TECHNICAL DESIGN DOCUMENT: RESILIENT KIWI MAGNIFICENT DREAM

fyy7680

RKMD AUT Game Programming Team

Table of Contents

Descriptions of key game logic and technical algorithms

Screen movement algorithm

Screen

- The screen movement algorithm relies on the players y-axis to be above halfway through the current section before moving the section down to simulate a screen moving downwards.
- Since all the entities such a platform, enemies, and powerups are tied through the section's coordinates, this will enable us to move everything downwards.

Upward gameplay

- The screen will start with the player on the ground and bottom of the screen.
- As the player moves upward, once the player reaches the centre of the y-axis, the screen will start to centre on the player, following them from then on as they move upward.
- If the player falls, the screen will not follow in the downward direction.
- If the player reaches the bottom of the screen they will die.
- As the player progresses higher, the screen will begin to "creep" upward, forcing the player to keep moving. If it catches the player, they die.

Downward gameplay

- The player will be falling downward the whole way – with no chance of the screen catching them. The screen will follow the player downward, again keeping them in the centre of the screen.

Player movement algorithms

Left/Right movement

- The player will be able to move left and right through a combination of method calls and input handling (GP2D framework's input handler). Calling these methods will update the players velocity x-axis based on the players movement speed, as well as the direction (Negative and positive sides of the players current x-axis).

Jumping

- Like the left and right movement, player jumping will be done through a combination of method calls and input handling (GP2D framework's input handler). This will update the players velocity y-axis based on the players jump power. The velocity's y-axis will be decremented according to the jump power which will simulate gravity.

Shooting

- Shooting will be handled by different key buttons (arrow left, right and up on keyboard). The players handle input method will take in a SDL key code which will create a new laser beam and initialise its direction based on the players input.

Platform placement

- The platform placement algorithm is done through a combination of reading the INI files which has the directory location for our pre-made sections, sections, section managers and different platform classes.
- These sections are made manually with different ASCIIS such as:
    - # = Static platforms.
    - ^ = Upward moving platforms.
    - < = Sideways moving platforms.
    - ~ = Collapsing platforms.
- A section manager will handle reading in our 'sec' files which are text files filled in with ASCIIS mentioned before. The section manager will then create an object based on this information, which will then be stored in a list ready to be drawn on screen.

Score keeping algorithm

- Our score keeping algorithm will be tied with our play state as it involved the players y-axis position and the game engines window height.
- The score uses meters as a unit and uses the distance camera (divided by 10) as reference of how much the player has moved.
- The score will only increment as soon as the player passes the centre of the game screen.

Level logic

- Our levels will be loaded in using a data driven design that will read in our section files. These section files are manually designed and are loaded in randomly using a loaded section list each time a section is placed back in the list.
- The levels will have a mix of difficult and easy sections for the players to navigate through with some challenging ones that involve the player to think fast and get creative with the wrap around feature.
- As the player progresses through the game, the theme of the game such as enemies, platforms and backgrounds will change, which in return will deliver a more immersive gameplay.

Collisions logic

Platform Collisions

- Shooting enemies will implement a two rectangle overlap check. (Geeks for geeks, 2020)
- For Platform collisions, the player will be able to jump through the bottom of the platforms and land on top. This collision will be checked by getting the players feet hitbox (X-axis, Y-axis, width, height) and checking if the platforms top height which collides with the feet hitbox. This collision will be checked on each process method of all the player states except the jetpack state.

Entity vs Entity Collision

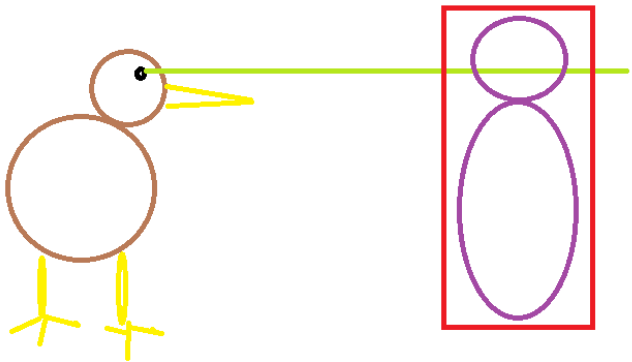
- Entity vs entity collision will be done similar with a circle vs circle overlap check. An enemy entity will have a method that requires an entity parameter and will return a Boolean.
- A circle around each entity is set up along with conditional statements if the two circles have overlapped which determines what Boolean value is returned.

Laser eyes

The kiwi will shoot lasers from its eyes. This will be accomplished by using a rectangle collision detection from the laser rectangle to the enemy rectangle.

The laser will be a subclass from the Entity class, this will allow the laser class to inherit all the methods and member data of entity. The laser class will have its own unique methods such as:

- Bool initialise (Sprite, Player, Laser beam direction)
    - Initialises the player, the laser's x and y coordinates, sprite, and switch cases the direction enum to its respective rotational values and velocity.
    - Returns a Boolean on entity's initialisation of the sprite.
- Void Process (deltatime, Enemy pointer list)
    - Updates the laser's position based on the velocity and delta time.
    - Checks for collision with enemies in the enemy list.
    - Checks if the laser is out of screen bounds, vertically and horizontally.
    - Performs the normal entity process method.
- Bool IsCollidingWith(entity)
    - Rectangle collision detection which is similar to the platforms collision detection.



Sketch for Kiwi rectangle collision with laser beams and enemies

Sprite and Animated sprite management

Sprites will be managed by a singleton which will enable sprites to be created and retrieved by the manager instance. The sprite manager can also be called anywhere with its GetInstance() method. The sprite manager will have multiple useful public member methods such as:

- Static SpriteManager& GetInstance()
  - Calls and returns an instance of SpriteManager.
- Static void DestroyInstance()
  - Deletes the instance of SpriteManager and sets the instance to point to null.
- Static void InitialiseManager(BackBuffer* backBuffer)
  - Initialises the sprite manager's BackBuffer pointer to the backbuffer parameter.
  - Creates a new AnimatedSpriteManager object and initialises it.
- Static Sprite* CreateSprite(const char* pcFileName)
  - Creates a sprite, returns it and also saves it into the managers member vector of Sprite pointers called m_SpriteList.
- Static Sprite* CreateSpriteFromSheet(const char* pcFileName, int x, int y, int width, int height)
  - Crops out a single sprite from a sprite sheet, stores the sprite in the sprite managers sprite vector and returns the sprite.
  - Initialises a new sprite of the whole sprite sheet.
  - Takes in the parameters such as file name (texture), x and y coordinate, width, height and passes them to the new sprite's initialise method.
  - Saves the new sprite into the sprite managers sprite vector and returns the new sprite.
- Static Sprite* GetSprite(const char* pcFileName)
  - Returns a sprite by passing the file name into the method.
- Static AnimatedSpriteManager* GetAnimatedSpriteManager()
  - Returns the instantiated member pointer of AnimatedSpriteManager.

The animated sprite manager however is more of an initialiser to set all of the animations on game start-up ready to be retrieved by its public methods. The animated sprite manager can also be scaled up by setting up additional animated sprite members, initialisation's and getter methods.

- Void AnimatedSpriteManager Initialise()
  - Instantiates all of the animated sprite members and sets all of their respective properties such as total frames, frame speed, frame width, frame height.
  - The animated sprite members are then initialised with the player sprite sheet texture and the y-axis value which determines what specific row will be animated from the sprite sheet.
- Getter methods (AnimatedSprite* AnimatedSpriteManager GetRunningRight ())

    - Multiple getter methods are available to be called which returns an animated sprite row from the sprite sheet.
- Destructor
    - The animated sprite managers destructor will delete all animated sprite members of the class and set them to a null pointer.

Player states

The player will have different Enum states which will act as behaviours.

The player will have Enum states such as:

- STANDING
- FALLING
- WALKING
- JUMPING

Each Enum state will have their own conditional statements under the player process that will specify and apply the respective velocities for the player action to happen. The player will also have its own getter and setter methods for each state.

Sound management

The sound manager will be a singleton like the sprite manager. This will enable us to call the sound manager instance anywhere in the code and initialise all sound effects.

The sound manager will have multiple useful methods such as:

- Void SoundManager ToggleSound()
    - Mutes or unmutes the sound effects
- Void SoundManager Play…SFX()
    - Plays a certain sound using the FMOD system with the WAV sound created when the sound manager was initialised. Only plays the sound if the sound on member condition is met.

Section management

The sections within the game will be managed through a combination of lists, methods, and the INIReader under a section manager class. The section manager will contain the following methods:

- Bool SectionManager LoadSections (char* sectionFileName)
    - Returns a Boolean and creates new sections and adds its respective entities such as platforms, enemies and pickups based on the initialisation file, section files, and conditional statements along with char values that represent certain entities.
    - This method also shuffles the sections for a randomized gameplay.
- Section* SectionManager DespawnSection()
    - Despawns the last section in the loaded section list and returns the front section from the active section list.
- Void SectionManager ShuffleSections()

    - Randomizes the sections around using random_shuffle.
- Void SectionManager ResetSections()
    - Resets all sections within the active and loaded section lists.
- PickUps* SectionManager SpawnRandomPowerUp()
    - Returns a random powerup using a random number generator. The spawn rates are:
        - 20% chance to spawn gumboots.
        - 20% chance to spawn invincibility.
        - 10% chance to spawn the jetpack.
        - 50% chance to spawn a coin.

Particle management

Particles will be managed using a particle base class and a particle emitter base class, both extending from Entity. Specific particle emitters will extend from the base particle emitter class, specific particles will extend from the particle base class.

Particle emitters will contain a fixed array of particles, whose type will be determined depending on the effect desired. The particle emitter will determine the location of the particles based on its own location. Particle attributes will belong to the particles, particle behaviour will be set by the emitter.

Objects in the game that emit particles in some cases will have their own particle emitter e.g. the kiwi will its own emitter. However, where particle emitters can be shared and reused, they will be, such as particles emitted from powerups and coins. Only objects currently being processed drawn need to have access to a particle emitter.

Particles (and emitters) required (but not limited to):

- dust particles on player landing
- sparkle particles associated to powerups (and kiwi when has a powerup)

Dust particle effects:

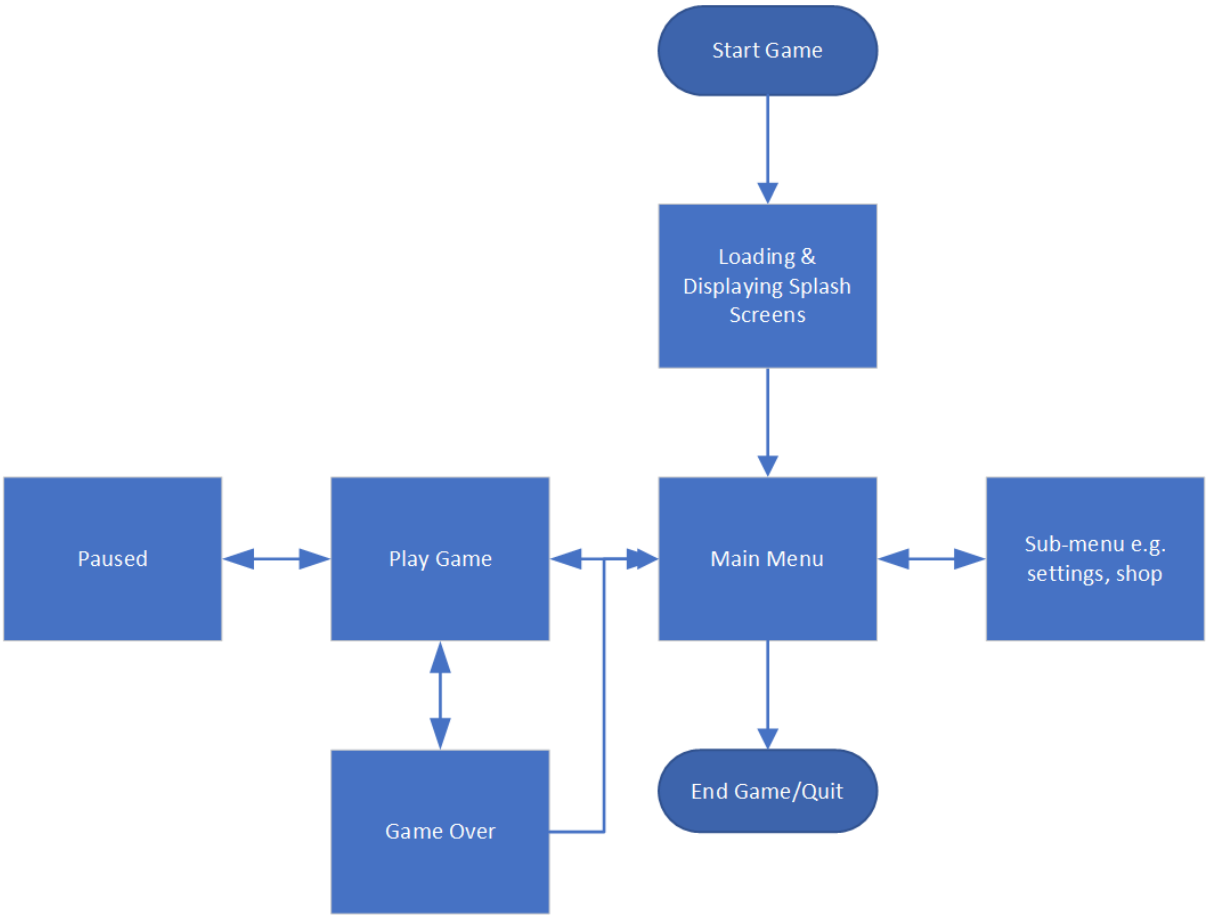
- will have a small brown sprite, not much larger than a single pixel
- they will ‘explode’ in all directions (randomly) from the base of the kiwi when it lands
- particles will be quickly alpha blended out and then reused
- may need gravity applied to them if they are drawn long enough

Sparkle particle effects:

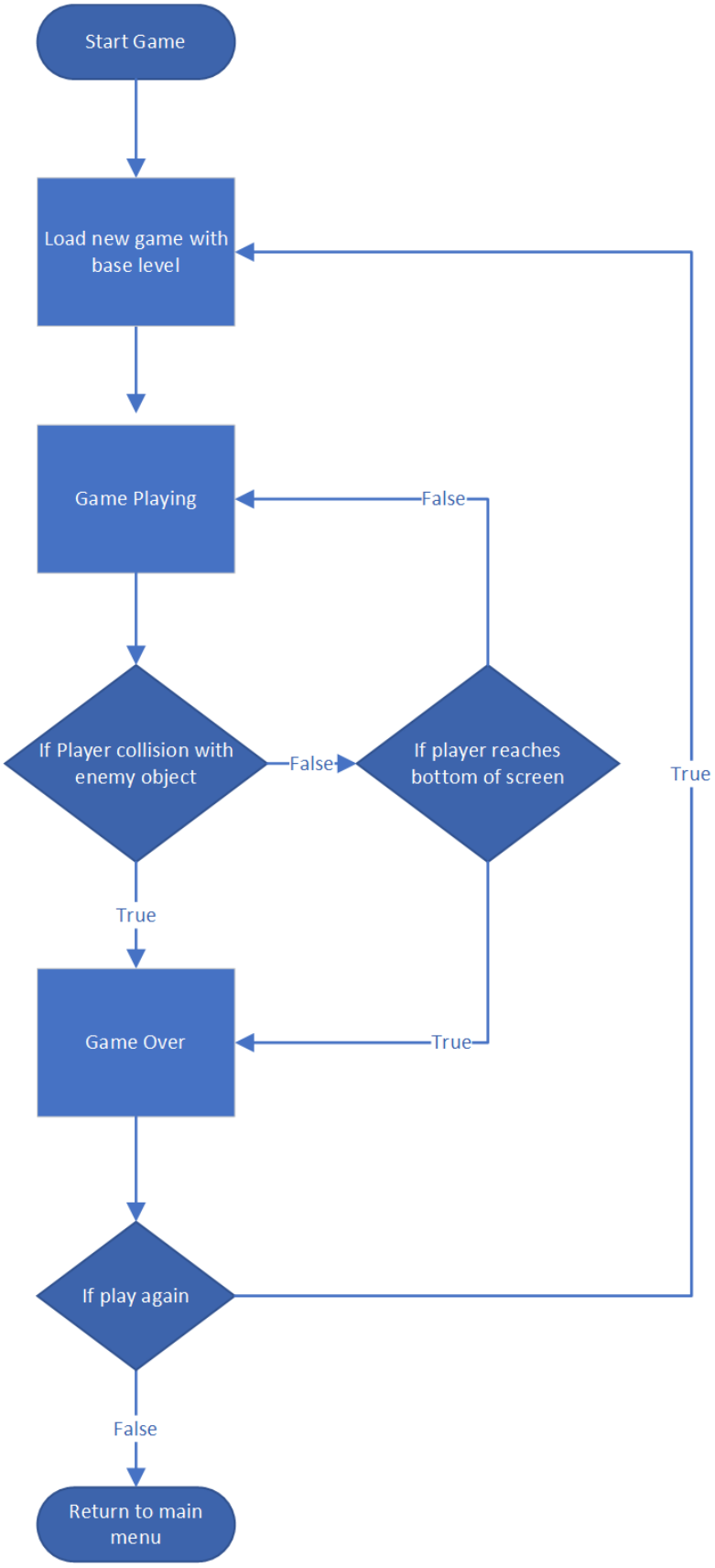
- will have small, shiny, multicoloured sprite
- will ‘float’ upward off of the pickup they are associated to
- will be randomly spawned across the width of the pickup they are associated to
- will be alpha blended out after travelling a short distance

- if time permits the kiwi will have particles associated to it when it picks up a powerup

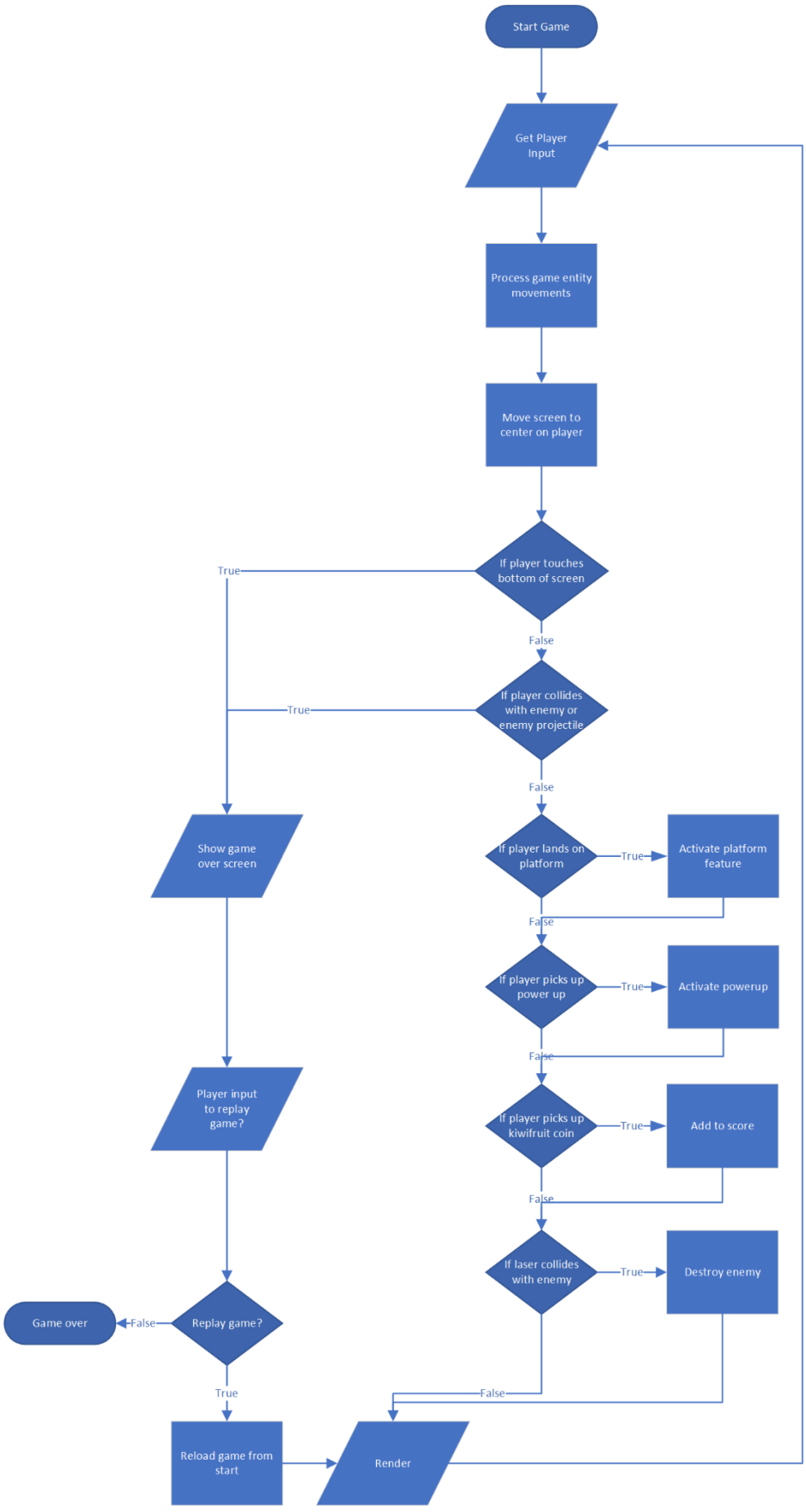
Overall game loop – menu included



Basic overall in-game loop



Real-time in-game loop



GameState Machine

Game states will be managed by a finite state machine. The main game.cpp class will contain a std::vector which will hold the various different game states. Game object will call Draw, Process and input handling for the GameState at the top of the vector stack. GameState can either be replaced altogether with ChangeState(), or otherwise can be pushed and popped for instances such as pulling up the game menu over the PlayState and then resuming play again.

Each state will inherit the following methods from the base GameState class:

- virtual bool Initialise(Game* game)
    - Initialises any required member variables belonging to the state
- virtual void CleanUp()
    - Deletes any member pointer variables that should not persist beyond the state
- virtual void Pause()
    - Method called when pausing the state when pushing another state on to the stack
- virtual void Resume()
    - Method called when unpausing the state when popping the top state off
- virtual void Process(float deltaTime)
    - Processing state method
- virtual void Draw(BackBuffer& backBuffer)
    - Draw state method
- virtual void HandleEvents()
    - Calls the state's event handler method
- void ChangeState(GameState* state)
    - Calls game.cpp change state method for the next state

The current state will control which state is called next based on the circumstance e.g. quitting to menu, starting play etc.

States will be singleton instances and will be destroyed at game quit.

Game states will include, but not be limited to:

- IntroState
- MenuState
- PlayState
- GameOverState
- SettingsState
- CreditsState
- ShopStatemethods
- StatsState

Data driven design

Data driven design will be implemented through an initialisation file reader. Most of the map design including platforms, enemies, powerups, and sections will all be data driven. This is done basically by reading in the section file location from an initialisation file and translating each ASCII value to their respective entities.

Key	Value
#	Static platform
~	Crumbling platform
^	Moving up platform
V	Moving down platform
<	Moving left platform
>	Moving right platform
G	Ground Passive Enemy
H	Ground Aggressive Enemy
F	Flying Aggressive Enemy
$	Kiwi Coin
B	Gumboots Powerup
L	L and P Powerup
J	Hi-Vis Powerup
P	Random Powerup

Debug features

Debug features will be available through key toggles:

F1 key for PC draws game debug information while playing the game:

- Player X and Y positions
- Player X and Y velocities
- Player Distance Jumped
- Current player state
- Current player facing direction
- Player previous Y position
- Player feet hitbox
- FPS

F2 key for PC gives the player an invincibility power-up.

User Interface

User Interface Class

The user interface class represents a single user interface that is in the game. EG: The main menu, GUI, or Settings.

Each User Interface object will have a list of InterfaceComponent objects. Each InterfaceComponent could be a button or a label etc. When the User Interface is drawn, it will call the draw function of each Interface Component, drawing them to screen.

Similarly, to check if a user interface element is clicked. There is a function that takes an X and Y coordinate for each user interface. This coordinate will be checked against each component and will return the interface component that is found at that location. If no component is found. A null pointer is returned.

Interface Component Class

The interface component class represents a single component in a user interface. This will contain several virtual functions that are inherited by sub classes so that their sub class code is called when the user interface calls a component's function.

Interface components we need in our game will be:

- Label
  - Used to display a text label
- Button
  - An interactive component the user can click. Will extend from label but build ontop of to make it more interactive with hover and clicked effects.
- Rectangle
  - A rectangle object that can be used as a background
- Toggle Button
  - Extended from Button and will have support for toggling displaying different effects if it is toggled or not.

Development standards

These development standards have been derived from Insomniac games coding standards (Insomniac Games, 2011).

Coding standards and naming schemes

Naming and stylistic standards

- Uppercase PascalCase for:
  - Type names (struct, class, Enum)
  - Methods
- Lowercase pascal Case for variable identifiers
- No leading or trailing underscores
- Member data start with m_ (e.g. m_variable)
- Pointers start with p_
- Member pointers start with m_p
- Pointers – group asterisk with the type, not the variable
- No using the comma operator to declare multiple variables on the same line
- Describing identifiers – long is OK
- Numerals OK in identifier names, but not leading
- Source code files all lowercase naming
- When choosing a name, prefer long names to abbreviated names. Prefer descriptive names to arbitrary ones

Key coding standards

- Return types of functions should have their own line in .cpp file
- Single return value in functions
- Do not use the using keyword - all namespaces explicitly referenced
- Tab indents, not double space
- Always use scope brackets and use separated line
- NZ spelling
- Accessors = set, get, is, does
- Do not use #include if a forward declaration would do

Best practice

- Regular and descriptive commenting
- Prune commented-out and unused code
- Avoid negative flags
- Clean and sensible whitespace
- Avoid magic numbers

Relevant file formats

- Sound files must be in wav format

- Art, animations, sprites must be in PNG format
- C++ source files (.cpp) and standard C/C++ headers (.h)
- Microsoft Visual Studio .NET solution (.sln).
- Microsoft Windows initialisation (INI) files for platform stage data driven design

Development environment, tools and required technology

IDE	Microsoft Visual Studio Enterprise 2017
Version control	TortoiseSVN - https://dctwsvn.aut.ac.nz/svn/COMP710-2020-S2/ repository
Development language	C++
Build tools: pre-processor, compiler, and linker	Visual Studio Platform Toolset: Visual Studio 2017 (v141)
Windows SDK	Windows SDK Version – 10.0.177763.0

Middleware and libraries

- COMP710 2D C++ Framework
- C++ Standard Template Library (STL)
- Simple DirectMedia Layer (SDL2) including SDL2_Image and SDL2_TTF APIs;
- FMOD Core (Low-Level) API.
- STL

TortoiseSVN Global Ignore Pattern Set-Up

All team members are required to configure their global ignore pattern for TortoiseSVN to avoid comitting unnecessary files.

IGNORE FILE SETUP STRING:

*.o *.lo *.la *.al .libs *.so *.so.[0-9]* *.a *.pyc *.pyo __pycache__ *.rej *~ #*# .#* .*.swp .DS_Store [Tt]humbs.db *.obj *.exe *.suo *.user *.pdb *.idb *.ilk *.sdf *.pch *.ipch *ipch* *.res *.asp *.tlog .git .vs *.sbr .gitignore .gitattributes

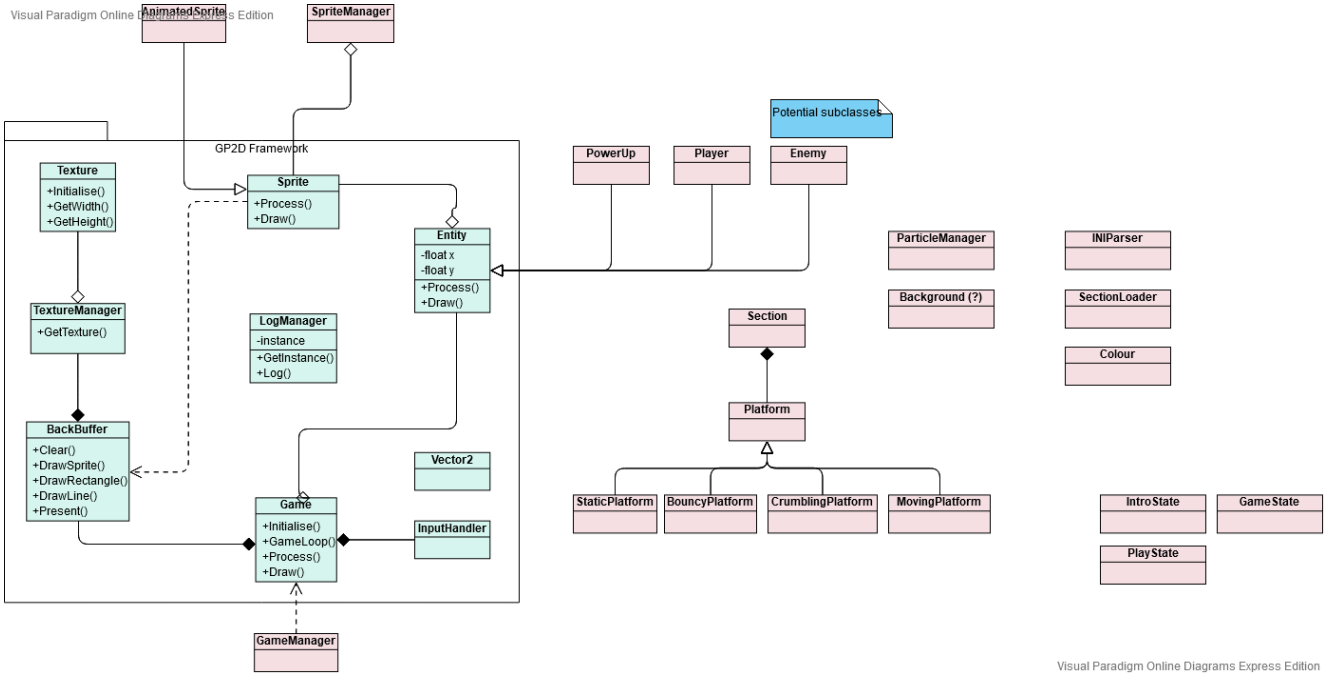
Go to TortoiseSVN > Settings > General > Subversion: Global Ignore Pattern

Key Technical Challenges and Possible Solutions

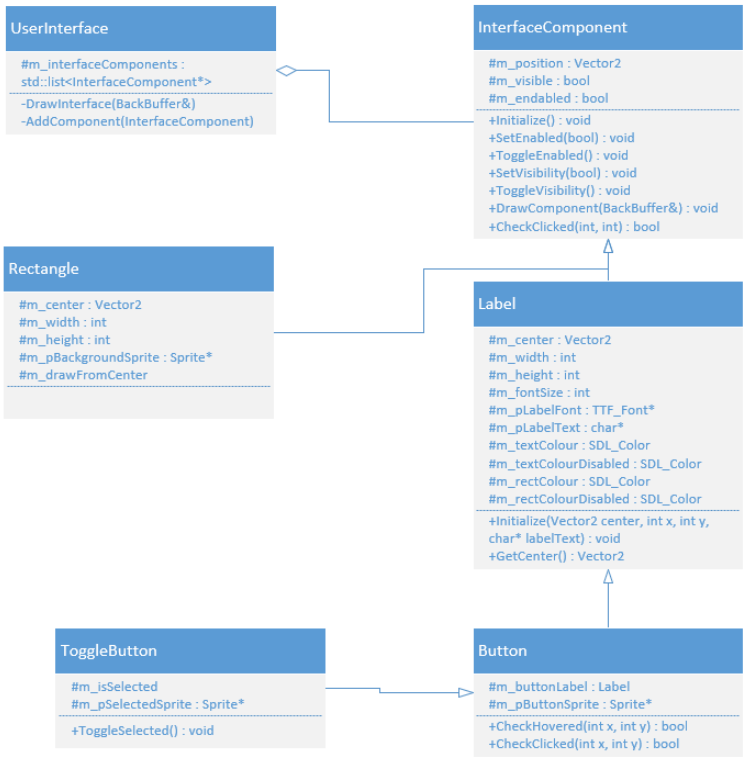
Potential challenges	Potential solutions
Issues with SVN merging, potentially deleting someone's work or not being able to correctly merge changes to a file.	• Be proactive and let the team know when any SVN issues happen. • Send a message in the Updates channel on Teams after every commit so other team members know when to update their local copy of the repository.
Handling memory leaks	• Bring up the memory leak in the daily stand-up so we can fix it together as a team.
RISKS – people risks, time etc, covid	• Online daily stand up meetings. • Online pair / mob programming.
Player or Map scrolling camera can potentially be difficult to implement.	• Tie all the entities within each section to the sections y coordinate. • Player crossing halfway through the screen will move the section down, moving all entities along with it simulating a scrolling effect.
Enum implementation can scale up the code quite fast. Closely related to the animated sprites.	• Closely related to the animated sprites. • Enum implementation however can scale up the code quite fast. • Player states classes having a hierarchy with their own processes and methods. Could also have a manager which will initialise all states and retrieves them respectively.
Moving Platforms might alter the players x and y coordinate when on top of it. Might affect player movement.	• Adjust the jump power and gravity according to the moving platforms velocity.

Diagrams of important system technical designs

Here are some class diagrams.



Class Diagram for expected classes



Class Diagram for interface classes

Key object hierarchies, interactions, properties, and functionality

Item object hierarchy

Items that are able to be picked up will include:

- Kiwicoins (in game shop money)
- Powerups
    - Spring gumboots
    - HiVis jacket and hardhat
    - L&P jetpack

All items will extend from pickup class which will extend from the Entity class. The Pickup class will have a unique method for collisions using the circle versus circle algorithm. Item effects will be different for each item. Multiple powerups can be picked up at once.

Kiwicoins are added to the players score/wallet and can be used in the shop, outside the main gameplay. These can be found frequently throughout the game.

Spring gumboots:

- 20% chance to spawn
- gives the player double height (factor to be tested) jump power for 5 seconds
- may auto jump upon landing
- player can shoot and move sideways during this power up

HiVis jacket and hardhat:

- gives the player invincibility against enemy attacks for 10 seconds (time to be tested)
- 20% chance to spawn
- all player functionality is available during this power up
- player can still die if they fall to the bottom of the screen

L&P Jetpack:

- 10% chance to spawn.
- fires the player up the screen by at least a screens length, and during which gives the player invincibility
- lasts for a set distance (to be determined)
- player can move sideways during this powerup and may be able to shoot

All powerups are simply classes of PickUps that can be collided with. Upon collision a Boolean value for the relevant powerup will be set in the Player class that indicated that the powerup is now on. If a power up Boolean is on then the process method for that Boolean should be run. The process method should also manage the powerup time limit as well as the effects.

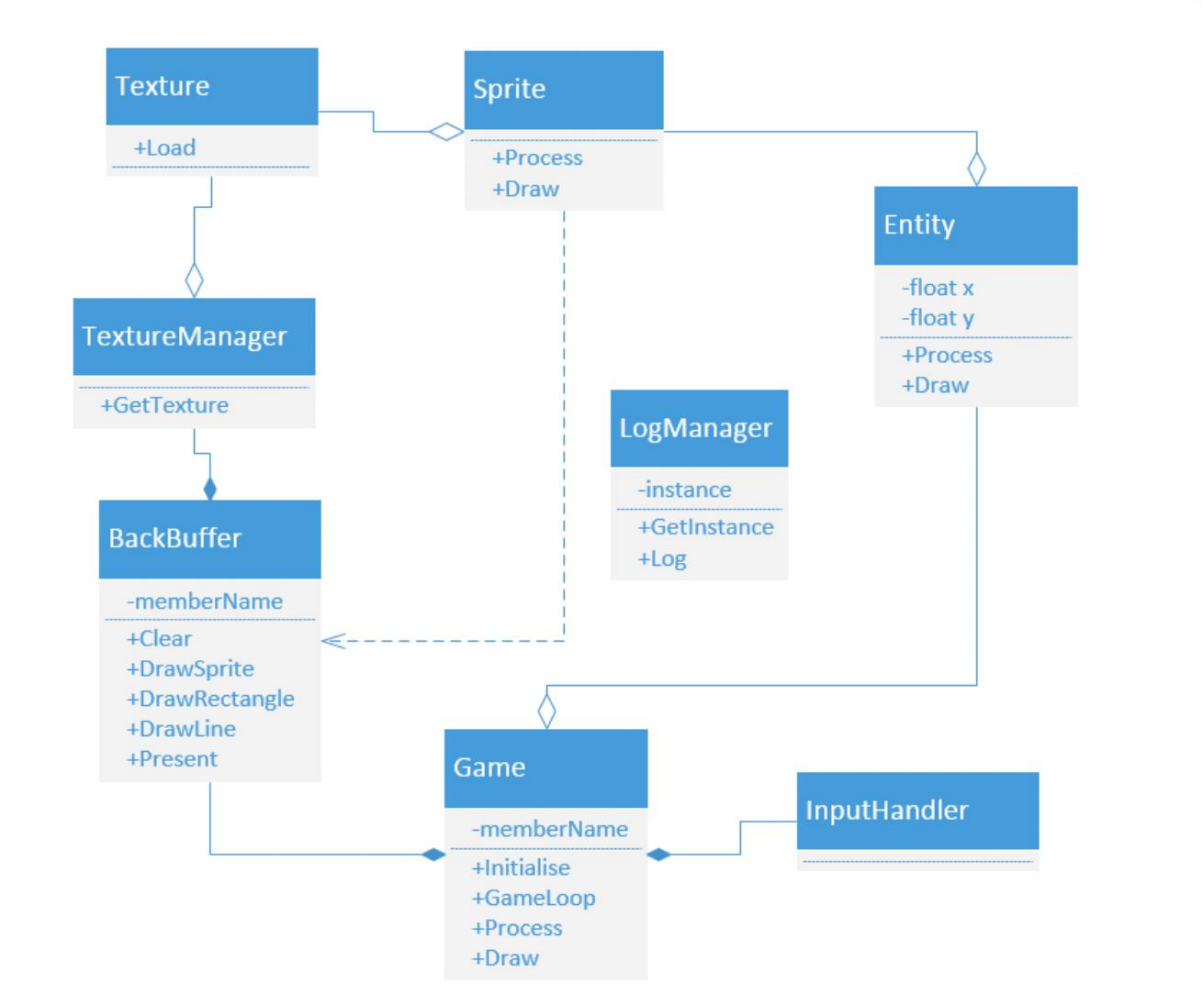
Power ups will be placed using the .ini file. Powerups should be kept in an object pool, only enough need to be created so that there are enough for the particular screen. After they are

outside the vertical bounds of the screen they can be set to dead and placed somewhere else. A class should be created to manage the placement and lifecycle of these.

UML Class Diagram(s) for Object Hierarchy and Object Interaction

Basic game framework

The basic framework which forms the basis of the following UML diagrams (Hooper, 2020b).



Object Pools

The following game objects will be stored in arrays and generated at the beginning of the game.

- Platforms
- Kiwifruit coins
- Powerups
- Particles

Particles will be alpha blended out before they are reused.

Acceptance plan

1. Screen moves upward as per this document and reverses at the top for downward gameplay.

2. Game includes a player kiwi which can be controlled by the player to walk/run (left and right), shoot lasers (left, right & up) and jump. Its movement is smooth, responsive and fun.

3. Game includes all listed powerups, platforms, and coins/kiwifruits and enemies which function as described in this document.

4. Game loads near instantly and runs smoothly.

5. Game features all five levels and can be won via a high score.

6. Game is lost upon player collisions with enemies and the bottom of the screen.

7. Game loads showing splash screens and loads menu which functions as described in the documents.
8. Can be started over from this menu without restarting application.

9. Game includes all HUD elements as described in this document and GDD.

10. Game plays with both keyboard and controller.

11. Game includes all sound effects, particle effects and animations as described in this document.

Team signoff

All team members agree with the content in this Technical Design Document for RKMD game.
Date signed: Friday 2nd of October, 2020.

Maya Ashizumi-Munn:

Kadin Honeyfield:

Devin Grant-Miles:

Robert Dumagan:

Bibliography

Hooper, S. (2020b). *COMP710 Game Programming: W02 Studio Session 04 Lec 04a - Real-Time Simulation, 2D Graphics, GP 2D Framework* [PowerPoint slides]. Blackboard. https://blackboard.aut.ac.nz/

Hooper, S. (2020f). *COMP710 Game Programming: W05 Studio Session 08 Lec 08b - Game Production* [PowerPoint slides]. Blackboard. https://blackboard.aut.ac.nz/

Insomniac Games. (2011, October 3). *Core Coding Standards.* https://gist.github.com/Kerollmops/fcad27cfef9e3552cb75a3d201494ba6

GeeksforGeeks.org. (2020). *Find if two rectangles overlap.* https://www.geeksforgeeks.org/find-two-rectangles-overlap/