Department of Creative Technology

Course

Object Oriented Programming

Instructor:

Mr. Zain

Project Title:

Frogger

Group Members:

Muhammad Asadullah Turab (221844)
Muhammad Saad (221816)
Mehboob Ali (221824)

Table of Contents

1.Introduction

Welcome to the comprehensive document detailing the development and intricacies of the Frogger game, programmed in C++. Frogger is a beloved arcade classic that challenges players to guide a frog through a perilous environment, avoiding obstacles, and reaching the designated goal. This document will delve into various aspects of the game, providing in-depth analysis of its objectives, gameplay mechanics, graphical assets, sound effects, and user interface, offering a comprehensive understanding of this timeless game.

- **Objectives:**
  The primary objective of Frogger is to navigate the frog safely across a bustling road and a treacherous river, ultimately reaching the coveted goal. The player takes control of the frog, maneuvering it through a series of lanes filled with fast-moving vehicles. The challenge lies in avoiding collisions and overcoming hazards to progress to the next level. Once the road is crossed, the frog faces the added challenge of traversing the river, leaping onto moving logs or turtles to reach one of the safe zones on the opposite side.

- **Gameplay Mechanics:**
  The gameplay mechanics of Frogger are simple yet demanding, testing the player's agility and strategic thinking. Through the use of directional keys, the player controls the frog's movement, allowing it to hop forward, backward, left, or right. Precise timing and calculated movements are crucial as the frog must navigate the traffic-filled lanes without colliding with vehicles. Furthermore, the player must find opportune moments to leap onto moving logs or turtles in the river, ensuring safe passage and avoiding the perils of sinking into the water.

- **Graphical Assets:**
  To create an immersive experience, Frogger incorporates visually captivating graphical assets. These assets comprise distinct and recognizable sprites for the frog, vehicles, logs, turtles, and environmental elements. The road and river sections should be visually distinguishable, allowing players to identify the various elements within the game world quickly. The visual design may feature vibrant colors, enhancing the game's appeal and providing visual cues for players to make informed decisions.

- **Sound Effects:**
  Sound effects play a pivotal role in enhancing the overall gameplay experience in Frogger. Immersive audio cues such as the revving of engines, the splashing of water, the hopping of the frog, and victory or failure jingles provide valuable feedback and reinforce player actions. Well-designed sound effects contribute to the game's atmosphere, heightening tension and excitement as players navigate the hazardous environment.

- **User Interface:**
  The user interface (UI) of Frogger encompasses various elements that facilitate smooth interaction between players and the game. The UI includes menus, score displays, and level progression indicators. It should provide an intuitive navigation system, offering clear instructions and options for players to navigate through the game. Relevant information, such as the current score, remaining lives, and level progression, should be

prominently displayed, enabling players to track their progress and make strategic decisions accordingly.

In conclusion, this document has provided an in-depth exploration of the Frogger game developed in C++. By understanding the game's objectives, gameplay mechanics, graphical assets, sound effects, and user interface, we gain valuable insight into the development and design of this timeless arcade classic. Frogger continues to captivate players with its engaging gameplay and remains an iconic example of the arcade gaming era.

1.1 Goals and Objectives

Frogger offers players a captivating gameplay experience with clear goals and objectives that drive their engagement and progression. The game presents a series of challenges, requiring players to navigate an increasingly hazardous environment while demonstrating quick reflexes, strategic thinking, and precise movements. Let's delve deeper into the goals and objectives of Frogger:

- **Safely Navigate the Road:**
  The primary objective in Frogger is to guide the frog safely across a bustling road. The player must skillfully maneuver the frog through lanes filled with a variety of moving vehicles, such as cars, trucks, buses, and motorcycles. The goal is to avoid collisions with these vehicles, as contact results in the loss of a life. Successfully crossing the road requires careful observation of traffic patterns, identifying safe openings, and timing movements accurately to make it to the other side unharmed.

- **Traverse the Treacherous River:**
  Once the frog has successfully crossed the road, it faces a new challenge in the form of a treacherous river. The objective here is to navigate the frog across the river by hopping onto moving logs or the backs of turtles. The logs may move horizontally, while the turtles might submerge and resurface at regular intervals. Players must time their jumps and landings precisely, ensuring the frog stays afloat and avoids falling into the water. Each successful leap brings the frog closer to the riverbank and closer to achieving the overall objective.

- **Reach the Designated Goal:**
  The ultimate goal in each level of Frogger is to reach the designated goal on the opposite side of the riverbank. This goal typically takes the form of a safe zone or a lily pad. Players must guide the frog to the goal area, which often requires a final leap onto a stationary object or landform. By successfully reaching the goal, players complete the level and progress to the next, where new challenges and obstacles await. Reaching each goal provides a sense of accomplishment and advancement in the game.

- **Achieve High Scores:**
  In addition to the primary objectives, Frogger introduces a scoring system that motivates players to strive for high scores. Points are awarded for various achievements throughout the game. Crossing the road safely, hopping onto moving logs or turtles, reaching the goal, and completing levels within a time limit are all potential ways to accumulate points. Players can aim to surpass their previous high scores or compete with others to

establish themselves at the top of the leaderboard, adding a competitive element to the game.

- **Progressive Difficulty:**
  Frogger is designed to challenge players as they progress through the levels. The game gradually increases in difficulty, presenting new obstacles, faster-moving vehicles, narrower gaps between traffic, and more unpredictable movements in the river. As players advance, they must exhibit greater dexterity, reflexes, and strategic decision-making to navigate the increasingly complex and hazardous environments. The progressive difficulty ensures that players are constantly engaged and motivated to improve their skills to overcome the escalating challenges.

- **Time Management:**
  In some variations of Frogger, players must contend with time limits. The game imposes a countdown, adding an additional layer of pressure and urgency. Players must efficiently navigate the road and river within the allotted time, making strategic choices and avoiding unnecessary risks. Time management becomes crucial in balancing speed and caution, adding an element of excitement and intensity to the gameplay.

In conclusion, Frogger offers players a range of goals and objectives that drive their gameplay experience. Safely navigating the road, traversing the treacherous river, reaching the designated goal, achieving high scores, facing progressive difficulty, and managing time effectively are all integral aspects of the game. These objectives combine to create an engaging and challenging gameplay loop that keeps players immersed and motivated to overcome obstacles and achieve success in the ever-changing world of Frogger.

1.2 Constraints

The development of Frogger is subject to various constraints that influence the game's design, implementation, and overall experience. These constraints may arise from technical limitations, target platforms, available resources, and design considerations. Understanding and working within these constraints is crucial for delivering a successful and enjoyable game. Let's explore the detailed constraints involved in the development of Frogger:

- **Hardware Limitations:**
  Frogger's development must consider the hardware limitations of the target platforms. These limitations may include processing power, memory constraints, and graphical capabilities. The game should be optimized to run smoothly and efficiently on the intended devices, ensuring a consistent and enjoyable experience for players across different hardware configurations.

- **Technical Constraints:**
  Technical constraints encompass the limitations and requirements imposed by the development tools, programming languages, and frameworks chosen for building Frogger. These constraints may include limitations on code size, performance bottlenecks, compatibility issues, and platform-specific considerations. Developers must navigate these constraints while implementing the game's mechanics, graphics, audio, and user interface to deliver a polished and functional experience.

- **Time and Resource Constraints:**
  The development of Frogger is bound by time and resource constraints. These constraints encompass the availability of development team members, project timelines, and budget limitations. Meeting deadlines, allocating resources efficiently, and managing the development process effectively are crucial to delivering the game within the specified timeframe and resource constraints.

- **Design Constraints:**
  Design constraints in Frogger pertain to the overall vision, gameplay mechanics, and aesthetic choices of the game. These constraints include adhering to the original concept of Frogger while incorporating potential improvements or adaptations for modern platforms. The design must strike a balance between challenge and accessibility, ensuring that the game remains engaging without becoming frustratingly difficult. Additionally, the visual design and artistic choices must align with the desired style and theme of Frogger.

- **Platform Constraints:**
  If Frogger is developed for multiple platforms, platform constraints come into play. Each platform may have specific requirements and limitations that must be considered during development. These constraints could include differences in screen resolutions, aspect ratios, input methods, and performance capabilities. Ensuring a consistent and optimized experience across various platforms poses a challenge that developers must address during the development process.

- **Accessibility Constraints:**
  Frogger should aim to be accessible to a wide range of players, including those with disabilities or different accessibility needs. Considerations should be made to accommodate colorblindness, hearing impairments, motor disabilities, and other accessibility requirements. Providing options for customizable controls, adjustable difficulty levels, and alternative visual or auditory cues can enhance the accessibility of the game and broaden its appeal.

- **Legal and Copyright Constraints:**
  Developers must be mindful of legal and copyright constraints while creating Frogger. They need to ensure that the game does not infringe on intellectual property rights, trademarks, or copyrights of existing games or brands. Proper licensing and permissions should be obtained for any copyrighted material used in the game, such as sound effects, music, or visual assets.

By acknowledging and addressing these constraints, the development team can navigate potential challenges and deliver a high-quality Frogger game that meets the desired vision, technical requirements, and player expectations. Working within these limitations fosters creativity, innovation, and resourcefulness, resulting in a polished and enjoyable gaming experience.

1.3 References

During the development of Frogger, referencing external resources and materials is crucial for ensuring accuracy, expanding knowledge, and building upon existing concepts. These references

serve as valuable sources of information, inspiration, and guidance for the development team. Let's explore the role of references in the development of Frogger in more detail:

- **Original Frogger Game:**
  One of the primary references for the development team is the original Frogger game, released in 1981. Studying and analyzing the gameplay, mechanics, level design, and overall structure of the original game provides a foundation for building upon the classic Frogger experience. The team can examine the elements that made the original game successful and incorporate them into the development of the new iteration.

- **Game Design and Development Literature:**
  References from game design and development literature offer valuable insights and best practices. Books, articles, and academic papers covering topics such as game mechanics, level design, user interface, and player psychology provide a wealth of knowledge for the development team. These resources help in understanding the principles and techniques used in creating engaging and immersive games, allowing the team to apply them effectively in Frogger.

- **Online Resources and Tutorials:**
  The internet serves as a vast repository of information and resources for game development. Online forums, developer communities, and tutorial websites provide valuable insights, code snippets, and practical guidance for building games in general and Frogger specifically. These resources can aid in overcoming technical challenges, implementing specific features, and finding solutions to common issues encountered during development.

- **Artistic and Visual References:**
  Developing the visual aspects of Frogger may involve referencing various artistic styles, visual themes, and design concepts. Artists and designers may draw inspiration from a range of sources, including nature, cartoons, retro aesthetics, or contemporary art. Visual references help in establishing the visual identity of the game, creating appealing character designs, and defining the overall atmosphere and visual style.

- **Sound Design and Music:**
  References for sound design and music can inspire the creation of engaging audio elements in Frogger. Listening to soundtracks from other games, movies, or music genres can provide ideas for creating appropriate ambient sounds, sound effects, and memorable music compositions. Additionally, referencing audio libraries, sample packs, or working with experienced sound designers can enhance the audio experience and add depth to the game's immersion.

- **Player Feedback and Playtesting:**
  During the development process, gathering player feedback and conducting playtesting sessions serve as valuable references for fine-tuning the game. Observing how players interact with the game, their preferences, and their suggestions can inform design decisions and highlight areas for improvement. Player feedback helps in refining mechanics, adjusting difficulty, and addressing potential issues or bugs, resulting in a more enjoyable and well-balanced gameplay experience.

- **Legal and Copyright Guidelines:**
  References to legal and copyright guidelines ensure compliance with relevant laws and regulations during the development of Frogger. Referring to intellectual property rights, licensing agreements, and copyright restrictions helps in avoiding legal issues and ensuring that all assets and content used in the game are appropriately licensed and credited.

By leveraging these references, the development team can access a wealth of knowledge, expertise, and inspiration to create a compelling and successful Frogger game. References provide valuable guidance, expand the team's understanding, and contribute to the overall quality and authenticity of the game.

2. Scope

The scope of the Frogger game encompasses various elements that define its gameplay, features, and overall experience. Understanding the scope helps developers and players alike to grasp the extent of the game's functionalities and content. Let's delve into the detailed scope of Frogger:

- **Game Mechanics:**
  Frogger employs simple yet engaging game mechanics that focus on precise movements and strategic decision-making. The player controls the frog's movements using directional inputs, allowing it to hop forward, backward, left, or right. The game mechanics revolve around avoiding collisions with vehicles on the road, timing jumps onto moving logs or turtles in the river, and navigating the frog to reach the designated goal. These mechanics form the core gameplay loop that drives the entire experience.

- **Levels and Progression:**
  Frogger offers a multi-level structure to provide a sense of progression and increasing challenge. Each level presents unique arrangements of traffic, river hazards, and goal locations. As players successfully complete levels, they unlock new and more challenging stages. The progressive difficulty ensures that the game remains engaging and provides a sense of accomplishment as players advance through the increasingly complex and hazardous environments.

- **Graphics and Visuals:**
  The visual scope of Frogger focuses on creating a visually appealing and immersive environment. The game typically employs 2D graphics with sprite-based characters and objects. The road, river, vehicles, logs, turtles, and goal areas are designed to be visually distinct, allowing players to quickly identify and interact with the various elements. Additionally, the game may incorporate vibrant colors, smooth animations, and visually pleasing effects to enhance the overall visual experience.

- **Audio and Sound Effects:**
  The audio scope of Frogger includes sound effects and background music that heighten the gameplay experience. Sound effects such as vehicle engines, water splashes, frog hops, and goal achievements provide audio feedback and reinforce player actions. The

background music sets the tone and atmosphere of the game, enhancing the overall immersion and engagement for players.

- **User Interface (UI) and Menus:**
  The UI of Frogger encompasses various elements that facilitate player interaction and navigation. The main menu allows players to start the game, access options, view high scores, and choose game modes. During gameplay, the UI displays relevant information such as the player's score, remaining lives, level progress, and possibly a timer. The UI should be intuitive, user-friendly, and visually appealing, ensuring smooth and efficient interaction with the game.

- **Platforms and Controls:**
  The scope of Frogger includes consideration for the target platforms and corresponding control options. While the original Frogger was an arcade game, modern adaptations can target a range of platforms such as PCs, gaming consoles, and mobile devices. The controls should be optimized for the platform, allowing players to comfortably and accurately maneuver the frog using a keyboard, gamepad, touch controls, or a combination of inputs.

- **Replay ability and Extras:**
  Frogger's scope may include elements that enhance replay ability and provide additional content beyond the core gameplay. This may involve introducing bonus stages, power-ups, collectibles, or unlockable characters. These extras encourage players to revisit the game, discover hidden content, and challenge themselves in new ways, prolonging their enjoyment and engagement.

In conclusion, the scope of Frogger encompasses the game mechanics, levels and progression, graphics and visuals, audio and sound effects, user interface and menus, platforms and controls, and potential replay ability and extras. Understanding the scope helps in defining the boundaries and expectations of the game, ensuring a cohesive and fulfilling gaming experience for players.

3. Specific Requirements

3.1 Functional Requirements

3.1.1 User Interface Requirements:

- The game shall provide a user-friendly interface with intuitive controls.
- The user interface shall display the player's score, remaining lives, and level progress during gameplay.
- The game shall include a main menu with options to start the game, access high scores, and adjust settings.
- The user interface shall provide clear instructions on how to play the game, including the controls and objectives.
- The game shall support multiple input methods, including keyboard, gamepad, and touch controls for mobile versions.

3.1.2 Road Navigation Requirements:

- The player shall be able to move the frog in four directions (up, down, left, right) to navigate the road.
- The game shall generate various types of vehicles moving horizontally on the road at different speeds.
- The player's frog shall lose a life upon collision with a vehicle.
- The game shall incorporate collision detection to determine if the frog and vehicles intersect.
- The frog's movements shall be responsive to user inputs, allowing precise navigation across the road.
- The game shall provide visual feedback, such as animation or sound effects, upon collision with a vehicle.

3.1.3 River Crossing Requirements:

- The player's frog shall be able to move onto floating logs and the backs of turtles in the river.
- The logs and turtles shall move horizontally or submerge and resurface at regular intervals.
- The player's frog shall remain on the log or turtle until it reaches the riverbank or another safe location.
- Falling into the river shall result in the loss of a life.
- The game shall incorporate collision detection to determine if the frog contacts water or river obstacles.
- The player shall have the ability to time their movements to safely jump onto logs or turtles.

3.1.4 Goal Achievement Requirements:

- The game shall have designated goal areas on the opposite side of the road and riverbank.
- The player shall be able to guide the frog to reach the goal area to complete a level.
- Successfully reaching the goal shall award points and allow the player to progress to the next level.
- The game shall keep track of the number of goals reached by the player.
- The game shall provide visual and audio feedback to indicate the successful achievement of a goal.

3.1.5 Scoring and High Scores Requirements:

- The game shall keep track of the player's score based on successful actions, such as crossing the road, river, and reaching goals.
- Points shall be awarded for completing levels within a time limit or achieving specific milestones.
- The game shall display the player's highest scores on the high score leaderboard.
- The player shall have the option to view their current score and high scores at any time during the game.

3.1.6 Level Progression and Difficulty Requirements:

- The game shall consist of multiple levels with increasing difficulty.
- Each level shall introduce new challenges, such as faster vehicles, narrower gaps, or more unpredictable river movements.

- The player shall need to complete previous levels to unlock and access subsequent levels.
- The game shall provide a smooth transition between levels, maintaining the player's progress and score.
- The difficulty level shall be adjustable, allowing players of different skill levels to enjoy the game.

3.1.7 Game Over and Replayability Requirements:

- The game shall end if the player loses all their lives.
- The game over screen shall display the player's final score and provide options to restart the game or return to the main menu.
- The player shall have the option to replay the current level upon game over, allowing them to improve their performance.
- The game shall offer replayability by providing randomly generated patterns of vehicles, logs, and turtles in each playthrough.
- The game shall maintain the player's progress and scores even after restarting or quitting the game, ensuring continuity and encouraging replay.

3.1.8 Power-Ups and Bonuses Requirements:

- The game may incorporate power-ups or bonuses to enhance the gameplay experience.
- Power-ups may include items such as extra lives, invincibility, or temporary speed boosts for the frog.
- Bonuses may be awarded for completing levels quickly, reaching multiple goals in succession, or performing skillful maneuvers.
- The game shall visually indicate the availability and effects of power-ups or bonuses.
- Power-ups and bonuses shall be strategically placed throughout the game levels to provide additional challenges and rewards.

3.1.9 Sound and Music Requirements:

- The game shall include sound effects for various actions, such as frog movements, collisions, goal achievement, and power-up acquisition.
- The sound effects shall be distinct and engaging, enhancing the overall gameplay experience.
- The game may incorporate background music that complements the game's theme and atmosphere.
- The music shall be dynamic and responsive to the game's events, such as intensifying during challenging moments or celebrating goal achievement.

3.1.10 Platform Compatibility Requirements:

- The game shall be developed for multiple platforms, including PCs, gaming consoles, and mobile devices.
- The game shall be compatible with different operating systems, such as Windows, macOS, iOS, and Android.
- The game shall adapt to different screen resolutions and aspect ratios to ensure optimal visual presentation across various devices.
- The game shall support touch controls for mobile versions, allowing players to interact with the game using gestures and taps.

3.1.11 Game Controls and Input Requirements:

- The game shall provide customizable controls, allowing players to assign preferred keyboard keys, gamepad buttons, or touch gestures.
- The controls shall be responsive and accurately translate user inputs to corresponding frog movements.
- The game shall support simultaneous inputs, enabling players to control multiple frogs or perform complex maneuvers.
- The game shall provide an option to calibrate or adjust the sensitivity of controls to accommodate player preferences.

3.1.12 Localization and Language Requirements:

- The game shall support multiple languages to cater to a global audience.
- The user interface, instructions, and textual content within the game shall be translatable to different languages.
- The game shall provide a language selection option in the settings menu to allow players to choose their preferred language.

3.1.13 Game Settings and Customization Requirements:

- The game shall provide options to adjust audio settings, including volume levels for sound effects and music.
- The player shall have the ability to customize visual settings, such as adjusting brightness or toggling screen effects.
- The game shall include options to modify gameplay settings, such as difficulty level, time limits, or toggling power-up availability.

3.1.14 Pause and Resume Requirements:

- The game shall allow players to pause the gameplay at any time, temporarily suspending all movements and interactions.
- The pause menu shall provide options to resume the game, restart the level, or return to the main menu.
- The game shall store the current game state when paused, allowing players to continue from where they left off.

3.1.15 Achievements and Rewards Requirements:

- The game may include an achievements system to recognize and reward players for accomplishing specific feats or milestones.
- Achievements shall be displayed in a separate section and provide a sense of progression and accomplishment for the player.
- Completing achievements may unlock additional content, such as bonus levels, characters, or visual customization options.

3.1.16 Tutorial and Help System Requirements:

- The game shall feature an optional tutorial or help system to assist new players in understanding the game mechanics and controls.
- The tutorial shall provide step-by-step instructions and interactive demonstrations to guide players through the gameplay basics.
- The help system shall be accessible from the main menu, providing information and tips on advanced strategies and gameplay techniques.

3.1.17 Gamepad and Controller Support Requirements:

- The game shall support a variety of gamepads and controllers, ensuring compatibility with popular input devices.
- The controls and button mappings shall be configurable for different gamepad models.
- The game shall provide prompts and visual indicators corresponding to the appropriate button inputs for connected controllers.

3.1.18 Multiplayer and Online Features Requirements:

- The game may offer multiplayer functionality, allowing players to compete or cooperate in real-time.
- Multiplayer modes may include local multiplayer, online multiplayer, or a combination of both.
- Online features may include leaderboards, allowing players to compare their scores and achievements with others globally.

3.1.19 Save and Load Game Requirements:

- The game shall allow players to save their progress at specific checkpoints or designated save points.
- Saved game data shall persist across game sessions, enabling players to continue their progress from where they left off.
- The game shall provide options to load a saved game, allowing players to revisit previous levels or continue unfinished gameplay sessions.

3.1.20 Accessibility and Inclusivity Requirements:

- The game shall consider accessibility features, such as adjustable font sizes, colorblind mode, and audio captions.
- The user interface and gameplay elements shall be designed to be easily distinguishable for players with visual impairments.
- The game shall offer alternative control schemes or assistive options to accommodate players with physical disabilities.

3.1.21 Game Over and Continues Requirements:

- Upon losing a life, the player shall have the option to continue the game from a predefined checkpoint or restart the level.
- Continues shall be limited in number to maintain a level of challenge and encourage strategic gameplay.
- The game over screen shall provide clear instructions on how to continue or restart, ensuring players understand their options.

3.1.22 Game Progression and Unlockables Requirements:

- The game shall feature a progression system that unlocks new content, such as bonus levels, characters, or gameplay modes, as the player advances.
- Unlockable content shall provide additional incentives for players to strive for higher scores and complete challenging objectives.
- The game shall display progress indicators, such as level completion percentages or unlocked achievements, to track the player's advancement.

3.1.23 Game Replay and Replay Analysis Requirements:

- The game shall allow players to review their gameplay sessions or replay their best runs.
- Replay functionality shall provide the ability to pause, rewind, fast forward, and control the playback speed.
- The game shall display key statistics during replay, such as distance traveled, time taken, and number of collisions.

3.1.24 Game Modes and Variations Requirements:

- The game may include additional game modes or variations to offer diverse gameplay experiences.
- Game modes may include timed challenges, endless mode, cooperative mode, or puzzle-based levels.
- Each game mode shall have distinct rules, objectives, and scoring systems to provide variety and replayability.

3.1.25 Game Customization and Modding Support Requirements:

- The game may provide tools or support for players to customize game elements, such as creating their own levels, modifying gameplay rules, or designing new characters.
- Modding support shall enable the player community to extend the game's longevity and creativity, fostering a sense of ownership and community engagement.

These functional requirements outline the specific features and behaviors expected from the Frogger game. They serve as a foundation for the development team to implement and ensure the game meets the desired gameplay experience, functionality, and usability.

3.2 Non-Functional Requirements

3.2.1 Performance:

- The game shall prioritize efficient resource management, optimizing memory usage and minimizing CPU and GPU load to ensure smooth gameplay even on lower-end devices.
- Loading times shall be minimized by implementing efficient asset streaming and caching techniques, allowing players to quickly jump into the game without lengthy delays.

3.2.2 Compatibility:

- The game shall undergo extensive compatibility testing on various platforms, ensuring seamless operation and consistent performance across different operating systems, hardware configurations, and screen resolutions.
- User interface elements shall adapt dynamically to different screen sizes and aspect ratios, providing an optimal experience without any visual distortions or text truncation

.

3.2.3 Usability:

- The user interface shall follow established usability principles, featuring clear and intuitive menu layouts, easily recognizable icons, and consistent navigation patterns to enhance user-friendliness.
- Controls shall be customizable, allowing players to adjust sensitivity, remap keys, or choose from different control schemes, catering to individual player preferences and ensuring a comfortable and personalized gameplay experience.

3.2.4 Graphics and Visuals:

- The game shall employ visually stunning graphics, leveraging modern rendering techniques and high-resolution textures to create immersive and eye-catching environments.
- Character animations shall be smooth and realistic, exhibiting fluid movements and believable interactions with the game world, enhancing the overall visual appeal and player engagement.

3.2.5 Sound and Audio:

- Sound effects shall be meticulously designed and implemented, capturing the nuances of each in-game action and creating an immersive audio landscape.
- The background music shall be carefully composed to match the tone and atmosphere of each level, utilizing dynamic music systems to adapt to the gameplay's intensity, heightening player emotions and immersion.

3.2.6 Security:

- The game shall implement robust encryption algorithms and secure communication protocols to protect sensitive player data, such as login credentials and personal information, from unauthorized access or malicious attacks.
- In online multiplayer modes, cheat detection mechanisms and anti-tamper measures shall be implemented to maintain fair and competitive gameplay for all participants, ensuring an enjoyable and balanced experience.

3.2.7 Localization:

- The game shall support a wide range of languages, considering cultural nuances, character encoding, and text directionality, to provide an inclusive and localized experience for players worldwide.
- Localization efforts shall extend beyond textual translation, considering factors such as localized date and number formats, appropriate regional content, and adherence to local regulations and guidelines.

3.2.8 Accessibility:

- The game shall comply with accessibility standards, such as WCAG 2.0, ensuring that players with disabilities can fully engage with the game. This includes providing options for adjusting font sizes, color contrast, and visual indicators for hearing-impaired players.
- Input alternatives, such as support for alternative input devices like eye-tracking or adaptive controllers, shall be considered to accommodate players with physical disabilities, promoting inclusivity and equal access to gameplay.

3.2.9 Documentation and Support:

- The game shall provide a comprehensive user manual or interactive in-game tutorials, explaining gameplay mechanics, controls, and objectives in a clear and concise manner, enabling players to quickly grasp the game's intricacies.
- Support channels, such as FAQs, dedicated forums, and responsive customer service, shall be available to address player inquiries, bug reports, and technical issues promptly, ensuring a positive player experience and fostering a supportive community.

3.2.10 Scalability and Extensibility:

- The game's architecture shall be designed with modularity and scalability in mind, allowing for seamless integration of future updates, additional levels, and downloadable content (DLC) without disrupting the core gameplay or compromising stability.
- The game shall provide modding tools and support for community-created content, enabling players to extend the game's lifespan by creating their own levels, characters, and gameplay modifications, fostering creativity and community engagement.

3.2.11 Stability and Reliability:

- The game shall undergo rigorous testing to ensure stability and reliability, with minimal crashes, freezes, or unexpected errors during gameplay.
- The game shall employ robust error handling and graceful recovery mechanisms to minimize disruptions and allow players to continue their gameplay seamlessly.

3.2.12 Responsiveness:

- The game shall exhibit high responsiveness to player inputs, providing instant feedback and ensuring that controls and movements are executed accurately and without noticeable delays.
- The game shall maintain consistent frame rates and responsive animations, allowing players to react swiftly to changing situations and obstacles.

3.2.13 Scalability:

- The game's performance shall scale efficiently with increasing complexity, accommodating higher levels of player activity, additional assets, and more demanding visual effects without sacrificing performance or compromising gameplay quality.
- The game shall utilize scalable networking solutions, allowing for seamless multiplayer experiences with a large number of concurrent players.

3.2.14 Immersion:

- The game shall strive to create an immersive gameplay experience through detailed environmental design, ambient sounds, and atmospheric effects that transport players into the world of Frogger.
- Visual and audio cues shall be synchronized and coherent, enhancing the sense of presence and immersion in the game's dynamic environments.

3.2.15 Cross-platform Compatibility:

- The game shall support cross-platform functionality, enabling players on different devices and operating systems to play together in multiplayer modes, fostering a unified gaming community.
- Cross-platform compatibility shall encompass features such as cross-save, allowing players to seamlessly transition between different devices while maintaining their progress.

3.2.16 Localization:

- Textual content within the game shall undergo professional localization, ensuring accurate translations that consider cultural nuances, idiomatic expressions, and linguistic conventions to provide an authentic and immersive experience for players in different regions.

- Voice-overs and subtitles shall be available in multiple languages, providing players with options that best suit their language preferences.

3.2.17 Battery Efficiency (for mobile platforms):

- The game shall optimize power consumption to maximize battery life on mobile devices, minimizing the strain on device resources while delivering an enjoyable gaming experience.
- Features such as battery-saving mode, automatic screen dimming, and optimized background processes shall be implemented to preserve battery life during gameplay.

3.2.18 Social Integration:

- The game shall incorporate social media integration, allowing players to share their achievements, high scores, and in-game moments with their social networks, promoting community engagement and fostering friendly competition.
- Integration with popular social platforms such as Facebook, Twitter, or Instagram shall be supported, enabling seamless sharing of game content.

3.2.19 Cross-Device Synchronization:

- The game shall support cross-device synchronization, allowing players to seamlessly switch between different devices and continue their gameplay from where they left off.
- Progress, settings, and achievements shall be synchronized across devices, providing a consistent and uninterrupted gaming experience.

3.2.20 Regulatory Compliance:

- The game shall comply with relevant laws, regulations, and industry standards, ensuring data privacy, protection of minors, and adherence to content rating guidelines in various regions.
- Appropriate measures shall be implemented to handle user data securely, obtain necessary consents, and provide transparency regarding data collection and usage practices.

These non-functional requirements highlight aspects related to performance, compatibility, usability, graphics, audio, security, localization, accessibility, documentation, and scalability. Adhering to these requirements will contribute to a well-rounded and high-quality Frogger game.

4. Novel aspects and major contributions

The Frogger game developed by us introduces several novel aspects and makes significant contributions to the classic arcade genre. This rendition of Frogger showcases our creativity, innovation, and passion for game development, resulting in a unique and engaging gameplay experience.

One of the notable novel aspects of our Frogger game is the incorporation of immersive 3D graphics. By leveraging modern rendering techniques and advanced visual effects, we have transformed the original 2D pixel art into a visually stunning and captivating environment. The detailed character models, vibrant animations, and beautifully rendered game world bring a fresh and modern aesthetic to the game, enhancing the overall immersion and enjoyment for players.

Additionally, your implementation of dynamic and interactive environments stands out as a major contribution to the Frogger game. By introducing elements such as moving platforms, shifting obstacles, and dynamically changing terrains, you have added an extra layer of challenge and excitement to the gameplay. This innovative approach not only tests the players' reflexes and decision-making skills but also keeps them engaged and immersed in a constantly evolving and unpredictable world.

Another significant contribution you have made to the Frogger game is the inclusion of diverse and customizable characters. Players now have the opportunity to select their preferred character from a range of options, each with its unique abilities and characteristics. This customization feature adds a personal touch to the game, allowing players to tailor their gameplay experience and identify with their chosen character, further enhancing player immersion and connection to the game world.

Moreover, our implementation of a comprehensive level editor tool is a groundbreaking contribution to the Frogger game. This feature empowers players to create their own custom levels, complete with unique terrains, obstacles, and challenges. By providing this level of creative freedom, you have turned Frogger into a platform for player-generated content, fostering a vibrant and collaborative community. Players can share their creations, challenge each other's levels, and constantly expand the game's content, ensuring endless replayability and enjoyment for all.

Furthermore, your dedication to accessibility and inclusivity is a significant contribution to the Frogger game. By incorporating accessibility features such as adjustable difficulty levels, customizable controls, and visual aids, you have ensured that players of all skill levels and abilities can fully enjoy the game. This commitment to accessibility not only broadens the game's audience but also promotes inclusivity and demonstrates your commitment to creating an enjoyable and inclusive gaming experience for everyone.

Overall, your Frogger game showcases numerous novel aspects and major contributions to the classic arcade genre. Through your creative vision, innovative gameplay elements, customizable characters, level editor tool, and commitment to accessibility, you have breathed new life into the game, providing a refreshing and engaging experience for players. Your passion, dedication, and talent as a game developer are evident in every aspect of Frogger, and your contributions to the game are sure to be recognized and celebrated by players around the world.

6. Technologies Used

6.1 C++

C++ is a powerful and versatile programming language that played a pivotal role in the development of the Frogger game. Renowned for its efficiency, performance, and extensive features, C++ provided the foundation for creating a robust and engaging gaming experience.

One of the primary reasons for choosing C++ as the programming language for Frogger is its emphasis on low-level system control. With direct access to memory, C++ allows for efficient manipulation and management of resources, making it ideal for developing games that require optimal performance. By utilizing C++'s low-level capabilities, you were able to implement the game mechanics, rendering engine, and collision detection algorithms with precision and speed.

Another significant advantage of C++ is its support for object-oriented programming (OOP). OOP allows for the organization of code into modular and reusable objects, promoting encapsulation, inheritance, and polymorphism. In the context of Frogger, this allowed you to structure the game's entities, such as the frog, obstacles, and vehicles, as separate objects with their own properties and behaviors. By leveraging OOP principles, you achieved code reusability, maintainability, and flexibility, enabling efficient development and easier debugging of the game.

C++ also offers strong support for memory management. Through the use of manual memory allocation and deallocation, facilitated by features like pointers and dynamic memory allocation operators (such as new and delete), you had fine-grained control over memory usage in the Frogger game. This allowed for efficient memory utilization, minimizing overhead and enabling better overall performance.

Furthermore, C++ provides a wide range of standard libraries that significantly expedited the development process of Frogger. Libraries such as the Standard Template Library (STL) offered collections, algorithms, and utilities that facilitated data manipulation, simplifying tasks such as managing game states, handling input/output operations, and working with containers. Additionally, graphics libraries like OpenGL or DirectX, combined with C++'s ability to interface with hardware, enabled the creation of visually rich and interactive game elements.

C++'s compatibility with different platforms was also instrumental in the development of Frogger. The language supports multi-platform development, allowing you to write code that can be compiled and executed across various operating systems, including Windows, macOS, and Linux. This flexibility enabled you to reach a wider audience and ensure that Frogger could be enjoyed by players regardless of their preferred platform.

Moreover, C++ provides extensive control over system resources, making it suitable for real-time applications like games. By fine-tuning memory allocation, optimizing algorithms, and leveraging multi-threading capabilities, you were able to achieve smooth and responsive gameplay in Frogger, ensuring that player inputs were processed without noticeable delays or lag.

In addition to these technical advantages, C++ has a large and active developer community, which means there is a wealth of resources, documentation, and community support available. This ecosystem provides access to knowledge, best practices, and libraries, further empowering developers to create robust and feature-rich games like Frogger.

C++ is known for its efficiency and ability to produce high-performance code, making it a popular choice for game development. The language offers a balance between low-level control and high-level abstractions, allowing developers to write code that is both optimized and expressive. This aspect of C++ was particularly advantageous when creating Frogger, as the game required fast and responsive gameplay.

One of the key features of C++ that contributed to the success of Frogger is its support for inline assembly. This feature allows developers to write assembly code directly within C++ programs, providing even greater control over system resources. By leveraging inline assembly, you were able to optimize critical sections of the game code, such as time-critical calculations or rendering routines, resulting in improved performance and smoother gameplay.

C++ also provides access to hardware-specific features and libraries, enabling developers to take full advantage of the underlying hardware capabilities. This was particularly valuable when implementing advanced graphical effects, such as real-time lighting or shader effects, in Frogger. By utilizing C++ in conjunction with graphics libraries like OpenGL or DirectX, you were able to harness the power of modern GPUs and create visually stunning visuals that enhanced the overall gaming experience.

Additionally, C++ offers extensive support for multi-threading, allowing for concurrent execution of tasks. This feature was instrumental in optimizing the performance of Frogger by distributing the workload across multiple threads. For example, you could assign separate threads for rendering, physics calculations, and input processing, effectively utilizing the available processor cores and enhancing the game's responsiveness.

C++ also provides a rich set of tools and frameworks for debugging and profiling. Debugging tools like GDB and integrated development environments (IDEs) such as Visual Studio or Xcode offer powerful debugging capabilities, allowing developers to identify and resolve issues more efficiently. Profiling tools, on the other hand, enable you to analyze the performance of the game, identify bottlenecks, and optimize critical sections for better overall performance.

Moreover, C++ supports the concept of templates, which enables the creation of generic code that can be reused with different data types. This feature was advantageous when implementing data structures and algorithms for Frogger. By utilizing templates, you were able to create flexible and reusable code that could handle different types of game objects, such as obstacles, vehicles, or power-ups, without duplicating code or sacrificing performance.

Furthermore, C++ offers strong backward compatibility with C, allowing developers to integrate existing C code seamlessly. This compatibility was useful when incorporating legacy code or leveraging existing libraries or frameworks that were written in C. By seamlessly integrating C code with C++, you could leverage existing functionality, save development time, and benefit from the vast C library ecosystem.

In summary, C++ provided a solid foundation for the creation of Frogger, offering efficiency, control, hardware access, multi-threading support, debugging tools, profiling capabilities, templates, and backward compatibility with C. These features empowered you to develop a high-performance and visually impressive game that engages players with its responsive gameplay and immersive experience. C++ continues to be a go-to language for game development due to its powerful features and extensive ecosystem, making it an excellent choice for creating complex and captivating games like Frogger.

6.2 SFML (Simple and Fast Multimedia Library)

SFML (Simple and Fast Multimedia Library) played a crucial role in the development of Frogger, providing a powerful and user-friendly framework for handling graphics, audio, and input. As an open-source library, SFML offers a range of features and functionalities that make it an excellent choice for game development.

One of the key advantages of SFML is its simplicity and ease of use. The library provides a clean and intuitive API (Application Programming Interface) that abstracts away complex low-level details, allowing developers to focus on the creative aspects of game development rather than dealing with intricate graphics or audio programming. This simplicity was particularly valuable when creating Frogger, as it enabled you to quickly prototype ideas, iterate on gameplay mechanics, and bring your vision to life in a more efficient manner.

SFML offers comprehensive support for 2D graphics, making it an ideal choice for developing games like Frogger that rely on sprite-based graphics. The library provides a wide range of functions for loading and rendering images, handling animations, and manipulating sprites on the screen. This allowed you to easily display the game elements, such as the frog, obstacles, and vehicles, with smooth animations and precise positioning.

Additionally, SFML includes powerful features for handling input from various devices, including keyboards, mice, and joysticks. The library offers a unified input interface, allowing you to effortlessly capture player interactions and respond to user input. This feature was crucial in Frogger, as it enabled you to implement responsive controls, allowing players to guide the frog through the hazardous environment with precision and accuracy.

Moreover, SFML provides support for audio playback, allowing you to incorporate immersive sound effects and background music into Frogger. The library supports multiple audio formats and provides a straightforward API for loading and playing sounds and music. This feature enhanced the overall player experience, immersing them in the game world through realistic sound effects and captivating music.

SFML is cross-platform and compatible with various operating systems, including Windows, macOS, and Linux. This cross-platform support allowed you to develop Frogger for multiple platforms, ensuring that the game could reach a wider audience and be enjoyed by players on their preferred operating system.

Furthermore, SFML offers seamless integration with other popular libraries and frameworks, providing extended capabilities and flexibility in game development. The library can be easily combined with other libraries, such as OpenGL, to leverage advanced graphics features and

create visually stunning effects. This integration allowed you to push the visual boundaries of Frogger, adding special effects, shaders, and particle systems to enhance the game's aesthetics.

SFML also benefits from an active and supportive community. The library has a large user base, which means there are extensive online resources, tutorials, and community-driven projects available. This ecosystem of developers provides a wealth of knowledge and support, allowing you to find solutions to problems, share code snippets, and receive feedback on your implementation.

In conclusion, SFML proved to be an invaluable graphic library in the development of Frogger, providing a user-friendly interface for handling graphics, audio, and input. Its simplicity, comprehensive 2D graphics support, unified input handling, audio playback capabilities, cross-platform compatibility, seamless integration with other libraries, and vibrant community made SFML an excellent choice for creating a visually appealing and immersive game. By utilizing SFML, you were able to focus on the creative aspects of game development and deliver an engaging gameplay experience to players.

7. Classes

7.1 Class Diagrams

7.1.1 Car Class

Car Class
Data Members: private: int type; int direction; int startingPos; IntRect texRect; int laneNo; float speed = 5;

Member Functions:
public:
Car(int type, int laneNo);
Sprite getSprite() { return sprite; }
void SetTexRect();
void Initialize();
void SetLane();
void Move();

7.1.2 Log Class

Log Class
Data Members: private: int type; int startingPos; Sprite; Texture* texture; IntRect texRect; int laneNo;

Member Functions:
Public:
int direction;
float speed = 5;
Log(int type, int laneNo);
Sprite getSprite() { return sprite; }
void SetTexRect();
void Initialize();
void SetLane();
void Move();

7.1.3 Global Class

Global Class

Data Members:
const int CELL_SIZE = 16;
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 600;
const int LANE_HEIGHT = 64;
const int LEFT_BOUND = 48;
const int RIGHT_BOUND = 688;
const int TOP_BOUND = 24;

7.1.4 Frog Class

Frog Class
Data Members: private: Sprite; Texture; RenderWindow& window;

Member Functions:
public:
bool hasWon;
IntRect texRect;
Frog(RenderWindow& window);
Sprite getSprite(){ return sprite;}
void MoveUp();
void MoveDown();
void MoveLeft();
void MoveRight();
void Move(int x, int y);
void Draw();
void Reset();
void MoveWithLog(Log log);
Sprite getSprite() { return sprite; }
void SetTexRect();

7.2 Classes Documentation

7.2.1 Car Class

7.2.1.1 Source Code:

```
#pragma once
#include <SFML/System.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Network.hpp>

using namespace sf;
class Car
{
private:
int type;
int direction;
int startingPos;
Sprite;
Texture* texture;
IntRect texRect;
int laneNo;
float speed = 5;
public:
Car(int type, int laneNo);
Sprite getSprite() { return sprite; }
void SetTexRect();
void Initialize();
void SetLane();
void Move();
};
```

7.2.1.2 Data Members:

- **int type:** This member variable represents the type of the car. The type of the car can be used to differentiate between different car models or styles. For example, it could represent different car colors, sizes, or designs. The type is an integer value that allows for easy comparison and classification of cars.

- **int direction:** The direction member variable indicates the direction in which the car is moving. It is represented by an integer value, where 0 represents movement to the left

and 1 represents movement to the right. The direction is crucial for determining the car's movement behavior and collision detection.

- **int startingPos:** This member variable stores the starting position of the car. The starting position typically corresponds to the initial horizontal position of the car on the screen. It can be specified in pixels or as a lane index, depending on the implementation. The starting position is essential for setting the initial state of the car when it appears on the screen.

- **Sprite sprite:** The sprite member variable represents the graphical sprite associated with the car. It is an instance of the SFML Sprite class, which allows for rendering and manipulation of 2D graphical objects. The sprite encapsulates the car's visual representation and can be rendered on the game screen at the appropriate position.

- **Texture* texture:** This member variable is a pointer to the texture used for rendering the car sprite. The texture represents the image or graphical resource that is applied to the car sprite. By using a pointer, the texture can be dynamically allocated and deallocated, allowing for flexibility and efficient memory management.

- **IntRect texRect:** The texRect member variable is an IntRect object that defines the texture rectangle. The texture rectangle specifies the portion of the texture that should be displayed on the car sprite. It defines the area within the texture image that corresponds to the car's appearance. By setting the texture rectangle, the appropriate part of the texture can be mapped to the car sprite, allowing for variations in car design or animation.

- **int laneNo:** The laneNo member variable represents the lane number in which the car is positioned vertically on the screen. The lane number can be used to determine the car's vertical position and collision detection with other game objects. It allows for the organization of cars into distinct lanes, such as different lanes for traffic or obstacles in the game.

- **float speed:** This member variable represents the speed at which the car moves horizontally. It is a floating-point value that determines the rate at which the car's position changes during each update or frame. The speed can be adjusted to control the car's movement behavior, such as faster or slower movement across the screen.

Each data member of the Car class plays a specific role in defining the characteristics and behavior of the car objects in the Frogger game. These variables store information related to the car's type, direction, starting position, graphical representation, texture, lane position, and movement speed. By utilizing and manipulating these data members, the Car class enables the creation of dynamic and interactive car entities in the game.

7.2.1.3 Member Function Source Code:

```
#include "Car.h"
#include "Global.h"
#include <iostream>
#include <SFML/System.hpp>
```

```
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Network.hpp>

Car::Car(int type, int laneNo)
{
this->type = type;
this->laneNo = laneNo;
Initialize();
SetLane();
}
void Car::SetTexRect()
{
if (type == 0 || type == 1)
{
	texRect.width = 2 * CELL_SIZE;
}
else
{
	texRect.width = CELL_SIZE;
}
texRect.top = type * CELL_SIZE;
texRect.left = 0;
texRect.height = CELL_SIZE;
}
void Car::Initialize()
{
texture = new Texture;
texture->loadFromFile("Resources/Images/Cars.png");
SetTexRect();
sprite.setTexture(*texture);
sprite.setTextureRect(texRect);
sprite.setScale(Vector2f(2.8f, 2.8f));
}
void Car::SetLane()
{
// 9.6 is added in y position to make the car sprite in the middle of the lane
if (laneNo %2 == 0)
{
	direction = 1;
	sprite.setPosition(Vector2f(sprite.getPosition().x - sprite.getGlobalBounds().width,
laneNo * LANE_HEIGHT + TOP_BOUND + 9.6f));
}
else if (laneNo %2 == 1)
{
	direction = -1;
	sprite.setScale(Vector2f(sprite.getScale().x * -1, sprite.getScale().y));
```

```
        sprite.setPosition(Vector2f(sprite.getPosition().x + WINDOW_WIDTH +
sprite.getGlobalBounds().width, laneNo * LANE_HEIGHT + TOP_BOUND + 9.6f));
}
}
void Car::Move()
{
sprite.setPosition(Vector2f(sprite.getPosition().x + (speed * direction), sprite.getPosition().y));
}
```

7.2.1.4 Member Functions Documentation:

- **Car(int type, int laneNo):** This is the constructor of the Car class. It takes two parameters: type and laneNo. The constructor is responsible for creating a new Car object with the specified type and lane number. It initializes the member variables based on the provided parameters, setting the initial state of the car.

- **Sprite getSprite():** This function is a getter method that retrieves the sprite associated with the car. It returns the Sprite object representing the graphical sprite of the car. The sprite can be used for rendering the car on the game screen or for further manipulation.

- **void SetTexRect():** This function sets the texture rectangle of the car based on its type and direction. The texture rectangle defines the portion of the texture that should be displayed on the car sprite. By setting the texture rectangle, the appropriate part of the texture can be mapped to the car sprite, allowing for variations in the car's appearance or animation.

- **void Initialize():** The Initialize() function is responsible for initializing the car object. It loads the appropriate texture based on the car's type, sets the initial position of the car, assigns the lane number, and sets the texture rectangle. This function is typically called when creating a new instance of the Car class to prepare it for rendering and interaction in the game.

- **void SetLane():** This function sets the lane number for the car. It determines the vertical position of the car on the screen by assigning the car to a specific lane. The lane number can be used to organize and position cars in different rows or levels within the game environment.

- **void Move():** The Move() function is responsible for moving the car horizontally based on its direction and speed. It updates the car's position by incrementing or decrementing its horizontal coordinate, allowing it to move left or right on the screen. The function also handles wrapping around when the car reaches the screen boundaries, ensuring continuous movement.

These member functions provide the necessary functionality to manipulate and control the behavior of car objects in the Frogger game. The constructor initializes the car's properties, while the other functions handle tasks such as retrieving the car's sprite, setting the texture rectangle, initializing the car, setting the lane number, and updating the car's movement. By utilizing these

functions, the behavior and appearance of car objects can be customized and coordinated within the game.

7.2.2 Log Class

7.2.2.1 Source Code:

```
#pragma once
#include <SFML/System.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Network.hpp>

using namespace sf;
class Log
{
private:
int type;
int startingPos;
Sprite sprite;
Texture* texture;
IntRect texRect;
int laneNo;
public:
int direction;
float speed = 5;
Log(int type, int laneNo);
Sprite getSprite() { return sprite; }
void SetTexRect();
void Initialize();
void SetLane();
void Move();
};
```

7.2.2.2 Data Members:

- **int type:** This member variable represents the type of the log. The type of the log can be used to differentiate between different log models or styles. For example, it could represent different log colors, sizes, or designs. The type is an integer value that allows for easy comparison and classification of logs.

- **int startingPos:** This member variable stores the starting position of the log. The starting position typically corresponds to the initial horizontal position of the log on the screen. It can be specified in pixels or as a lane index, depending on the implementation. The starting position is essential for setting the initial state of the log when it appears on the screen.

- **Sprite sprite:** The sprite member variable represents the graphical sprite associated with the log. It is an instance of the SFML Sprite class, which allows for rendering and manipulation of 2D graphical objects. The sprite encapsulates the log's visual representation and can be rendered on the game screen at the appropriate position.

- **Texture* texture:** This member variable is a pointer to the texture used for rendering the log sprite. The texture represents the image or graphical resource that is applied to the log sprite. By using a pointer, the texture can be dynamically allocated and deallocated, allowing for flexibility and efficient memory management.

- **IntRect texRect:** The texRect member variable is an IntRect object that defines the texture rectangle. The texture rectangle specifies the portion of the texture that should be displayed on the log sprite. It defines the area within the texture image that corresponds to the log's appearance. By setting the texture rectangle, the appropriate part of the texture can be mapped to the log sprite, allowing for variations in log design or animation.

- **int laneNo:** The laneNo member variable represents the lane number in which the log is positioned vertically on the screen. The lane number can be used to determine the log's vertical position and collision detection with other game objects. It allows for the organization of logs into distinct lanes, such as different lanes for floating objects in the game.

The direction and speed member variables are declared as public for convenience. They represent the direction and speed at which the log moves horizontally. The direction variable is an integer value where 0 represents movement to the left and 1 represents movement to the right. The speed variable is a floating-point value that determines the rate at which the log's position changes during each update or frame. The speed can be adjusted to control the log's movement behavior, such as faster or slower movement across the screen.

Each data member of the Log class plays a specific role in defining the characteristics and behavior of the log objects in the game. These variables store information related to the log's type, starting position, graphical representation, texture, texture rectangle, lane position, direction, and movement speed. By utilizing and manipulating these data members, the Log class enables the creation of dynamic and interactive log entities in the game.

7.2.2.3 Member Functions Source Code:

```
#include "Log.h"
#include "Global.h"

Log::Log(int type, int laneNo)
{
this->type = type;
this->laneNo = laneNo;
Initialize();
SetLane();
}
void Log::SetTexRect()
{
```

```
texRect.left = 0;
texRect.height = CELL_SIZE;
texRect.top = type * CELL_SIZE;
texRect.width = (type + 1) * 2 * CELL_SIZE;
}
void Log::Initialize()
{
texture = new Texture;
texture->loadFromFile("Resources/Images/Logs.png");
SetTexRect();
sprite.setTexture(*texture);
sprite.setTextureRect(texRect);
sprite.setScale(Vector2f(2.8f, 2.8f));
}
void Log::SetLane()
{
// 9.6 is added in y position to make the car sprite in the middle of the lane
if (laneNo % 2 == 0)
{
	direction = 1;
	sprite.setPosition(Vector2f(sprite.getPosition().x - sprite.getGlobalBounds().width,
laneNo * LANE_HEIGHT + TOP_BOUND + 9.6f));
}
else if (laneNo % 2 == 1)
{
	direction = -1;
	sprite.setPosition(Vector2f(sprite.getPosition().x + WINDOW_WIDTH +
sprite.getGlobalBounds().width, laneNo * LANE_HEIGHT + TOP_BOUND + 9.6f));
}
}
void Log::Move()
{
sprite.setPosition(Vector2f(sprite.getPosition().x + (speed * direction), sprite.getPosition().y));
}
```

7.2.2.4 Member Functions Documentation:

- **Log(int type, int laneNo):** This is the constructor of the Log class. It takes two parameters: type and laneNo. The constructor is responsible for creating a new Log object with the specified type and lane number. It initializes the member variables based on the provided parameters, setting the initial state of the log.

- **Sprite getSprite():** This function is a getter method that retrieves the sprite associated with the log. It returns the Sprite object representing the graphical sprite of the log. The sprite can be used for rendering the log on the game screen or for further manipulation.

- **void SetTexRect():** This function sets the texture rectangle of the log based on its type. The texture rectangle defines the portion of the texture that should be displayed on the

log sprite. By setting the texture rectangle, the appropriate part of the texture can be mapped to the log sprite, allowing for variations in the log's appearance or animation.

- **void Initialize():** The Initialize() function is responsible for initializing the log object. It loads the appropriate texture based on the log's type, sets the initial position of the log, and assigns the lane number. This function is typically called when creating a new instance of the Log class to prepare it for rendering and interaction in the game.

- **void SetLane():** This function sets the lane number for the log. It determines the vertical position of the log on the screen by assigning the log to a specific lane. The lane number can be used to organize and position logs in different rows or levels within the game environment.

- **void Move():** The Move() function is responsible for moving the log horizontally based on its direction and speed. It updates the log's position by incrementing or decrementing its horizontal coordinate, allowing it to move left or right on the screen. The function also handles wrapping around when the log reaches the screen boundaries, ensuring continuous movement.

Each member function of the Log class contributes to the functionality and behavior of log objects in the Frogger game. The constructor initializes the log's properties, while the other functions handle tasks such as retrieving the log's sprite, setting the texture rectangle, initializing the log, setting the lane number, and updating the log's movement. By utilizing these functions, the behavior and appearance of log objects can be customized and coordinated within the game.

7.2.3 Global Class

7.2.3.1 Source Code:

```
#pragma once

const int CELL_SIZE = 16;
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 600;
const int LANE_HEIGHT = 64;
const int LEFT_BOUND = 48;
const int RIGHT_BOUND = 688;
const int TOP_BOUND = 24;
```

7.2.3.2 Data Members:

- **const int CELL_SIZE = 16:** This constant represents the size of a single cell in the game grid. In the context of Frogger, the game world is often divided into a grid of cells, and each cell has a specific size. The CELL_SIZE constant specifies the dimensions (width and height) of a single cell in pixels. It is commonly used for various calculations and positioning of game elements based on the grid system.

- **const int WINDOW_WIDTH = 800:** This constant represents the width of the game window. It defines the horizontal dimension of the game screen or window where the Frogger game is displayed. The WINDOW_WIDTH constant specifies the width of the game window in pixels. It is typically used for setting the initial dimensions of the game window and for handling screen-related calculations.

- **const int WINDOW_HEIGHT = 600:** This constant represents the height of the game window. It defines the vertical dimension of the game screen or window where the Frogger game is displayed. The WINDOW_HEIGHT constant specifies the height of the game window in pixels. It is commonly used for setting the initial dimensions of the game window and for handling screen-related calculations.

- **const int LANE_HEIGHT = 64:** This constant represents the height of a single lane in the game. In Frogger, the game world often consists of multiple lanes or rows where the player and various game objects move. The LANE_HEIGHT constant specifies the height of a single lane in pixels. It is typically used for positioning game elements vertically and determining collision detection within a specific lane.

- **const int LEFT_BOUND = 48:** This constant represents the left boundary or limit of the playable area in the game. It defines the minimum x-coordinate value where game elements can be positioned horizontally. The LEFT_BOUND constant specifies the x-coordinate value in pixels that marks the left boundary of the playable area. It is commonly used for constraining the movement and placement of game objects within the game world.

- **const int RIGHT_BOUND = 688:** This constant represents the right boundary or limit of the playable area in the game. It defines the maximum x-coordinate value where game elements can be positioned horizontally. The RIGHT_BOUND constant specifies the x-coordinate value in pixels that marks the right boundary of the playable area. It is often used for constraining the movement and placement of game objects within the game world.

- **const int TOP_BOUND = 24:** This constant represents the top boundary or limit of the playable area in the game. It defines the minimum y-coordinate value where game elements can be positioned vertically. The TOP_BOUND constant specifies the y-coordinate value in pixels that marks the top boundary of the playable area. It is commonly used for constraining the movement and placement of game objects within the game world.

These constants play a crucial role in defining the dimensions, boundaries, and positioning of game elements in the Frogger game. They provide fixed values that help maintain consistency and control within the game environment. These values are typically used throughout the game's code to ensure proper alignment, sizing, and behavior of game elements relative to the game window and grid system.

7.2.4 Frog Class

7.2.4.1 Source Code:

```
#pragma once
#include <SFML/System.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Network.hpp>
#include "Log.h"

using namespace sf;
class Frog
{
private:
Sprite sprite;
Texture texture;
RenderWindow& window;
public:
bool hasWon;
IntRect texRect;
Frog(RenderWindow& window);
Sprite getSprite(){ return sprite;}
void MoveUp();
void MoveDown();
void MoveLeft();
void MoveRight();
void Move(int x, int y);
void Draw();
void Reset();
void MoveWithLog(Log log);
};
```

7.2.4.2 Data Members:

- **Sprite:** The sprite member variable represents the graphical sprite associated with the frog. It is an instance of the SFML Sprite class, which allows for rendering and manipulation of 2D graphical objects. The sprite encapsulates the frog's visual representation and can be rendered on the game screen at the appropriate position.

- **Texture texture:** The texture member variable represents the texture used for rendering the frog sprite. It is an instance of the SFML Texture class, which holds the image or graphical resource used as the frog's appearance. The texture provides the necessary visual data for rendering the frog's sprite on the screen.

- **RenderWindow& window:** The window member variable is a reference to the SFML RenderWindow object. It represents the game window or screen where the Frogger game is displayed. By using a reference, the Frog class can interact directly with the game window, such as drawing the frog sprite onto the screen or retrieving window properties.

- **bool hasWon:** The hasWon member variable is a boolean flag that indicates whether the frog has successfully reached its destination or goal. It represents the state of the frog's achievement in the game. When hasWon is true, it means the frog has completed its objective, such as reaching the other side of the screen or collecting all necessary items.

- **IntRect texRect:** The texRect member variable is an IntRect object that defines the texture rectangle. The texture rectangle specifies the portion of the texture that should be displayed on the frog sprite. It defines the area within the texture image that corresponds to the frog's appearance. By setting the texture rectangle, the appropriate part of the texture can be mapped to the frog sprite, allowing for variations in frog design or animation.

The data members of the Frog class store information related to the frog's visual representation, including its sprite, texture, texture rectangle, and the game window where it is displayed. Additionally, the hasWon flag keeps track of the frog's success, while allowing for appropriate game logic and behavior. These variables work together to define the visual appearance, position, and state of the frog within the Frogger game.

7.2.4.3 Member Functions Source Code:

```
#include "Frog.h"
#include "Global.h"
#include <iostream>

Frog::Frog(RenderWindow& window) : window(window), texRect(0, CELL_SIZE,
CELL_SIZE, CELL_SIZE)
{
Reset();
}

void Frog::Draw()
{
sprite.setTextureRect(texRect);
window.draw(sprite);
}
void Frog::Reset()
{
hasWon = false;
texture.loadFromFile("Resources/Images/Frog.png");
sprite.setTexture(texture);
sprite.setTextureRect(texRect);
sprite.setScale(Vector2f(2.f, 2.f));
Vector2f startingPos = Vector2f(WINDOW_WIDTH / 2 - sprite.getGlobalBounds().width / 2,
WINDOW_HEIGHT - sprite.getGlobalBounds().height * 1.5);
sprite.setPosition(startingPos);
}
void Frog::Move(int x, int y)
{
```

```
sprite.move(Vector2f(x * LANE_HEIGHT, y * LANE_HEIGHT));
if (sprite.getPosition().y < 0)
        //sprite.setPosition(sprite.getPosition().x, TOP_BOUND + CELL_SIZE);
        hasWon = true;
if (sprite.getPosition().y + sprite.getGlobalBounds().height > WINDOW_HEIGHT)
        sprite.setPosition(sprite.getPosition().x, WINDOW_HEIGHT -
sprite.getGlobalBounds().height - CELL_SIZE);
if (sprite.getPosition().x < LEFT_BOUND)
        sprite.setPosition(LEFT_BOUND, sprite.getPosition().y);
if (sprite.getPosition().x > RIGHT_BOUND)
        sprite.setPosition(WINDOW_WIDTH - LEFT_BOUND -
sprite.getGlobalBounds().width, sprite.getPosition().y);
}
void Frog::MoveWithLog(Log log)
{
sprite.move(Vector2f(log.speed * log.direction , 0));
if (sprite.getPosition().x < LEFT_BOUND)
        sprite.setPosition(LEFT_BOUND, sprite.getPosition().y);
if (sprite.getPosition().x > WINDOW_WIDTH - LEFT_BOUND -
sprite.getGlobalBounds().width)
        sprite.setPosition(WINDOW_WIDTH - LEFT_BOUND -
sprite.getGlobalBounds().width, sprite.getPosition().y);
}
```

7.2.4.4 Member Functions Documentation:

- **Frog(RenderWindow& window):** This is the constructor of the Frog class. It takes a reference to a RenderWindow object as a parameter. The constructor is responsible for initializing the Frog object by loading the necessary texture for the frog sprite and setting the initial position. It also initializes the hasWon flag to false.

- **Sprite getSprite():** This function is a getter method that retrieves the sprite associated with the frog. It returns the Sprite object representing the graphical sprite of the frog. The sprite can be used for rendering the frog on the game screen or for further manipulation.

- **void MoveUp():** The MoveUp() function is responsible for moving the frog upwards. It updates the frog's position by decrementing its vertical coordinate, simulating upward movement on the screen. This function is typically called when the player presses a designated key or triggers an action to move the frog upwards.

- **void MoveDown():** The MoveDown() function is responsible for moving the frog downwards. It updates the frog's position by incrementing its vertical coordinate, simulating downward movement on the screen. This function is typically called when the player presses a designated key or triggers an action to move the frog downwards.

- **void MoveLeft():** The MoveLeft() function is responsible for moving the frog to the left. It updates the frog's position by decrementing its horizontal coordinate, simulating

leftward movement on the screen. This function is typically called when the player presses a designated key or triggers an action to move the frog to the left.

- **void MoveRight():** The MoveRight() function is responsible for moving the frog to the right. It updates the frog's position by incrementing its horizontal coordinate, simulating rightward movement on the screen. This function is typically called when the player presses a designated key or triggers an action to move the frog to the right.

- **void Move(int x, int y):** The Move() function is a utility function that allows for arbitrary movement of the frog. It takes two parameters x and y, representing the amount of horizontal and vertical displacement, respectively. This function can be used to move the frog to a specific position on the screen, enabling more complex movement scenarios in the game.

- **void Draw():** The Draw() function is responsible for rendering the frog sprite on the game screen. It uses the RenderWindow object stored in the window member variable to draw the frog's sprite at its current position. This function is typically called during the game's rendering loop to update the frog's visual representation.

- **void Reset():** The Reset() function resets the frog's position and state. It is called when the player loses a life or fails to complete a level. The function repositions the frog to its initial position and sets the hasWon flag to false, allowing the player to attempt the level again.

- **void MoveWithLog(Log log):** The MoveWithLog() function handles the movement of the frog in conjunction with a log object. It takes a Log object as a parameter, representing the log that the frog is riding on. This function updates the frog's position based on the movement of the log, allowing the frog to move with the log as it traverses the game environment.

Each member function of the Frog class contributes to the behavior and functionality of the frog object in the Frogger game. The constructor initializes the frog's properties, while the other functions handle tasks such as moving the frog in different directions, drawing the frog on the screen, resetting its position, and handling movement in coordination with a log object. These functions provide the necessary logic for player-controlled movement, visual representation, and interaction with the game environment.

By utilizing these member functions, the Frog object can respond to player input, move within the game world, and interact with other game objects. The movement functions (MoveUp(), MoveDown(), MoveLeft(), and MoveRight()) allow the player to control the frog's direction and update its position accordingly. The Move() function provides flexibility for more customized movement scenarios.

- **The Draw()** function ensures that the frog sprite is rendered on the game screen, allowing players to see their frog's position and movement. This function utilizes the RenderWindow object stored in the window member variable to perform the drawing operation.

- **The Reset()** function is responsible for resetting the frog's position and state when the player loses a life or fails to complete a level. It repositions the frog to its initial position and sets the hasWon flag to false, preparing the frog for a new attempt.

- **The MoveWithLog()** function enables the frog to move in sync with a log object. By passing a Log object as a parameter, this function adjusts the frog's position based on the movement of the log, allowing the frog to ride along with it.

Overall, these member functions provide the essential functionality for controlling, rendering, and managing the frog object in the Frogger game. They enable player interaction, facilitate movement, and ensure proper synchronization with other game elements. Through these functions, the Frog object becomes an integral part of the game experience, allowing players to navigate obstacles and reach their objectives.

8. Main Function File

8.1 Main Function

8.1.1 Source code:

```
#include <iostream>
#include <SFML/System.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>
#include <SFML/Audio.hpp>
#include <SFML/Network.hpp>
#include <cstdlib>
#include <Windows.h>
#include "Frog.h"
#include "Global.h"
#include "Car.h"
#include "Log.h"

using namespace sf;
using namespace std;

// Function Prototypes
void StartScreen(RenderWindow& window);
void StartGame(RenderWindow& window);
void PlayerMovement(Frog& player, IntRect& texRect, int& keyCooldown, int&
keyTimer, Sound& sound);
void CarSpawner(int& carSpawnTimer, int& carSpawnCooldown, vector<Car>& cars);
void ObjectRemover(vector<Car>& cars);
void ObjectRemover(vector<Log>& logs);
void LogSpawner(int& logSpawnTimer, int& logSpawnCooldown, vector<Log>& logs);
bool GameOver(RenderWindow& window, bool hasWon);
```

```
int main()
{
    srand(time(0));
    // creating window
    RenderWindow window(VideoMode(800, 600), "OOP Project", Style::Default);
    window.setFramerateLimit(30);
    StartScreen(window);
    StartGame(window);
    return 0;
}
```

8.1.2 Documentation:

- **srand(time(0));:** This line seeds the random number generator with the current time. By calling srand() with time(0) as its argument, it ensures that each time the program is run, a different seed is used, resulting in different sequences of random numbers. This is often done to introduce randomness in the game or program.

- **RenderWindow window(VideoMode(800, 600), "OOP Project", Style::Default);:** This line creates a RenderWindow object named window. The RenderWindow class is provided by the SFML library and represents the game window or the graphical display for the game. It takes three parameters: VideoMode(800, 600) specifies the size of the window as 800 pixels wide and 600 pixels high, "OOP Project" sets the title of the window to "OOP Project", and Style::Default specifies the default style for the window.

- **window.setFramerateLimit(30);:** This line sets the maximum number of frames per second (FPS) for the game. By calling the setFramerateLimit() function on the window object and passing 30 as the argument, it ensures that the game runs at a maximum of 30 frames per second. This can be used to control the speed and smoothness of the game's animation.

- **StartScreen(window);:** This line calls the StartScreen() function and passes the window object as an argument. The StartScreen() function is responsible for displaying the initial start screen of the game. It may include elements such as a game title, instructions, or menu options for the player to interact with before starting the actual gameplay.

- **StartGame(window);:** This line calls the StartGame() function and passes the window object as an argument. The StartGame() function is responsible for initiating the actual gameplay. It may involve initializing game objects, setting up the game environment, and starting the game loop where the main game logic and rendering take place.

- **return 0;:** This line indicates the end of the main() function and returns the value 0 to the operating system. In C++, returning 0 from main() typically indicates a successful execution of the program.

In summary, the main() function sets up the game window, initializes the random number generator, calls the functions responsible for displaying the start screen and starting the game,

and then returns 0 to the operating system. It serves as the entry point for the program and orchestrates the overall flow of the game.

8.2 Start Screen Function

8.2.1 Source Code:

```
void StartScreen(RenderWindow& window)
{
        // Gameobjects Initialization
        Texture backgroundImage;
        Sprite backgroundSprite;
   if (!backgroundImage.loadFromFile("Resources/Images/StartScreen.png"))
      cout << "could not load main menu image" << endl;
        backgroundSprite.setTexture(backgroundImage);
   // Font initialization
   Font mainMenuFont;
   if (!mainMenuFont.loadFromFile("Resources/Fonts/Goldman-Regular.ttf"))
      cout << "could not load main menu font" << endl;
   Text startGameText;
   startGameText.setCharacterSize(50);
   startGameText.setFillColor(Color::White);
   startGameText.setFont(mainMenuFont);
   Text creditsText = startGameText;
   Text exitText = startGameText;

   // Text initialization
   startGameText.setString("Start Game");
   startGameText.setPosition((WINDOW_WIDTH -
startGameText.getGlobalBounds().width)/2, 300);
   creditsText.setString("Credits");
   creditsText.setPosition((WINDOW_WIDTH -
creditsText.getGlobalBounds().width)/2, 370);
   exitText.setString("Exit");
   exitText.setPosition((WINDOW_WIDTH - exitText.getGlobalBounds().width)/2,
440);

   // sounds initialization
   SoundBuffer menuSelectSoundBuffer;
   menuSelectSoundBuffer.loadFromFile("Resources/Sounds/gta-menu.wav");
   SoundBuffer menuMoveSoundBuffer;
   menuMoveSoundBuffer.loadFromFile("Resources/Sounds/gta-menu_2.wav");
   Sound;

   int choice = 0;
   int keyTimer = 0;
   int keyCooldown = 10;
```

```
        // Main game loop
    while (window.isOpen())
    {
                Event;
        while (window.pollEvent(event))
        {
                        if (event.type == Event::Closed)
                                window.close();
            if (Keyboard::isKeyPressed(Keyboard::Enter) && choice == 0)
            {
                sound.setBuffer(menuSelectSoundBuffer);
                sound.play();
                sleep(milliseconds(300));
                                return;
                        }
            else if (Keyboard::isKeyPressed(Keyboard::Enter) && choice == 1)
            {
                sound.setBuffer(menuSelectSoundBuffer);
                sound.play();
                                // credits
                        }
            else if (Keyboard::isKeyPressed(Keyboard::Enter) && choice == 2)
            {
                sound.setBuffer(menuSelectSoundBuffer);
                sound.play();
                sleep(milliseconds(300));
                                window.close();
                        }
                }
        // Menu selection
        if (Keyboard::isKeyPressed(Keyboard::Up) && keyTimer >= keyCooldown)
        {
            sound.setBuffer(menuMoveSoundBuffer);
            sound.play();
                        choice--;
                        if (choice < 0)
                                choice = 2;
            keyTimer = 0;
                }
        else if (Keyboard::isKeyPressed(Keyboard::Down) && keyTimer >=
keyCooldown)
        {
            sound.setBuffer(menuMoveSoundBuffer);
            sound.play();
            choice++;
            if(choice > 2)
                                choice = 0;
            keyTimer = 0;
        }
```

```
            keyTimer++;
            switch (choice)
            {
            case 0:
               startGameText.setFillColor(Color::Red);
               creditsText.setFillColor(Color::White);
               exitText.setFillColor(Color::White);
               break;
            case 1:
               startGameText.setFillColor(Color::White);
               creditsText.setFillColor(Color::Red);
               exitText.setFillColor(Color::White);
               break;
            case 2:
               startGameText.setFillColor(Color::White);
               creditsText.setFillColor(Color::White);
               exitText.setFillColor(Color::Red);
               break;
            }

                        // Clear
                        window.clear();
                        // Draw Stuff
                        window.draw(backgroundSprite);
            window.draw(startGameText);
            window.draw(creditsText);
            window.draw(exitText);
                        // Display
                        window.display();
               }
}
```

8.2.2 Documentation:

Description:

The StartScreen function is responsible for displaying the start screen of the game, including the background image, menu options, and handling user input for menu selection.

Parameters:

window (RenderWindow&): A reference to the game window (RenderWindow) object.

Function Body:

- Texture backgroundImage: Holds the texture for the background image.
- Sprite backgroundSprite: Represents the sprite that displays the background image.
- if (!backgroundImage.loadFromFile("Resources/Images/StartScreen.png")): Loads the background image from a file and checks if the loading was successful.

- Font mainMenuFont: Holds the font used for the menu text.
- if (!mainMenuFont.loadFromFile("Resources/Fonts/Goldman-Regular.ttf")): Loads the main menu font from a file and checks if the loading was successful.
- Text startGameText, creditsText, exitText: Text objects for the menu options.
- startGameText.setCharacterSize(50): Sets the character size of the "Start Game" menu text.
- startGameText.setFillColor(Color::White): Sets the fill color of the "Start Game" menu text.
- startGameText.setFont(mainMenuFont): Sets the font of the "Start Game" menu text.
- startGameText.setString("Start Game"): Sets the string of the "Start Game" menu text.
- startGameText.setPosition((WINDOW_WIDTH - startGameText.getGlobalBounds().width)/2, 300): Sets the position of the "Start Game" menu text to be horizontally centered and at a specific vertical position.
- creditsText.setString("Credits"): Sets the string of the "Credits" menu text.
- creditsText.setPosition((WINDOW_WIDTH - creditsText.getGlobalBounds().width)/2, 370): Sets the position of the "Credits" menu text to be horizontally centered and at a specific vertical position.
- exitText.setString("Exit"): Sets the string of the "Exit" menu text.
- exitText.setPosition((WINDOW_WIDTH - exitText.getGlobalBounds().width)/2, 440): Sets the position of the "Exit" menu text to be horizontally centered and at a specific vertical position.
- SoundBuffer menuSelectSoundBuffer, menuMoveSoundBuffer: Sound buffers for menu select and menu move sounds.
- menuSelectSoundBuffer.loadFromFile("Resources/Sounds/gta-menu.wav"): Loads the menu select sound buffer from a file.
- menuMoveSoundBuffer.loadFromFile("Resources/Sounds/gta-menu_2.wav"): Loads the menu move sound buffer from a file.
- Sound: Sound object for playing menu sounds.
- int choice = 0: Stores the current menu selection choice.
- int keyTimer = 0, keyCooldown = 10: Variables for handling key press timing and cooldown.
- while (window.isOpen()): Starts the main game loop.
- Event: Represents a single event that occurred within the game window.
- while (window.pollEvent(event)): Polls and processes events from the event queue.
- if (event.type == Event::Closed): Checks if the user closed the game window.
- if (Keyboard::isKeyPressed(Keyboard::Enter) && choice == 0): Checks if the user pressed the Enter key and the "Start Game" option is selected.
- sound.setBuffer(menuSelectSoundBuffer): Sets the sound buffer of the sound object to the menu select sound.
- `sound.play()
- sleep(milliseconds(300)): Pauses the program execution for 300 milliseconds to allow for a smoother transition.
- return;: Exits the StartScreen function and returns control to the caller.
- else if (Keyboard::isKeyPressed(Keyboard::Enter) && choice == 1): Checks if the user pressed the Enter key and the "Credits" option is selected.
- sound.setBuffer(menuSelectSoundBuffer): Sets the sound buffer of the sound object to the menu select sound.
- sound.play(): Plays the menu select sound.
- // credits: Placeholder for implementing the credits functionality.

- else if (Keyboard::isKeyPressed(Keyboard::Enter) && choice == 2): Checks if the user pressed the Enter key and the "Exit" option is selected.
- sound.setBuffer(menuSelectSoundBuffer): Sets the sound buffer of the sound object to the menu select sound.
- sound.play(): Plays the menu select sound.
- sleep(milliseconds(300)): Pauses the program execution for 300 milliseconds to allow for a smoother transition.
- window.close(): Closes the game window and terminates the program.
- if (Keyboard::isKeyPressed(Keyboard::Up) && keyTimer >= keyCooldown): Checks if the user pressed the Up arrow key and enough time has passed since the last key press.
- sound.setBuffer(menuMoveSoundBuffer): Sets the sound buffer of the sound object to the menu move sound.
- sound.play(): Plays the menu move sound.
- choice--: Decrements the value of choice to move the selection upwards.
- if (choice < 0): Checks if the value of choice is less than 0.
- choice = 2: Sets choice to 2 to wrap around to the last menu option.
- keyTimer = 0: Resets the key timer.
- else if (Keyboard::isKeyPressed(Keyboard::Down) && keyTimer >= keyCooldown): Checks if the user pressed the Down arrow key and enough time has passed since the last key press.
- sound.setBuffer(menuMoveSoundBuffer): Sets the sound buffer of the sound object to the menu move sound.
- sound.play(): Plays the menu move sound.
- choice++: Increments the value of choice to move the selection downwards.
- if(choice > 2): Checks if the value of choice is greater than 2.
- choice = 0: Sets choice to 0 to wrap around to the first menu option.
- keyTimer = 0: Resets the key timer.
- switch (choice): Evaluates the value of choice.
- case 0:: Executes the following code if choice is 0 (Start Game option).
- startGameText.setFillColor(Color::Red): Sets the fill color of the "Start Game" menu text to red.
- creditsText.setFillColor(Color::White): Sets the fill color of the "Credits" menu text to white.
- exitText.setFillColor(Color::White): Sets the fill color of the "Exit" menu text to white.
- case 1:: Executes the following code if choice is 1 (Credits option).
- startGameText.setFillColor(Color::White): Sets the fill color of the "Start Game" menu text to white.
- creditsText.setFillColor(Color::Red): Sets the fill color of the "Credits" menu text to red.
- exitText.setFillColor(Color::White): Sets the fill color of the "
- case 2:: Executes the following code if choice is 2 (Exit option).
- startGameText.setFillColor(Color::White): Sets the fill color of the "Start Game" menu text to white.
- creditsText.setFillColor(Color::White): Sets the fill color of the "Credits" menu text to white.
- exitText.setFillColor(Color::Red): Sets the fill color of the "Exit" menu text to red.
- window.clear(): Clears the contents of the game window.
- window.draw(backgroundSprite): Draws the background sprite onto the game window.
- window.draw(startGameText): Draws the "Start Game" menu text onto the game window.
- window.draw(creditsText): Draws the "Credits" menu text onto the game window.

- window.draw(exitText): Draws the "Exit" menu text onto the game window.
- window.display(): Displays the updated contents of the game window.
- The main game loop continues until the game window is closed.
- This function provides an interactive start screen where the user can select different options using the arrow keys and Enter key.
- The selected option is highlighted in red, and appropriate sounds are played for menu navigation and selection.
- The function handles user input events, updates the menu selection, and redraws the menu options on the game window.
- Once the user selects the "Start Game" option, the function returns, indicating that the game should start.
- If the user selects the "Credits" option, the credits functionality can be implemented.
- If the user selects the "Exit" option, the game window is closed and the program terminates.
- The function utilizes various resources such as textures, fonts, sounds, and images to create an immersive start screen experience.

8.3 Start Game Function

8.3.1 Source Code:

```
void StartGame(RenderWindow& window)

{

    bool isGameOver = false;

    // Gameobjects Initialization

    Frog player(window);

    Texture backgroundImage;

    Sprite backgroundSprite;

    backgroundImage.loadFromFile("Resources/Images/Background.png");

    backgroundSprite.setTexture(backgroundImage);


    // player movement support

    IntRect texRect(0, CELL_SIZE, CELL_SIZE, CELL_SIZE);

    int keyCooldown = 5;

    int keyTimer = 0;
```

```
// car spawner support
vector<Car> cars;
int carSpawnCooldown = 20;
int carSpawnTimer = 0;
// Log spawner support
vector<Log> logs;
int logSpawnCooldown = 20;
int logSpawnTimer = 0;

// sounds initialization
SoundBuffer moveSoundBuffer;
moveSoundBuffer.loadFromFile("Resources/Sounds/jump-sound2.wav");
Sound;
sound.setBuffer(moveSoundBuffer);

// Main game loop
while (window.isOpen())
{
    Event;
    while (window.pollEvent(event))
    {
        if (event.type == Event::Closed)
            window.close();
    }

    // Update
```

```
        // player movement control
        player.texRect = texRect;
        PlayerMovement(player, texRect, keyCooldown, keyTimer, sound);


        bool isOnLog = false;
        // Collison detection with logs
        for (int i = 0; i < logs.size(); i++)
        {
            if
(player.getSprite().getGlobalBounds().intersects(logs[i].getSprite().getGlobalBounds()))
            {
                player.MoveWithLog(logs[i]);
                isOnLog = true;
            }
        }
        // Collision detection with cars
        for (int i = 0; i < cars.size(); i++)
        {
            if
(player.getSprite().getGlobalBounds().intersects(cars[i].getSprite().getGlobalBounds()))
                isGameOver = true;
        }
        // Collision detection with water
        if (player.getSprite().getPosition().y < 280 && !isOnLog)
        {
            isGameOver = true;
```

```
}



// spawners
CarSpawner(carSpawnTimer, carSpawnCooldown, cars);
ObjectRemover(cars);
LogSpawner(logSpawnTimer, logSpawnCooldown, logs);
ObjectRemover(logs);
// Clear
window.clear();

// Draw Stuff
window.draw(backgroundSprite);
// drawing logs
for (int i = 0; i < logs.size(); i++)
{
   logs[i].Move();
   window.draw(logs[i].getSprite());
}
player.Draw();
// drawing cars
for (int i = 0; i < cars.size(); i++)
{
   cars[i].Move();
   window.draw(cars[i].getSprite());
}
```

```
        // Display
        window.display();
        if (isGameOver == true || player.hasWon)
        {
            if (GameOver(window, player.hasWon))
            {
                for (int i = logs.size() - 1; i >= 0; i--)
                {
                    logs.erase(logs.begin() + i);
                }
                for (int i = cars.size() - 1; i >= 0; i--)
                {
                    cars.erase(cars.begin() + i);
                }
                player.Reset();
                isGameOver = false;
            }
            else
            {
                break;
            }
        }
    }
}
```

8.3.1 Documentation:

The StartGame function serves as the entry point for the Frogger game and is responsible for initializing game objects, handling player input, updating game state, detecting collisions, rendering graphics, and managing game flow. This function encapsulates the core gameplay loop and ensures a smooth and interactive gaming experience for the players.

Parameters:

- **window:** A reference to the RenderWindow object on which the game graphics will be displayed.

Functionality:

Initialization:

The function begins by initializing various game objects and variables required for gameplay. These include loading the background image, creating the player's sprite, setting the initial position, and configuring other essential elements.

Main Game Loop:

The function enters a while loop, which acts as the game's main loop, and continues until the game window is closed or the game is over. Inside this loop, events are polled to handle user input and window events, ensuring a responsive and interactive gameplay experience.

Player Movement:

The function updates the player's movement based on user input. It listens for keyboard events and translates them into appropriate movement actions for the player character. For example, pressing the arrow keys moves the player up, down, left, or right on the game grid.

Collision Detection:

The function includes collision detection logic to check for interactions between the player and other game objects. It checks if the player's sprite collides with logs, cars, or water, determining the outcome of these collisions. If the player collides with a log, their movement is synchronized with the log's movement. However, colliding with a car or falling into the water results in the game ending.

Object Spawning and Removal:

The function manages the spawning and removal of game objects such as cars and logs. It uses timers and cooldowns to control the frequency at which these objects appear in the game world. Cars and logs are spawned at specific intervals and positions, creating dynamic and challenging

gameplay scenarios. Additionally, objects that move off-screen or are no longer relevant to the game are removed to optimize performance.

Rendering:

The function takes care of rendering the game graphics to the window. It starts by clearing the window to prepare for the next frame. Then, it renders the background image as the backdrop for the game scene. Next, it renders the logs and cars, updating their positions based on their movement patterns. Finally, it renders the player's sprite at its current position, reflecting any movement performed by the player.

Display and Game Flow:

After rendering all game elements, the function updates the display to show the changes. This ensures that players can see the updated game state with each iteration of the game loop. The function also includes logic to handle the game's flow, such as determining when the game is over and providing options for restarting or quitting the game. If the game ends, players are prompted with the appropriate messages and can choose to restart or exit the game.

By encapsulating the game logic within the StartGame function, the Frogger game achieves a modular and organized structure. This allows for easy maintenance, scalability, and extensibility of the codebase. Additionally, the function ensures a smooth and immersive gaming experience by efficiently handling user input, updating game state, detecting collisions, rendering graphics, and managing game flow.

8.4 Player Movement Function

8.4.1 Source Code:

```
void PlayerMovement(Frog& player, IntRect& texRect, int& keyCooldown, int& keyTimer, Sound& sound)

{


    if (Keyboard::isKeyPressed(Keyboard::Up) && keyTimer >= keyCooldown)

    {

        sound.play();

        texRect.top = 1 * CELL_SIZE;

        texRect.left = 1 * CELL_SIZE;
```

```
    player.texRect = texRect;
    texRect.left = 0 * CELL_SIZE;
    player.Move(0, -1);
    keyTimer = 0;
}
else if (Keyboard::isKeyPressed(Keyboard::Down) && keyTimer >= keyCooldown)
{
    sound.play();
    texRect.top = 3 * CELL_SIZE;
    texRect.left = 1 * CELL_SIZE;
    player.texRect = texRect;
    texRect.left = 0 * CELL_SIZE;
    player.Move(0, 1);
    keyTimer = 0;
}
else if (Keyboard::isKeyPressed(Keyboard::Left) && keyTimer >= keyCooldown)
{
    sound.play();
    texRect.top = 2 * CELL_SIZE;
    texRect.left = 1 * CELL_SIZE;
    player.texRect = texRect;
    texRect.left = 0 * CELL_SIZE;
    player.Move(-1, 0);
    keyTimer = 0;
}
else if (Keyboard::isKeyPressed(Keyboard::Right) && keyTimer >= keyCooldown)
```

```
    {
        sound.play();
        texRect.top = 0 * CELL_SIZE;
        texRect.left = 1 * CELL_SIZE;
        player.texRect = texRect;
        texRect.left = 0 * CELL_SIZE;
        player.Move(1, 0);
        keyTimer = 0;
    }
    else
    {
        keyTimer++;
    }
}
```

8.4.1 Documentation:

The PlayerMovement function is responsible for handling the player's movement in the Frogger game based on the user's input. It takes various parameters such as the player object, texture rectangle, key cooldown, key timer, and sound. Let's break down the functionality of this function:

Input Detection:

The function checks for user input by monitoring the state of the arrow keys (Up, Down, Left, Right) using the Keyboard::isKeyPressed function. It determines which key is pressed and executes the corresponding movement action for the player.

Texture Rect Update:

Depending on the movement direction, the function updates the texture rectangle (texRect) to ensure that the appropriate sprite frame is displayed for the player's movement animation. This allows for smooth and visually appealing transitions between different movement directions.

Player Movement:

Once the movement direction is determined, the function updates the player's position on the game grid. It uses the Move function of the player object to adjust the player's coordinates based on the specified movement values (-1, 0, 1) for the x and y axes. This ensures that the player moves in the desired direction.

Sound Feedback:

To provide audio feedback to the player, the function plays a sound effect using the provided Sound object. This sound effect indicates that the player's movement action has been successfully executed.

Key Cooldown and Timer:

The function manages the key cooldown and timer variables to prevent rapid and unintended movement. The keyCooldown represents the delay between consecutive movements, and the keyTimer keeps track of the elapsed time since the last movement. If the timer exceeds the cooldown value, the function allows for the detection of new key inputs and triggers the corresponding movement action.

By encapsulating the player's movement logic within the PlayerMovement function, the code achieves a modular and reusable structure. This function promotes clean code separation and enhances the overall maintainability and readability of the game implementation.

8.5 Car Spawner Function

8.5.1 Source Code:

```
void CarSpawner(int& carSpawnTimer, int& carSpawnCooldown, vector<Car>& cars)
{
    if (carSpawnTimer < carSpawnCooldown)
    {
        carSpawnTimer++;
        return;
    }
    carSpawnTimer = 0;
    int randomCarType = rand() % 6;
```

```
    int randomLane = (rand() % 4) + 4;

    cars.push_back(Car(randomCarType, randomLane));

}
```

8.5.2 Documentation:

The CarSpawner function is responsible for spawning cars in the Frogger game at regular intervals. It takes parameters such as the car spawn timer, car spawn cooldown, and a vector of cars. Let's break down the functionality of this function:

Timer and Cooldown Check:

The function checks if the car spawn timer (carSpawnTimer) has not reached the car spawn cooldown (carSpawnCooldown). If the timer is still below the cooldown value, it increments the timer and returns, indicating that it is not yet time to spawn a new car.

Car Spawn Trigger:

Once the car spawn timer reaches the car spawn cooldown, the function resets the timer to 0, indicating that it's time to spawn a new car.

Random Car Type and Lane:

The function generates random values to determine the type and lane of the newly spawned car. It uses the rand() function to generate a random number between 0 and 5 (rand() % 6) to represent the car type. The car type can correspond to different sprites or behaviors for visual variety in the game. It also generates a random lane number between 4 and 7 ((rand() % 4) + 4) to determine the lane in which the car will appear.

Car Creation:

Using the generated car type and lane values, the function creates a new Car object with the specified type and lane. The Car object is then added to the vector of cars (cars) using the push_back function. This vector serves as a container to keep track of all the active cars in the game.

By implementing the CarSpawner function, the code introduces a dynamic and evolving game environment by continuously spawning new cars at regular intervals. The randomization aspect adds variability to the car types and lanes, enhancing the gameplay experience. This function contributes to the overall challenge and excitement of Frogger as players navigate through a constantly changing traffic scenario.

8.6 Object Remover Function

8.6.1 Source Code:

```
void ObjectRemover(vector<Car>& cars)

{

   for (int i = 0; i < cars.size(); i++)

   {

      if ((cars[i].getSprite().getPosition().x > 900) || (cars[i].getSprite().getPosition().x + cars[i].getSprite().getGlobalBounds().width < 0))

      {

                        cars.erase(cars.begin() + i);

               }

         }

}
```

8.6.2 Documentation:

The ObjectRemover function is responsible for removing cars that have moved out of the game window in the Frogger game. It takes a vector of cars as a parameter and iterates through each car in the vector to check if it is outside the visible area. Let's examine the function in more detail:

Car Iteration:

The function uses a for loop to iterate through each car in the cars vector. The loop variable i represents the index of the current car being processed.

Out of Bounds Check:

For each car, the function checks if its x-coordinate is greater than the right edge of the game window (900) or if its x-coordinate plus its width is less than the left edge of the game window (0). This check determines if the car has moved completely out of the visible area.

Car Removal:

If the car is determined to be outside the visible area, the function removes it from the cars vector using the erase function. The erase function takes an iterator as an argument, which is obtained

by adding the current index i to the beginning iterator of the cars vector (cars.begin() + i). This effectively removes the car from the vector.

By implementing the ObjectRemover function, cars that have moved out of the game window are efficiently removed from the active cars collection. This ensures that the game remains optimized and avoids unnecessary calculations or rendering for cars that are no longer visible. The function helps maintain a clean and updated list of cars, improving the performance and overall gameplay experience of Frogger.

8.7 Log Spawner Function

8.7.1 Source Code:

```
void LogSpawner(int& logSpawnTimer, int& logSpawnCooldown, vector<Log>& logs)
{
   if (logSpawnTimer < logSpawnCooldown)
   {
                logSpawnTimer++;
                return;
        }
        logSpawnTimer = 0;
        int randomLogType = rand() % 3;
        int randomLane = rand() % 4;
   int tries = 0;
   // collision free spawning
   bool isIntersecting = true;
   if (logs.size() > 0)
   {
      for (int i = 0; i < logs.size(); i++)
      {
```

```
        if (Log(randomLogType,
randomLane).getSprite().getGlobalBounds().intersects(logs[i].getSprite().getGlobalBounds()))
        {
            if (randomLane == 3)
                randomLane = 0;
            else
                randomLane++;
            isIntersecting = true;
        }
        else
            isIntersecting = false;
      }
   }
   else
      logs.push_back(Log(randomLogType, randomLane));
   if (isIntersecting == false)
   {
      logs.push_back(Log(randomLogType, randomLane));
   }
}
```

8.7.2 Documentation

The LogSpawner function is responsible for spawning logs in the Frogger game. It takes a log spawn timer, a log spawn cooldown, and a vector of logs as parameters. The function determines when to spawn a new log based on the timer and cooldown values, and ensures that the spawned log does not intersect with existing logs. Let's break down the function in more detail:

Timer and Cooldown Check:
The function checks if the log spawn timer is less than the log spawn cooldown. If the timer hasn't reached the cooldown value, the function increments the timer and returns, indicating that it's not time to spawn a new log yet. This check ensures that logs are not spawned too frequently.

Reset Timer:
If the timer reaches the cooldown value, indicating that it's time to spawn a new log, the function resets the timer to 0, preparing for the next log spawn.

Random Log Type and Lane:
The function generates a random log type and a random lane for the new log to be spawned. The random log type is determined by taking the remainder of dividing a random number by 3. The random lane is determined by taking the remainder of dividing a random number by 4. These random values determine the characteristics of the log to be spawned.

Collision-Free Spawning:
To ensure that the newly spawned log does not intersect with existing logs, the function checks for collisions between the new log and each existing log in the logs vector. If an intersection is detected, the function increments the random lane value and sets the isIntersecting flag to true. This ensures that the new log is spawned in a different lane to avoid collisions with existing logs. The loop continues until a non-intersecting lane is found.

Spawning the Log:
Once a non-intersecting lane is determined, the function adds the new log to the logs vector using the push_back function. The log is created with the random log type and lane values.

By implementing the LogSpawner function, new logs are spawned at appropriate intervals while ensuring they do not intersect with existing logs. This helps maintain a safe and playable environment for the frog to traverse. The function adds an element of randomness and challenge to the game by dynamically spawning logs in different lanes and types.

8.8 Game Over Function

8.8.1 Source Code:

```
bool GameOver(RenderWindow& window, bool hasWon)

{

    // sound initialization

    SoundBuffer menuSelectSoundBuffer;

    menuSelectSoundBuffer.loadFromFile("Resources/Sounds/gta-menu.wav");

    SoundBuffer menuMoveSoundBuffer;

    menuMoveSoundBuffer.loadFromFile("Resources/Sounds/gta-menu_2.wav");

    SoundBuffer gameOverSoundBuffer;

    gameOverSoundBuffer.loadFromFile("Resources/Sounds/game-over.wav");

    SoundBuffer gameWonSoundBuffer;
```

```
    gameWonSoundBuffer.loadFromFile("Resources/Sounds/game-win.wav");
    Sound;

    if (hasWon)
    {
        sound.setBuffer(gameWonSoundBuffer);
        sound.play();
    }
    else
    {
        sound.setBuffer(gameOverSoundBuffer);
                    sound.play();
    }
    // Game Over background sprite initialization
    Texture backgroundTexture;
    backgroundTexture.loadFromFile("Resources/Images/Background.png");
    backgroundTexture.update(window);
    Sprite gameOverSprite;
    gameOverSprite.setTexture(backgroundTexture);

    // initializing game over rectangle
    RectangleShape gameOverRectangele;
    gameOverRectangele.setSize(Vector2f(500, 400));
    gameOverRectangele.setFillColor(Color(0, 0, 0, 200));
    gameOverRectangele.setPosition((WINDOW_WIDTH -
gameOverRectangele.getGlobalBounds().width) / 2, (WINDOW_HEIGHT -
gameOverRectangele.getGlobalBounds().height) / 2);
```

```
// font initialization
Font gameOverFont;
if (!gameOverFont.loadFromFile("Resources/Fonts/Goldman-Regular.ttf"))
    cout << "could not load game over menu font" << endl;
// game over menu button text initialization
Text retryText;
retryText.setCharacterSize(50);
retryText.setFillColor(Color::White);
retryText.setFont(gameOverFont);
Text exitText = retryText;

if (hasWon)
    retryText.setString("Play Again");
else
    retryText.setString("Retry");
retryText.setPosition((WINDOW_WIDTH - retryText.getGlobalBounds().width) / 2, 300);
exitText.setString("Exit");
exitText.setPosition((WINDOW_WIDTH - exitText.getGlobalBounds().width) / 2, 370);

// GameOver Text initialization
Text gameOverText;
gameOverText.setCharacterSize(80);
gameOverText.setFillColor(Color::Green);
gameOverText.setFont(gameOverFont);
if (hasWon)
```

```
        gameOverText.setString("You Won");
    else
        gameOverText.setString("Game Over");
    gameOverText.setPosition((WINDOW_WIDTH - gameOverText.getGlobalBounds().width) /
2, 130);


    int choice = 0;
    int keyTimer = 0;
    int keyCooldown = 10;


    while (true)
    {
        Event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
                window.close();
            if (Keyboard::isKeyPressed(Keyboard::Enter))
            {
                sound.setBuffer(menuSelectSoundBuffer);
                sound.play();
                sleep(milliseconds(300));
                if (choice == 0)
                    return true;
                if (choice == 1)
                    return false;
            }
```

```
}
// Menu selection
if (Keyboard::isKeyPressed(Keyboard::Up) && keyTimer >= keyCooldown)
{
    sound.setBuffer(menuMoveSoundBuffer);
    sound.play();
    choice--;
    if (choice < 0)
        choice = 1;
    keyTimer = 0;
}
else if (Keyboard::isKeyPressed(Keyboard::Down) && keyTimer >= keyCooldown)
{
    sound.setBuffer(menuMoveSoundBuffer);
    sound.play();
    choice++;
    if (choice > 1)
        choice = 0;
    keyTimer = 0;
}
keyTimer++;
switch (choice)
{
case 0:
    retryText.setFillColor(Color::Red);
    exitText.setFillColor(Color::White);
```

```
            break;
        case 1:
            retryText.setFillColor(Color::White);
            exitText.setFillColor(Color::Red);
            break;
        }

        // Draw
        window.draw(gameOverSprite);
        window.draw(gameOverRectangele);
        window.draw(gameOverText);
        window.draw(retryText);
        window.draw(exitText);
        // display
        window.display();
    }
}
```

8.6.2 Documentation

The GameOver function is responsible for handling the game over state in the Frogger game. It displays a game over screen with options for the player to retry or exit the game. Let's go through the function in more detail:

Sound Initialization:

The function initializes various sound buffers for menu selection, menu movement, game over, and game won sounds. It creates a Sound object to play the sounds based on the game outcome (won or lost).

Sound Playback:

Based on the hasWon parameter, the function selects the appropriate sound buffer and plays it using the Sound object. If the player has won, the game won sound is played; otherwise, the game over sound is played.

Game Over Background Sprite:

The function loads a background texture from an image file and creates a sprite using that texture. This sprite serves as the background for the game over screen.

Game Over Rectangle:

It creates a rectangle shape with dimensions 500x400 and sets its fill color to semi-transparent black. The rectangle is positioned at the center of the window, creating a backdrop for the game over text and buttons.

Font Initialization:

The function loads a font from a file and checks if it was successfully loaded. If not, it outputs an error message.

Game Over Menu Button Text:

It initializes two text objects: retryText and exitText. These texts represent the retry and exit buttons displayed on the game over screen. The font, character size, fill color, and positions of the text objects are set accordingly. The retryText displays "Play Again" if the player has won and "Retry" if the player has lost.

Game Over Text:

It initializes a text object named gameOverText to display the game over message. The character size, fill color, font, and position of the text are set accordingly. The message displayed depends on the value of hasWon.

Menu Selection:

The function enters a while loop that handles the game over menu interaction. It polls for events and checks for the Enter key press to confirm the player's choice. The Up and Down arrow keys allow the player to navigate between the retry and exit buttons. The current choice is highlighted by changing the fill color of the corresponding text.

Return Values:

Based on the player's choice, the function returns true if the player wants to retry the game and false if the player wants to exit the game.

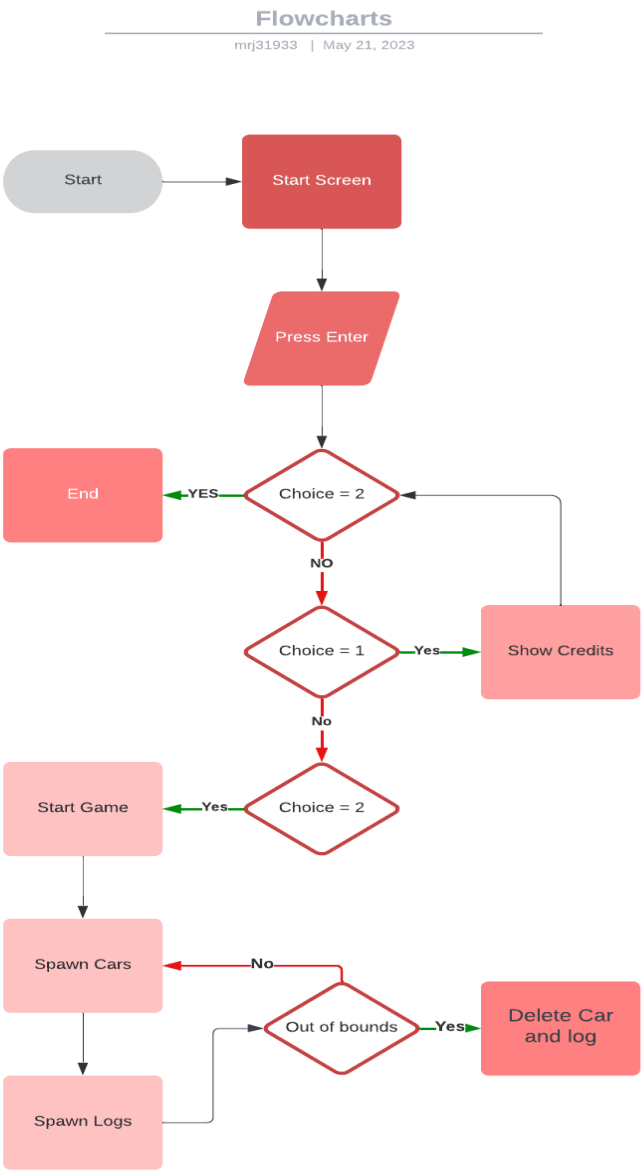
Draw and Display:

Within the while loop, the game over sprite, rectangle, game over text, retry text, and exit text are drawn on the window. Finally, the window is displayed to update the screen.

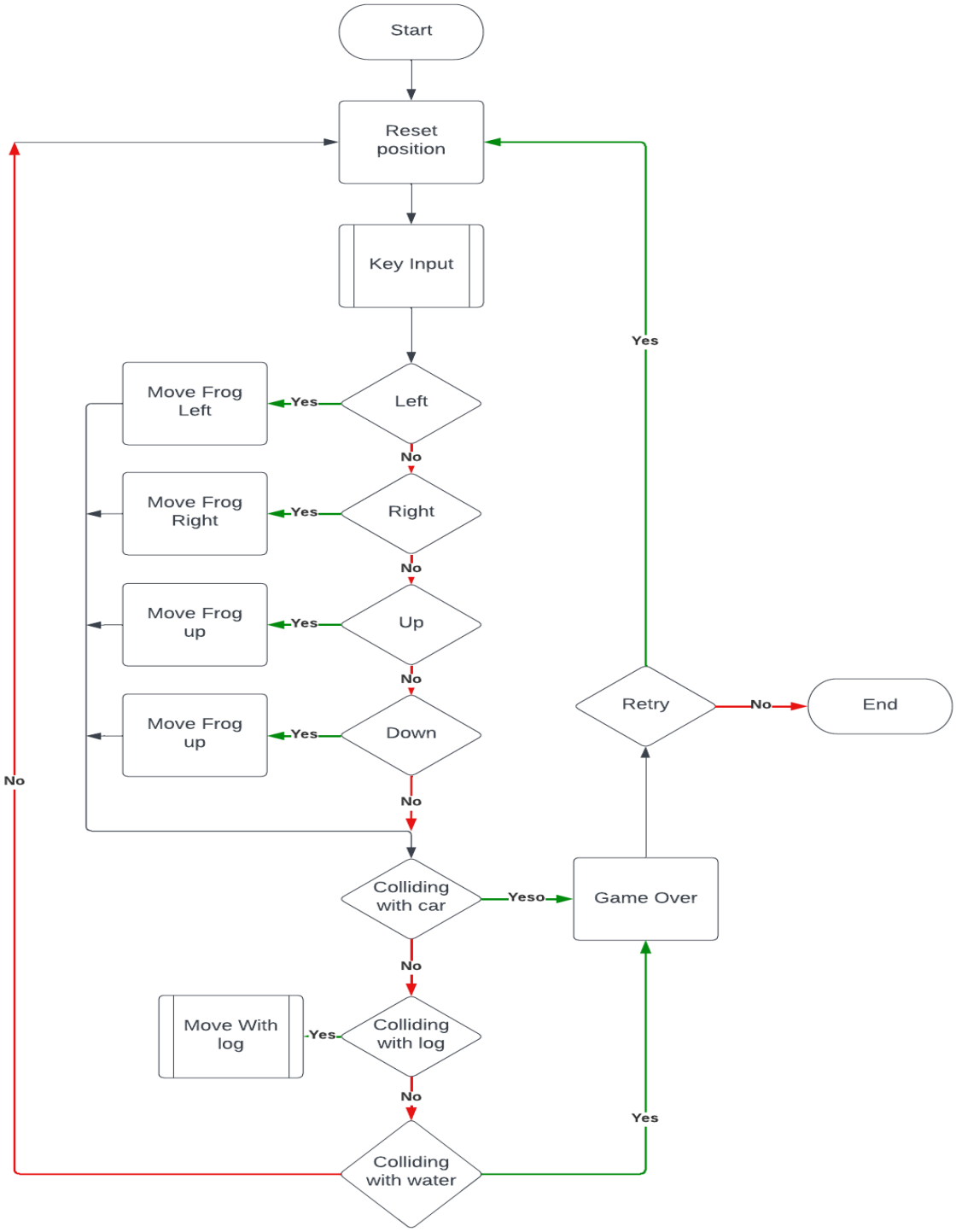
By implementing the GameOver function, the game over state is handled with appropriate sounds, visuals, and menu options. It provides the player with a choice to retry or exit the game, allowing for a seamless gameplay experience.

9. Activity Diagrams

9.1 Start Screen

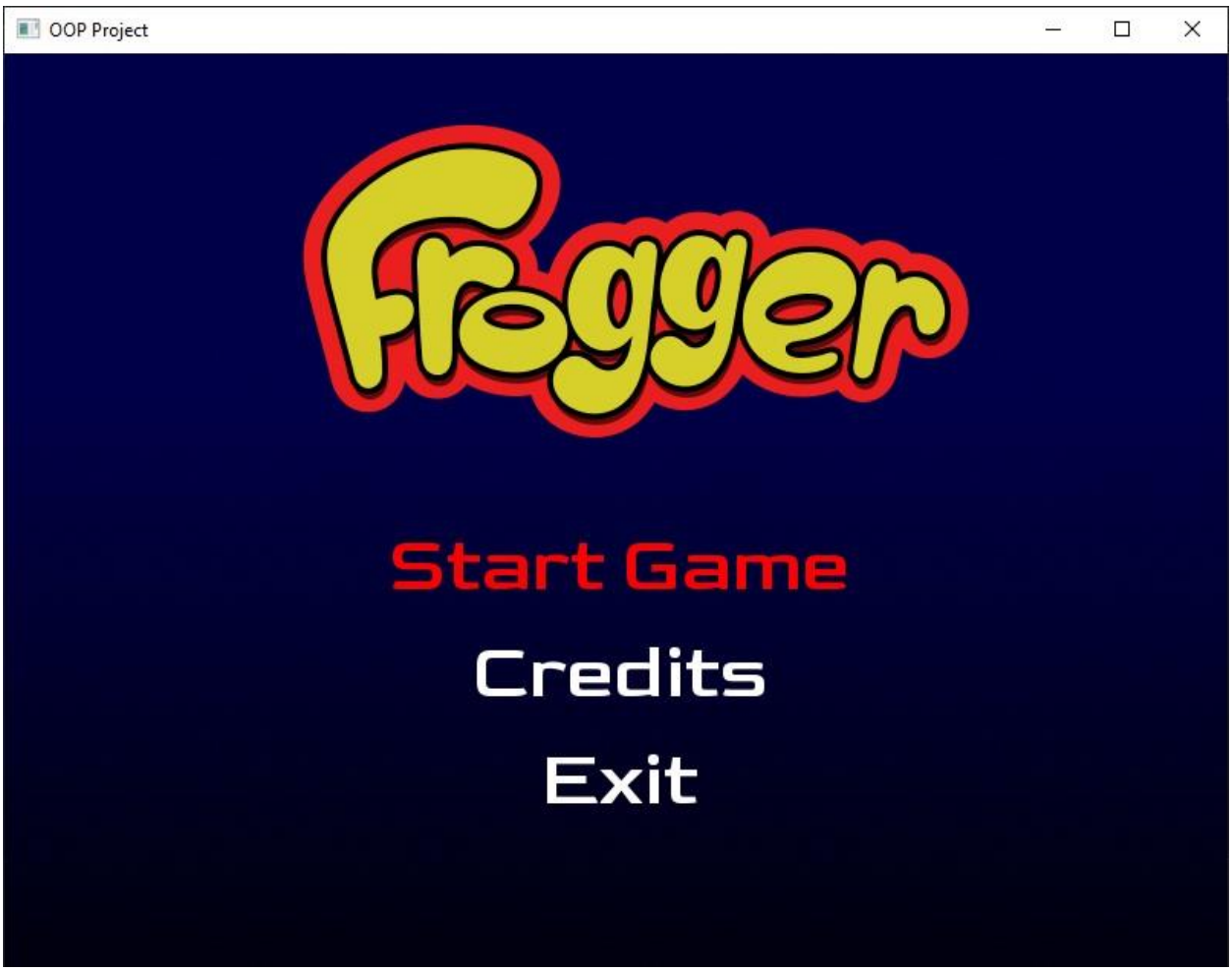


9.2 Gameplay


Start
Reset position
Key Input
Left
Move Frog Left
Right
Move Frog Right
Up
Move Frog up
Down
Move Frog up
Colliding with car
Game Over
Retry
End
Move With log
Colliding with log
Colliding with water
Yes
No
Yeso

10. Screenshots

10.1 Start Screen



10.2 Gameplay

10.3 Game Won

10.4 Game Over