | Fill this in |

### Features

#### Essential (i.e. stakeholder requirements)

|  |  |
| --- | --- |
| Feature | Justification |
| Reasonably accurate simulated physics | If the game isn’t a realistic enough adaptation of analogue pool, it may feel uncomfortable or unintuitive to play (potentially putting off some players). If it’s overly realistic it will be slower to run on low-end machines, so it will be less accessible to people whose computers don’t meet the requirements.  This would exclude a section of my game’s stakeholders, so I will need to thread the line between overly realistic and not realistic enough. |
| Local multiplayer (2+ players, turn-based) | This allows gatherings to be fully entertained while playing my game, without people waiting on the side lines too long.  It should be noted that with a large number of players (4+), players may have to wait their turn anyway because of how long a round could get (but that’s better than them not being able to play at all). |
| A Score system | If the game is scored in-game, it allows ***STK.2*** players to compete without needing pen-and-paper scorekeeping. |
| Accessibility features | These will help those with specific needs have an easier and more fun time playing my game. Since people with accessibility needs are a portion of my stakeholders, these are high on my list. |

#### Desirable

|  |  |
| --- | --- |
| Feature | Justification |
| Controller support (on top of keyboard and mouse) | Controllers are preferred by a lot of STK.1 because they can be used from further away (e.g. on a sofa while the game is on a TV) and are easy to move around (when players are cycling who’s currently playing).  Also, some people have specialised controllers to help with accessibility needs. Adding controller support provides an easier way for these people to re-map the controls to their controller so they can play. |
| Control re-mapping | This enables the above feature (controller support) to be used more effectively by people with specialised hardware. It also allows anyone else who feels uncomfortable with the controls the ability to customise them to suit their needs. |
| Minimalistic/simplistic graphics (including menus) | Simple menus would help players change settings easier and start game sessions quicker, streamlining their experience. Simple graphics would make it easier for players to keep track of what’s important mid-game and for players with lower-end computers to play without performance issues. |
| Multiple/clear ways for players to receive information and differentiate objects in the game | This would reduce potential confusion for players, lessening the chance of mistakes made by players like potting opponent’s object balls. This issue would be exemplified in players with vision/reading difficulties, so having ‘back up’ methods for viewing objects. |
| For example:   * Colours and patterns to differentiate between different object balls * High contrast between borders |

#### Optional

|  |  |
| --- | --- |
| Feature | Justification |
| Bot players | While this feature isn’t a requirement from stakeholders, it could still make it nicer for smaller/one-person groups so that they can play with more ‘players’. |
| Difficulty options (if applicable) | This allows for a more customisable experience and more fun for players who aren’t looking for too hard of a challenge. But it can only be implemented if there are already features that can be tweaked (e.g. item boxes or bots). |
| Options for different game modes / toggleable features | This is a way of adding some variety to my game. If players either don’t find pool fun or have played my game too much they could still hopefully find some enjoyment in it. |
| For example:   * Enabling environmental hazards (allowing for more ways for players to make use of / be challenged by their environment without requiring any new controls to learn) * Enabling item boxes (or equivalent) so that players who are behind have a chance to catch up.   Multiple cue balls, enabling interactivity between players (by knocking into other player’s cue balls to send them off course). |

#### Out-of-scope/Excluded

|  |  |
| --- | --- |
| Feature | Justification |
| Online multiplayer | The main focus of the game is for local multiplayer, so it wouldn’t meaningfully benefit the majority of the stakeholders. Also, online multiplayer can be difficult and time-consuming to implement. It would take up a lot of development time for a feature that most people wouldn’t use. |
| Multiple platforms | This amount of development would take a lot more time to implement than I have available. My game will be limited to an application that can be installed on PCs and other similar hardware. |
| Inappropriate imagery, or wording | My game must be suitable for as wide of an age range as possible, so anything that isn’t suitable for young children shouldn’t be added. |

“As part of the analysis candidates are expected to identify suitable measurable success criteria. In many cases the success criteria identified are far too generic often referring to ease of use or aesthetic considerations. The success criteria will be used together with evidence of testing to establish the effectiveness of the solution. It is important the student identifies success criteria that can be definitively tested and used to determine the success, or otherwise, in developing that feature of the solution. The table above is a part of the success criteria for a game. The student identifies a measurable feature, establishes how that feature can be tested and provides justification for that feature as part of the success criteria.”

Limitations of mobile apps: hard for many people to use/see at one time

# Design of the solution

The problem outlined in the Analysis section should be solved/helped by my solution (my game). I have outlined the structure of the solution (the menu layout, gameplay loop, etc.), so I will go into more detail on specific sections here, relating more to the implementation through programming.

I’ll use C# to develop my game because I have the most experience with it and it has a good reputation for game development. I will also be using the MonoGame framework; it seems intuitive to learn and doesn’t have as many built-in features as Unity. This is ideal since my goal for this project is to gain more experience with game development, so building more features myself would be ideal (and less confusing). The integrated development environment I’ll be working in is Visual Studio 2022 because it allows me to initialise C# programs easier than VSCode and was built with .NET in mind, so development should be a more streamlined process than it would be using VSCode.

## The rules

Since there are many different varieties of analogue pool, I will be basing the rules of my game on the World Pool Association’s rules (although some things like playing out of turn or jump shots will be impossible in a simulated environment, so they don’t need to be included). I will create simplified rules to make them easier to understand, so they may differ from the WPA’s exact rules.

### Definitions

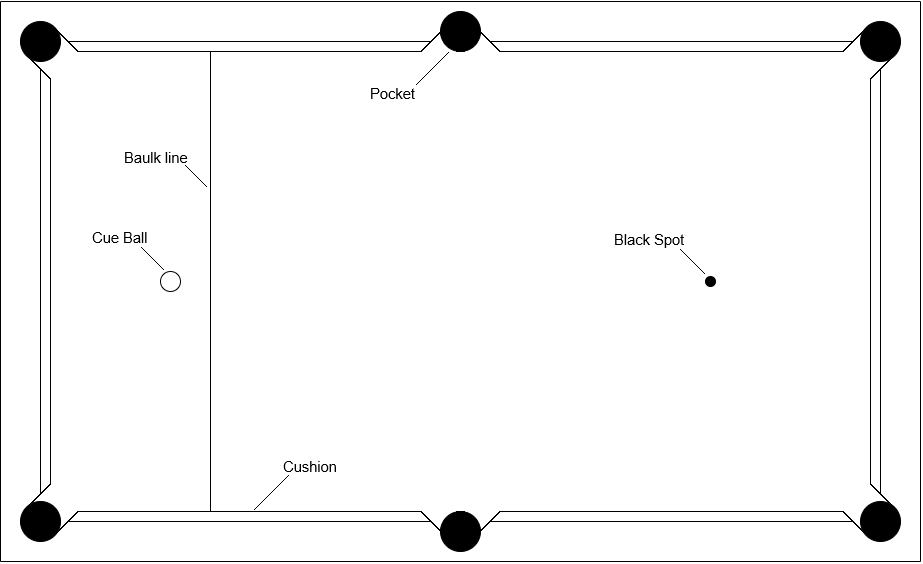
Objects

Table – the plane that pool is played on, [in real life its dimensions are about 213 cm by 122cm] (aspect ratio 7:4)

Pool ball – ball that players either shoot with or at, in real life its diameter is about 5cm

Cue ball – the pool ball that the players shoot with

Object ball – a pool ball that is hit by other balls

Coloured ball – an object ball that isn’t the eight-ball

Striped/solid – the striped and solid coloured balls respectively

Eight-ball – the object ball that isn’t a coloured ball

Baulk line – the line where the cue ball starts in the middle of, it lies 1/5th the way across the table

Cushion – lining around the table that pool balls bounce off

Pocket – holes that consume pool balls that collide with them, in real life its diameter is about 9 cm

Black spot – a mark that shows where the eight-ball is placed before a break, it lies 1/5th the way across the table

#### Terms

Pot [a ball] – when a pool ball goes into a pocket

Scratch – when a cue ball is potted

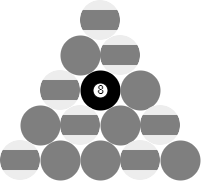
Triangle – the collection of object balls at the start of the match

Break – the first shot of the match, before the triangle is disturbed

Standard foul – an illegal move that doesn’t result in a loss for the player who committed it

Loss of match foul – an illegal move that results in a loss for the player who committed it

### Rules

* The match is split into turns. Each player’s turn begins when the previous player’s turn is over and no balls are moving on the table. Each player’s turn ends when they have shot their final shot on their turn.
* If a player pots one of their coloured balls on their turn, they gain another shot. A player can only gain one shot per shot they make (meaning they can’t get two extra shots for potting two balls in one shot).
* A player wins the match by potting all of their object balls and then the eight-ball.
* If a player has committed a standard foul, the next player can choose to place the cue ball anywhere on the table (except on top of other pool balls or in pockets). Committing a foul also negates any extra turns gained by potting object balls. A player has committed a foul if, on their turn:
  + The cue ball is potted
  + An opponent’s object ball is potted
  + The first object ball hit is an opponent’s object ball
  + The first object ball hit is the eight-ball (while at least one of their coloured balls is still in play)
  + No object balls are hit
* If a player has committed a loss of match foul if they:
  + Commit a foul during the same shot that the eight-ball is potted
  + Pot the eight-ball while at least one of their coloured balls is still in play
  + Pot the eight-ball and their last coloured ball in the same shot
* At the start of each match, 7 coloured balls of each colour are arranged in a triangle with the eight-ball in the centre (placed on the black spot). They are arranged as such, with the top of the triangle pointing towards the baulk line:  
  
* It is randomly decided who plays first each match. But if matches are played one after the other, the player who played first last match cannot play first. If there are more than two players, a player cannot play twice before another player has played once.
* The type of a player’s coloured balls is decided by the first player to hit an object ball; the colour of object ball they hit is their colour for the rest of the match. (Note that while typical eight-ball pool uses yellow and red coloured balls, I have chosen to use striped and solid ones to help people with visual impairments tell them apart.)

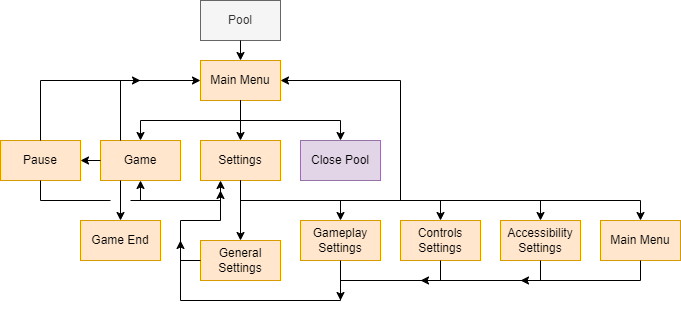
## Decompose the problem

I’ve broken down my game into different components that can be brought together to form the game. Each component is associated with a screen in the game and has the ability to handle the underlying logic itself. They should be structured so that they are all connected (directly or indirectly) so that each of them is quick and easy to get to for a streamlined experience.

Splitting up different screens also ensures that processes (e.g. collision detection) aren’t run and objects aren’t rendered pointlessly while in another screen, which will improve overall performance.

The aspect ratio of all screens will be 16:9 because that is what the vast majority of device screens are. While a variable aspect ratio would be ideal, the pool table must always have a length-width ratio of 2:1 and so other things must be laid out around it to fit with that. If the aspect ratio was variable and if the width is extended too much, it would squash all of the peripheral parts of the graphical user interface (e.g. score or current player). To streamline this, users will get to pick one of a few different resolutions (e.g. 1920x1080, 1280x720, 854x480).

Overall, the connections between screens should look something like this:



[change Game to Match]

Orange rectangles represent screens, arrows represent the screens that can be accessed from a screen. The diagrams for individual screens will only have the screens they lead to, not the ones that lead to them, because otherwise it would look too cluttered.

### Menus

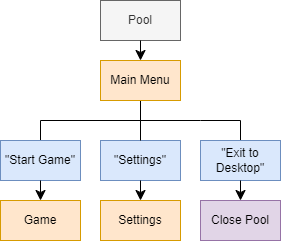
Most of the screens will be menus, mainly featuring buttons, dropdown menus, and/or sliders. Buttons will be used to move between screens (e.g. buttons labelled “Settings” go to the Settings screen); they are intuitive to use and, when labelled, are clear what they do. Dropdown menus will be used for changing discrete settings (e.g. controls or toggles); they let players view all of the options rather than cycling through them one at a time, making it easier to find specific setting options. Sliders will be used to change continuous settings (e.g. sound-effect volume) from 0% to 100%; they allow for precise customisation over specific settings without having vague labels (e.g. “low” or “medium”) while also not needing a separate button for mute (since players can just set the slider to 0%).

The layout of menu screens will be quite plain, with only the above features (except the Match End screen, which displays more information). This makes it clear what the players should be focusing on in each menu, making it easier to find the settings they’re looking for.

#### Main Menu

The Main Menu screen is the screen that will initially be shown when my game is opened, it is linked to by any button labelled “Exit to Main Menu”. Its main function is to be a hub for most of the other screens, while allowing users to exit the program. There will be 3 buttons, each button’s destination screen will be given in brackets (if applicable):

* “Start Match” (Match) with default [or previously-saved] gameplay settings
* “Settings” (Settings)
* “Exit to Desktop” – closes the program, requires a confirmation to prevent accidental exits.



[change Game to Match]

Blue rectangles represent buttons, with their display text in quotes.

###### Justification

The Main Menu should be displayed when a match starts so that players can access settings before they start a match. This should allow players to modify settings to enhance their experience as soon as they boot up the game if they need.

Because of the Main Menu’s utility, it is linked to by every screen (except Settings’ sub-screens) so that it can be accessed easily.

It should be noted that because closing the program and exiting a match in my game could both come under the term “Exit Game” (despite being different), so “Exit to Desktop” will be used to close the program and “Exit to Main Menu” will be used to exit a match and go to Main Menu.

#### Settings

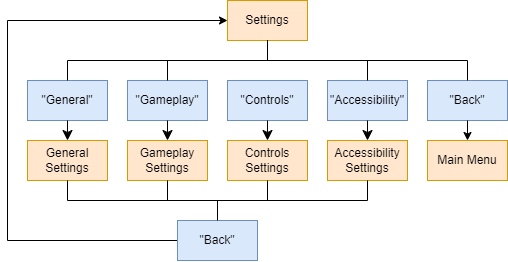
The Settings screen is linked to by any button labelled “Settings” and is much like the Main Menu screen in that both are hubs to reach other screens. This screen will have 5 buttons, each leading to their own sub-menu:

* “General” (General Settings)
  + Volume (slider) – since there are no music or voices, only one slider is needed
  + Resolution (dropdown) – only certain values for resolution will be supported for reasons noted above
  + Fullscreen/Windowed Mode (button) – [placeholder]
  + [TBD]
* “Gameplay” (Gameplay Settings) [– if the Settings screen is opened during a match, this menu’s settings (or at least some of them) will not be changeable.]
  + [TBD]
* “Controls” (Controls Settings)
  + [TBD]
* “Accessibility” (Accessibility Settings)
  + [TBD]
* “Back” (Main Menu) – in Settings’ sub-menus, it will be used to go back to Settings

###### Justification

This screen gives users the ability to customise their experience. It’s separated into different menus so that users can quickly find the settings they are looking for, instead of having to search through a big list.

[Gameplay Settings’ settings will not be fully changeable mid-match because it could lead to some game-breaking effects.]



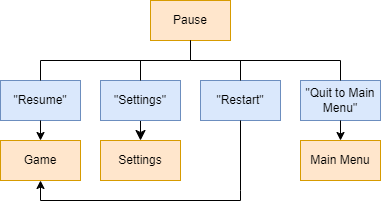
#### Pause

The Pause screen will popup when a certain key is pressed during a match. It will have the following buttons:

* “Resume” (Match) – Resumes the current match
* “Settings” (Settings)
* “Restart” (Match) – Initialises a new match, requires a confirmation to prevent accidental restarts
* “Exit to Main Menu” (Main Menu) – requires a confirmation to prevent accidental exits

###### Justification

The pause screen will allow players to pause the match if they need to take a break or do something else. It will also let players change settings mid-match to test them out or correct them without restarting a match and they can exit or restart matches smoothly.



[change Game to Match]

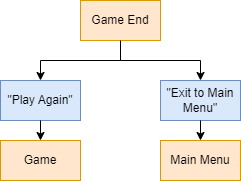
#### Match End

The Match End screen will only appear when a match is over (therefore, it can only be reached through the Match screen). It will display the scores of the players and announce a winner. It will have the following buttons:

* “Play Again” (Match) – Re-loads the Match screen with the same setup as before.
* “Exit to Main Menu” (Main Menu) – does not require a confirmation because no data can be lost when a match is already over.

###### Justification

The Match End screen serves as a transition screen between Match and Main Menu (or Match and Match again if players choose to play again). It allows players to instantly play again if they want, rather than having to go back to Main Menu, minimising repetitive button presses if a few matches have been played.



[change Game to Match]

### Match

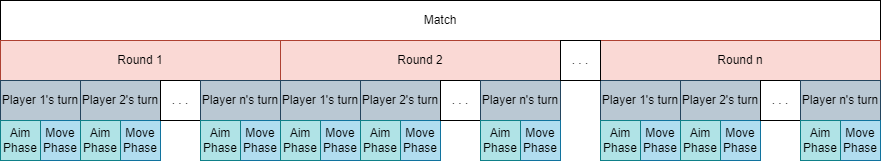
The Match screen holds most of the substance of my game, it displays matches of pool. Scores are reset to zero when a match is restarted or ended, not when the Match screen is loaded, to not reset scores every time the pause button is pressed.

Here’s an example of what the screen might look like mid-match:

[Example here]

#### Turn order

To clear up some terminology: each match encompasses ‘n’ rounds (where ‘n’ rounds are played before the match ends), each round encompasses ‘m’ turns (where ‘m’ is the number of players), each turn encompasses two phases (the Aim and Move phases). This is illustrated below (where it follows from 1 to ‘m’ turns in 1 to ‘n’ rounds):



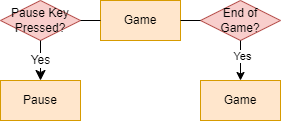
When any round starts, player 1’s turn begins and they enter the Aim phase. In this phase, they aim and fire their cue ball. Aiming the ball should feel as close to analogue pool as possible (i.e. pick a direction and an intensity for each shot), they should be able to click a point on the board and the ball should move to that point on the Move phase. Its initial speed being changed depending on how far back they choose a second point to be, as illustrated below:

[Example here]

After player 1 aims and sets the initial velocity, the Move phase begins in which, the cue ball moves according to the Aim phase. Any collisions between balls or between a ball and the cushions will be calculated and the balls’ velocities will be altered accordingly (i.e. they will bounce if they hit something).

###### Justification

Each turn is split into two phases to avoid it checking for collisions or handling similar logic when in the aim phase to save memory. The aim and move phases are both part of the same screen because the display doesn’t radically change between the two; also, it’s simpler to initialise all of the objects at the start of a match (rather than re-initialising them every move phase).

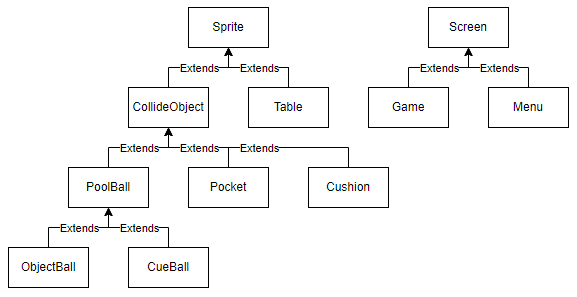


[change Game to Match]

Red diamonds represent decisions, with their outcomes leading off as arrows.

## Describe the solution

Continuing from the layout of my game outlined above, here is a simple initial class diagram:



[change Game to Match, add other classes in list below]

I will be using “Subclass : Superclass” to indicate when certain classes inherit from a superclass.

Most of the classes are sub-classes of Sprite because the main portion of the game revolves around the various pool balls, pockets, cushions, and the table. The former three come under CollideObject, a subclass of sprite which handles collisions. Cushions and pockets are separate from table because they are the only parts with collisions on it and they both interact with pool balls in different ways.

[I will be using pre-existing code to add most GUI elements because the main focus of this project is pool, not menu design.]

MonoGame uses Update() and Draw() methods inside of a Game1() class. Some of my classes will have their own versions of those methods that can be placed inside of Game1()’s. This should make Game1()’s file simpler and therefore easier to read.

### Screen

Screen is a superclass for Match and Menu, it will display the background for each screen. Each instance of Screen will have a collection of relevant Sprites (and Sprite’s subclasses) which will only be initialised/active when their associated screen is displayed.

#### Justification

Screen provides a stable baseline for Match and Menu, handling the things they have in common to prevent those classes from repeating too much code and making them easier to setup.

Each Screen has a collection of relevant Sprites so that it can always initialise in the same way and save data when, for example, a match is paused and resumed (going from Match to a Menu and back to Match).

|  |
| --- |
| Screen |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### Match

Match is the Screen that displays a match of pool. It holds all of the PoolBalls, Pockets, Cushions, and the Table for each match. It also keeps track of score and turn order, switching to the Match End screen (an instance of Menu) when a match is over.

#### Justification

Match is unique in that it is the only Screen that has CollideObjects and doesn’t have any Buttons, Sliders, and Dropdowns. So, setting it up to accommodate them is unnecessary.

Also, Match needs to keep track of different information than menu, only being saved for the duration of a match instead of the entire time the game is open [or after it’s closed].

|  |
| --- |
| Match : Screen |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### Menu

Menus are all of the Screens that aren’t Match. They will only feature Buttons, Sliders, and Dropdowns (with some only featuring Buttons), no CollideObjects.

They will also be able to store changed settings (if applicable) [even after the game is closed].

#### Justification

Since most of the Screens in my game are similar (e.g. just have Buttons, Sliders, and Dropdowns), having a Menu class prevents the need to setup lots of instances of Screen with repeated code. Instead, Buttons, Sliders, and Dropdowns will be built-in and therefore easy to add, configure, and position for different instances of Menu.

|  |
| --- |
| Menu : Screen |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### Sprite

Sprite is the superclass for most of the things the players interact with and see. It handles textures and positions. Also, the (0,0) point (or origin point) will be the top left corner for all sprites.

#### Justification

Since all of Sprite’s subclasses have both a position and a texture, Sprite handles that simple aspect for all of them. Rather than each of them needing to implement their own methods, Sprite keeps it all consistent and can be overrode if necessary.

The (0,0) point of all Sprite objects will be the top left corner because that is the default value. While having the origin in the centre would make it easier to find points on the circumference of a circle (and thus handle circle collisions), that can easily be adjusted by adding its radius to its x and y coordinated to find the centre. Also, since screen coordinates have the origin at the top left, this approach makes origin points consistent across all images.

|  |
| --- |
| Sprite |
| - texture: Texture2D  - position: Vector2D |
| + Update(gameTime)  + Draw(spriteBatch) |

[placeholder description, explanation, and justification]

I have chosen to use Texture2D as the

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### InputWidget : Sprite

InputWidget encompasses all Buttons, Sliders, and Dropdowns that players can interact with (defining which one a given InputWidget is with an [enum]). Players can interact with them with their mouse, keyboard, or controller.

Buttons are rectangles, often labelled, that can be clicked/pressed. They are used to go between Screens or toggle some binary settings.

Sliders are horizontal lines, each with a rectangle that users can drag (if using a mouse) or shift incrementally (if using a controller or keyboard) left and right along the line. They are used to change some continuous settings.

Dropdowns are similar to buttons but display a dropdown menu when clicked (and hide the menu when clicked again). The dropdown will have several options for discrete settings that players can see all at once (rather than cycling through via a button or slider).

#### Justification

There are three types of InputWidgets because they each serve a different functionality (as explained). Each type’s characteristics provide an easier and more intuitive way for players to change settings. Buttons are simple to use, dropdowns show every option clearly, and sliders are intuitive to use. For example, changing volume with a button would take forever or just be impossible.

All input widgets are in the same class because they all have similar functionality (i.e. react to being pressed/slid).

It also allows me to have an array of InputWidgets for any page that will encompass all types, instead of, for example, needing to define different arrays for buttons and sliders on a screen.

### CollideObject

CollideObject encompasses all circles, rectangles, and lines that can collide with each other (switching between types with an [enum], like InputWidget). [All collisions will be assumed to be perfectly elastic (meaning more or less that their speed doesn’t change)].

CollideObject will have its own methods for circle-circle, circle-rectangle, circle-line, rectangle-rectangle, rectangle-line, and line-line collisions.

Lines can be vertical, horizontal, 45 degrees left of the horizontal, or 45 degrees right of the horizontal.

The three types of CollideObjects will be used in the Match screen in the following ways: circles will be used for all pool balls and pockets, rectangles will be used for [placeholder reason], and lines will be used for the cushions around the edges of the table.

[Cushions will only detect collisions with other objects when they are in the same portion of the board as (or close to) each other].

#### Justification

An [enum] is used rather than splitting the types into subclasses so that all of the collision methods only have to be written in one class. For example, this prevents the need for both rectangle-circle and circle-rectangle collision to exist, since both have the same logic and can be handled with one method.

[Collisions are elastic because their real-life loss of speed isn’t very impactful (especially when friction already slows them down). This mechanic will also save on computational power needed, reducing the chances of a crash if too many pool balls collide at once (like during the break at the start of the game).]

Each collision method will only change attributes of the instance of CollideObject that it is in to prevent attributes being changed twice when two objects collide.

Only those specific line types are needed since that is what the cushions are made up of. Adding a general line, θ degrees from the horizontal, would be more complicated and ultimately unneeded.

[The range limitation for collision detection should increase performance, as it stops everything from always needing to check for collisions.]

|  |
| --- |
| CollideObject : Sprite |
| - objectType: enum  - radius: float  - boundingBox: Rectangle  [placeholder line collision thing (45 degrees)] |
| [placeholder] |

[placeholder description, explanation, and justification]

I’m using a variable for the type of object so that objects can be compared efficiently without needing separate classes. If they were in separate classes, some code may be redundant (i.e a circle-rectangle collision in the circle class and a rectangle-circle collision in the rectangle class would both do the same thing).

For a circle, its radius is given in their constructor (as well as the other attributes from its superclass, Sprite).

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### Pocket

Pockets are circle-type CollideObjects that initialise at the start of a match and are unmovable. They will remove any pool balls that come into contact with them. [This will then be detected by Game, which will update scores and replace pool balls as needed.]

#### Justification

Pockets are a core part of pool, with a unique function compared to other objects. As such, they need their own logic and therefore their own class.

They remove any pool balls that hit them so that Game, which holds a list off all pool balls, can know and update the scores. It also removes the need to simulate these now-irrelevant balls.

|  |
| --- |
| Pocket : CollideObject |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### Cushion

Cushions are a collection of line-type CollideObjects that initialise at the start of a match and are unmovable. They keep all the pool balls in bounds. [An enum will be used to tell what type of cushion is being placed (e.g. end cushion or side cushion).]

This method will mean that an option to rotate the cushion type 180 degrees is required, since, while they have the same dimensions, opposite cushions are flipped.

#### Justification

Having cushions as their own class (instead of just multiple different instances of CollideObjects) makes it simpler to set up the table each time.

[All of the side cushions have the same dimensions and both of the end cushions have the same dimensions, so there can be two generalised cushion types. This will make it easier to set up the table at the start of a match.]

|  |
| --- |
| Cushion : CollideObject |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

### PoolBall

PoolBall will encompass both ObjectBall and CueBall. They are circle-type CollideObjects that initialise at the start of a match and move around in response to players’ actions.

They will have a deceleration due to friction that will slow them down until they stop.

They will also either need a Boolean attribute that defines whether they have been potted or not, or at least a method that will delete them if they are (however that may cause issues, for example with arrays of a fixed size).

#### Justification

Since cue balls and object balls are so similar (despite serving different purposes), PoolBall covers their similarities such as collisions and speed. This reduces repeated code, reducing the change for mistakes when copying and potential bug-causing differences.

While their friction makes their movements more realistic, it also importantly stops them from moving forever.

|  |
| --- |
| PoolBall : CollideObject |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### ObjectBall

ObjectBalls are PoolBalls that cannot be directly controlled by players. They function almost the same as PoolBalls, but they are also assigned a type (either solid or striped).

#### Justification

Each ObjectBall is assigned a type so that players can tell them apart and scores can be kept track of by counting the number of potted ObjectBalls.

|  |
| --- |
| ObjectBall : PoolBall |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

Enum for solid/striped/eight-ball

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### CueBall

[placeholder]

#### Justification

[placeholder]

|  |
| --- |
| CueBall : PoolBall |
| [placeholder] |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

### Table

[placeholder]

#### Justification

[placeholder]

[Is table needed?]

|  |
| --- |
| Table : Sprite |
| - objectBalls: List<ObjectBall> |
| [placeholder] |

[placeholder description, explanation, and justification]

|  |  |  |
| --- | --- | --- |
| Attribute | Explanation | Justification |
| AttributeName | [placeholder] | [placeholder] |
| AttributeName | [placeholder] | [placeholder] |

|  |  |  |
| --- | --- | --- |
| Method | Explanation | Justification |
| MethodName | [placeholder] | [placeholder] |
| MethodName | [placeholder] | [placeholder] |

ADD:

* Classes:
  + Button
  + Slider
  + Dropdown

“Break down the problem into smaller parts suitable for computational solutions justifying any decisions made.”

* Explanation of all the game objects.
* Algorithms in pseudocode for the methods in the scripts.
* Usability features.

(a) Explain and justify the structure of the solution.

(b) Describe the parts of the solution using algorithms justifying how these algorithms form a complete solution to the problem.

(c) Describe usability features to be included in the solution.

(d) Identify key variables / data structures / classes justifying choices and any necessary validation.

Abstraction

* assuming all collisions are elastic
* velocity treated as momentum since all balls have a mass of 1

player attributes:

* pool ball colour

prevent balls from touching at the end of the move phase to stop instant fouls

|  |  |  |
| --- | --- | --- |
| Computational Method | Where it can be used | Justification |
| [placeholder] | [placeholder] | [placeholder] |
| [placeholder] | [placeholder] | [placeholder] |

.NET is intermediate code

* A structure diagram illustrating the problem decomposition.
* Explanation of all the game objects.
* Algorithms in pseudocode for the methods in the scripts.
* Usability features.
* Data structures to be used.
* Input validation.

1. Explain and justify the structure of the solution.
2. Describe the parts of the solution using algorithms justifying how these algorithms form a complete solution to the problem.
3. Describe usability features to be included in the solution.
4. Identify key variables / data structures / classes justifying choices and any necessary validation.

## Describe the approach to testing

* Input validation.
* Test data to be used during the development of the coded solution.
* Test data to be used post-development.

1. Identify the test data to be used during the iterative development and post development phases and justify the choice of this test data.

# Developing the solution

[mention use of polymorphism when iterating through poolBalls]

[credit monogame docs for descriptions of Initialize(), draw(), etc]

## MonoGame logic

By default, MonoGame sets up an initial Game1.cs file with a Game1 class (which itself is a subclass of Game, a class that handles some behind-the-scenes logic) where most of the boilerplate logic of the game should be and a Program.cs file to create an instance of Game1 and run it. The game loop is separated into different methods of Game1: Game1 (the constructor), Initialize, LoadContent, Update, and Draw.

### Gameplay loop

When the program is run, an instance of Game and an instance of Game1 are created and Run is called on the instance of Game1 which calls Initialize. Initialize handles some of the initialisation logic and calls LoadContent. Then Update is called once and the game loop starts.

The loop starts by Update being called which handles most of the logic for objects in-game such as updating variables, calculating hitboxes, etc. After that Draw is called which renders any sprites. Then the loop repeats.

### Update and Draw

(Since the both of these methods are for various different objects, I will refer to each with [object].method to clear up potential confusion.)

Some rendered Sprites need their own Sprite.Update method so that Game1.Update can call them each game loop so that all objects are successfully updated. Sprite.Update in turn houses the update logic for its Sprite for ease of use. A Game1.Update method might look something like this:

public override void Update(GameTime gameTime)

{

Base.Update(gameTime); // calling Game’s Update method

int exampleNumber = 1;

int exampleWord = “Hello, world”;

ExampleMethod1(exampleNumber);

ExampleMethod2(exampleWord);

}

All rendered Sprites also need their own Sprite.Draw method so that Game1.Draw can call them when rendering a SpriteBatch (a collection of sprites to be rendered). Sprite.Draw takes a SpriteBatch as a parameter and, in turn, that SpriteBatch has a SpriteBatch.Draw method that takes in various parameters (e.g. texture, position, origin, scale) and renders the sprite. A Game1.Draw method might look something like this:

public override void Draw(GameTime gameTime)

{

Base.Draw(gameTime); // calling Game’s Draw method

\_spriteBatch.Begin();

exampleSprite1.Draw(\_spriteBatch);

exampleSprite2.Draw(\_spriteBatch);

exampleSprite3.Draw(\_spriteBatch);

\_spriteBatch.End();

}

## Iterative development process

I’ve split the development into [n] iterations, implemented via code. The specifics of each iteration may stray from its initial purpose, but that is to be expected since some things I didn’t account for may happen or I may have new ideas mid-way through coding. The iterations are split as follows:

1. Setting up. This iteration encompasses creating both an environment and a cue ball which moves around in that environment with reasonably accurate physics. The cue ball should be able to be shot by the player (who can input a direction and magnitude of the initial velocity), move around the screen, bounce off the edges of the screen, and slow down due to friction. Despite its simplicity, this iteration will provide a lot of the groundwork on which the later iterations are built on, making it important for it to be both as simple and as developed as possible.
2. Expanding. This iteration primarily focuses on adding a lot of features to make the game resemble a game of pool. One feature will be developed at a time to reduce potential conflicts if they merge incorrectly and to ensure that the game is fully functioning after each addition.
3. Finishing. This final iteration will resemble the solution described throughout the ‘[Design of the solution](#_Design_of_the)’ section, adhering to all of the requirements laid out there. It will also potentially include things I didn’t think of during that stage that would bring it closer to solving my initial problem. It is then ready for post-development evaluation.

### 1st iteration

To begin, I need to setup a basic environment and implement some physics so that I can later implement, test, and tweak the logic in my game as needed.

I started by implementing a table rim (mine was essentially just a 2x1 rectangle) to get the hang of the framework and because I would eventually need one.

I experimented with using a scale so that the image could be enlarged/reduces to fit the window, but eventually got rid of it because enlarging PNGs proved to be awkward (even with the scale parameter in Sprite.Draw). This was mainly because when a low-resolution sprite is scaled up, its individual pixels become more noticeable and if multiple different sprites are scaled up different amounts, the size difference of pixels is jarring.

I used tableRimTexture = Content.Load<Texture2D>("2-1 rectangle (transparent)"); with an image I’d made to represent the rim and rendered it in the window, looking like this:

[example here]

Once that was implemented, I moved onto rendering a pool ball. I started by creating a folder titled ‘Classes’ that would house the .cs files that had individual classes in them so that they could be viewed and edited easier. The subclasses of a class would then be stored in a folder titled ‘[ClassName] inheritors’ so that I would essentially have a class diagram in my solution’s file directory. For example, one branch of the folder tree might look like this:

Classes/

├── Sprite.cs

└── Sprite Inheritors/

├── CollideObject.cs

└── CollideObject Inheritors/

└── PoolBall.cs

Following my class diagram from the Design phase, I made a basic sprite class in Sprite.cs that stored a sprite’s texture as a Texture2D and position as a Vector2 (with position representing the coordinates of the centre of the object). Apart from a constructor, it only needed a Draw method:

public virtual void Draw(SpriteBatch spriteBatch)

{

spriteBatch.Draw(

texture,

position,

null,

Color.White,

0f,

new Vector2(texture.Width / 2, texture.Height / 2),

Vector2.One,

SpriteEffects.None,

0f

);

}

Most of the parameters in the spriteBatch.Draw method don’t alter anything (apart from ‘texture’, ‘position’, and ‘new Vector2(…)’), but this particular override of the Draw method is needed in order to specify where the centre of the Sprite is (in this case, it’s represented by: new Vector2(texture.Width / 2, texture.Height / 2).

I then made a CollideObject class that featured an enum called ObjectType (either Rectangle or Circle) that would let collision detectors know which type it was when calculating collisions. Originally, there was two constructors, utilising whether a Rectangle (the bounding box) or an int (the radius) was passed in to determine which of the two types the particular instance was (then using Type = ObjectType.Rectangle or Type = ObjectType.Circle to set the attribute during construction). Later, however, I realised that there were no rectangles that could be collided with in my game (since the cushions were better modelled as individual trapeziums), so I removed the Rectangle ObjectType.

I left the Update method empty for now so that I could focus on developing the movement of a pool ball first (as explained above), but creating the CollideObject class was still necessary so that PoolBall could inherit it later without me having to define a class after its own subclass (which could get messy).

For the PoolBall class, I defined velocity and acceleration as Vector2s. All PoolBalls have been abstracted to have a mass of 1 unit, since they all weight the same in analogue pool and having a mass of 1 makes velocity equal to momentum and acceleration equal to force (greatly reducing the number of attributes each PoolBall needs). However, the only force acting on a PoolBall is friction (ignoring air resistance, which has been abstracted) since velocity is only changed by friction or when a collision occurs (in which case velocities are changed instantly, so acceleration isn’t needed). So I re-named acceleration to decelerationDueToFriction later on to plainly layout what it is.

After defining velocity, I created a simple ChangePosition method to be called inside of PoolBall’s Update method (which, as mentioned above, is called by Game1’s Update method each frame).

public void ChangePosition()

{

position += velocity;

}

I also defined a DoBoundsCollision method to be called inside of PoolBall’s Update method. It reflects a PoolBall when it reaches the edges of the screen to keep it in bounds. It also changes the PoolBall’s position appropriately to place it outside of the bounds. This prevents repeated velocity changes and PoolBalls escaping if a one gets partially outside of the frame (which is possible when velocity is large enough).

public void DoBoundsCollision()

{

// with top:

if (position.Y - radius < 0)

{

position = new Vector2(position.X, radius); // keeping in bounds if it clips out

velocity = new Vector2(velocity.X, -velocity.Y); // reversing part of it to give the effect of an elastic collision

}

// with bottom:

if (position.Y + radius > Game1.windowHeight)

{

position = position = new Vector2(position.X, Game1.windowHeight - radius); // keeping in bounds if it clips out

velocity = new Vector2(velocity.X, -velocity.Y); // reversing part of it to give the effect of an elastic collision

}

// with left:

if (position.X - radius < 0)

{

position = new Vector2(radius, position.Y); // keeping in bounds if it clips out

velocity = new Vector2(-velocity.X, velocity.Y); // reversing part of it to give the effect of an elastic collision

}

// with right:

if (position.X + radius > Game1.windowWidth)

{

position = new Vector2(Game1.windowWidth - radius, position.Y); // keeping in bounds if it clips out

velocity = new Vector2(-velocity.X, velocity.Y); // reversing part of it to give the effect of an elastic collision

}

}

Moving onto the CueBall class, it inherits from PoolBall but also has a Shoot method. This takes in a mouse position (given by new Vector2(Mouse.GetState().X, Mouse.GetState().Y)) and finds a relative position vector between the centre of the CueBall and the mouse position to give a direction and magnitude of initial velocity when shooting. This means that the CueBall will always move in the direction you click (provided it’s stationary beforehand), making aiming intuitive. It also means that the further you click away from it, the faster it moves.

public void Shoot(Vector2 mousePosition)

{

Vector2 movementVector = mousePosition - position;

velocity += movementVector;

}

However, since this approach causes the CueBall to move the distance between the click every frame, it moves way too fast. To fix this, I added a constant float VelocityMultiplier which equals 1 / 25 (a value found through experimentation that moves the CueBall just a bit further than the distance of the mouse from it). movementVector is then multiplied by this before velocity is changed, as shown:

public void Shoot(Vector2 mousePosition)

{

Vector2 movementVector = mousePosition - position;

velocity += movementVector \* VelocityMultiplier;

}

I also added a DebugMove method that just sets the position of the CueBall to the mouse position (and the velocity to 0 to stop it moving).

public void DebugMove(Vector2 \_position)

{

velocity = Vector2.Zero;

position = \_position;

}

In the Update method, it is triggered with the right mouse button and teleports the CueBall to the mouse position.

if (Mouse.GetState().RightButton == ButtonState.Pressed)

{

DebugMove(currentMousePosition);

}

Since that is all implemented, it’s time to add friction to all PoolBalls. For simplicity, we will always consider stationary PoolBalls to be on the point of moving, meaning that for all situations we can use the formula: friction on the PoolBall = coefficient of friction between the PoolBall and the surface \* reaction force of the PoolBall due to the surface. By resolving forces vertically, we find that the reaction force is equal to the mass of the PoolBall multiplied by the gravitational field strength the PoolBall is experiencing. (For simplicity, we will set the gravitational field strength to 1 unit). This means that the force of friction the PoolBall experiences has a constant magnitude and acts opposite to the motion of the PoolBall, letting me decide the exact value myself through experimenting with only changing one variable (the coefficient of friction between the PoolBall and the surface). In the end, I landed on [placeholder (see NOTE below), 0.01].

Putting this inside of a DoFriction method, we get:

public void DoFriction()

{

// horizontal:

if (Math.Abs(velocity.X) > 0.1)

{

velocity = new Vector2(velocity.X - decelerationDueToFriction.X, velocity.Y); // decelerating

}

else

{

velocity = new Vector2(0, velocity.Y);

decelerationDueToFriction = new Vector2(0, decelerationDueToFriction.Y);

}

// vertical:

if (Math.Abs(velocity.Y) > 0.1)

{

velocity = new Vector2(velocity.X, velocity.Y - decelerationDueToFriction.Y); // decelerating

}

else

{

velocity = new Vector2(velocity.X, 0);

decelerationDueToFriction = new Vector2(decelerationDueToFriction.X, 0);

}

}

[NOTE: change friction in the code to work as above]



### 1st iteration review

## Notes

1. Made cue ball
2. Added collisions to side of wall
3. Added friction
4. Added main menu (partly using the Myra library) and the ability to go from main menu to match screen

## Problems

* PoolBall suddenly stops when moving
  + Fixed: code originally set velocity to zero when either v.x or x.y was close to zero, now it only sets the component of velocity that is close to zero to zero instead.
  + Before: velocity = Vector2.Zero;
  + After: velocity = new Vector2(0, velocity.Y);
* PoolBall occasionally accelerates forever either vertically or horizontally
  + Probably due to decellerationDueToFriction being the negative of what it’s supposed to be.

## Things worked out

### Elastic collision between circles

The DoCircleCircleCollision() method is inside of CueBall only for now because that way I can simplify the problem by looking at moving-stationary circle-circle collisions.

I started by adding an array of PoolBalls to Match1 so that it can itterate through it to collide with all the objects on the table. Then it checks to see if the centre of the cueBall is within two radius lengths of the centre of the other PoolBall (i.e. checking if they are colliding).

Initially, I just set both velocities to zero to check if it worked, and it did.

public void DoCircleCircleCollision()

{

foreach (PoolBall poolBall in Match1.poolBalls)

{

if (poolBall == this) // no need to check if it collides with itself

{ continue; }

else

{

if (Vector2.Distance(poolBall.position, position) < radius \* 2)

{

// [placeholder, for testing]:

poolBall.velocity = Vector2.Zero;

velocity = Vector2.Zero;

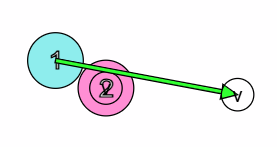
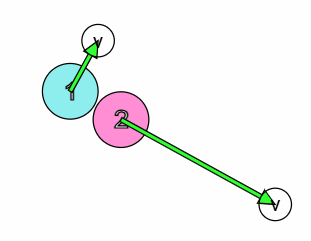
}

}

}

}

To calculate the velocities of both colliding balls after the collision, I began by looking at different scenarios using the PhET Collision Lab. For example, if the two balls collide like below, I could calculate a multiple of ball 2’s velocity by calculating the distance between the two centres. Ball 2’s is similar to ball 1’s, except it is rotated 90 degrees anticlockwise. Using rotation matrixes, I arrived at the following vector equations for the path of the two balls.

## Testing to inform development

# Evaluation

## Testing to inform evaluation

## Success of the solution

## Describe the final product

## Maintenance and development

# Bibliography

* IMDB Gang Beasts Parental Guide (<https://www.imdb.com/title/tt10950132/parentalguide>)
* Jackbox Games “How many players can join each game?” (<https://support.jackboxgames.com/hc/en-us/articles/15794756085015-How-many-players-can-join-each-game>)
* World 8 Ball Pool Rules 2009 (<https://www.wepf.org/docs/rules_2014.pdf>)
* Myra documentation (<https://github.com/rds1983/Myra/wiki>)
* PhET Collision Lab (<https://phet.colorado.edu/en/simulations/collision-lab>)
* Colorado State University Billiards (<https://billiards.colostate.edu/>)